HAW HAMBURG

Masterarbeit

Yegor Yefremov

A Hardware-in-the-Loop Test Module for UART and its Integration into the RIOT Ecosystem

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Yegor Yefremov

A Hardware-in-the-Loop Test Module for UART and its Integration into the RIOT Ecosystem

Mastertarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 12. April 2019

Yegor Yefremov

Thema der Arbeit

A Hardware-in-the-Loop Test Module for UART and its Integration into the RIOT Ecosystem

Stichworte

CI, git, RIOT, UART, IoT, Robot Framework

Kurzzusammenfassung

Hardware-in-the-Loop (HIL) Tests stellen sicher, dass die Kombination von Software und Hardware wie angegeben funktioniert. In dieser Arbeit analysieren wir die Struktur von einem Universal Asynchronous Receiver Transmitter (UART), erweitern seine API in RIOT OS und entwickeln eine Testsuite für die aktuelle und erweiterte Funktionalität. Das Endergebnis dieser Arbeit war eine funktionierende HIL-Simulation für UART, die in den CI-Prozess von RIOT integriert wurde.

Yegor Yefremov

Title of Thesis

A Hardware-in-the-Loop Test Module for UART and its Integration into the RIOT Ecosystem

Keywords

CI, git, RIOT, UART, IoT, Robot Framework

Abstract

Hardware-in-the-Loop (HIL) testing ensures that the combination of software and hardware works as specified. In this thesis, we analyze Universal Asynchronous Receiver Transmitter (UART) structure, extend its API in RIOT OS and develop a test suite for the current and extended functionality. The final result of this thesis was a working HIL simulation for UART integrated into RIOT's CI process.

Contents

List of Figures

List of Tables

Acronyms

CI Continuous Integration.

CTS Clear To Send.

DUT Device under Test.

FIFO First-In-First-Out.

HIL Hardware-in-the-Loop.

IC integrated circuit.

IoT Internet of things.

LLMemMapIf Low Level Memory Map Interface.

OS operating system.

PHiLIP Primitive Hardware In the Loop Integration Product.

RTS Request To Send.

TDD Test Driven Development.

UART Universal Asynchronous Receiver Transmitter.

1 Introduction

RIOT is an open source operating system for Internet of things (IoT) and other embedded devices [7], [6]. It provides support for more than a hundred boards and new ones being added constantly. Meanwhile, the project surpassed 20000 commits and to keep projects reliability at such a quick development pace, one needs a suitable testing ecosystem. RIOT currently relies on tests like static code analyzes performed by Travis-CI and build and unit tests executed by custom distribution load worker called Murdock.

The RIOT project allows developers to write a program for an 8-bit platform like for example Arduino Mega 2560 and then reuse it on 16-bit or 32-bit platforms. This heterogeneity of IoT hardware imposes some constraints on the testing. Some tests can be performed in software using device emulators or simulators. However, in order to ensure that the combination of software and hardware works as specified, system tests must be performed on real devices [25]. In addition, it is very unlikely that this ever-growing amount of devices can be owned and centralized in one testing rack.

PHiLIP project was developed as a low-cost and flexible Hardware-in-the-Loop (HIL) simulator. It provides a qualified firmware simulating slaves for a number of interfaces like I2C, UART, SPI and can be used both for automated regression testing and development/debugging needs. Together with Murdock's distributed properties and PHiLIP's affordability any institution or even persons can contribute to better test coverage as the devices do not have to be concentrated and owned at one place.

Though COM connectors disappear from modern personal computers, the UARTs are still widely used: Various wireless technologies like Bluetooth, LoRa use them as an interface to a CPU. Also such fieldbus standards as Modbus, BACnet use serial communication as a physical layer.

The current UART API in RIOT allows only basic functions sufficient for a shell or a simple serial communication with standard settings like 8-bit, no parity and one stop bit (8-N-1). Serial frame configuration is a feature shared by all UARTs supported in RIOT.

In the current implementation, such changes cannot be made from a user application and require custom changes on the platform driver. Therefore, extending the UART API with a serial frame configuration increases UART usability in RIOT and opens it to more protocols requiring other settings than 8-N-1.

In this work we analyze the UART structure and its API in RIOT and develop a HIL module for RIOT's CI infrastructure. Following contributions were made:

1. Extend the UART API so that it is possible to alter serial frame configuration (number of data bits, parity and the number of stop bits).
2. Develop a UART slave for the HIL testing based on the PHiLIP project.
3. Develop a test suite for the UART and integrate it into the RIOT's CI environment.

The thesis has the following structure. The next Section summarizes the related works. Chapter 2 on page 4 introduces CI in the embedded systems development process and how it is implemented in RIOT. Chapter 3 on page 8 shows current HIL testing approach used in RIOT. UART is described in Chapter 4 on page 14. Chapter 5 on page 22 shows how UART API is extended. Chapter 6 on page 27 describes UART test integration into RIOT's CI process. Tests evaluation is presented in Chapter 7 on page 43. And finally Chapter 8 on page 50 provides conclusion and outlook.

1.1 Related Work

The introduction and main postulates of the Continuous Integration (CI) were presented in an article of Martin Fowler [12]. [16] shows how such techniques as Test Driven Development (TDD) and Continuous Integration that are widely used in the software development process can be applied to the embedded software development process. [36] gives an overview of the peculiarities in an automotive deployment pipeline.

The basics of Universal Asynchronous Receiver Transmitter (UART), its usage and protocols were described in [5]. Also, UART's data sheets and application notes provide in-depth insight into integrated circuits of various vendors [9], [30], [28], [17], [14], [15], [4], [35], [34], [27], [31]. [8] covers the causes of serial communication timing issues.

A Hardware-in-the-Loop (HIL) approach in general is described in [19]. [3] shows HIL validation of an over-current relay. [26] describes testing method for embedded control

systems. The HIL simulation of a cardiovascular system is described in [13]. [22] gives an overview of the HIL systems for Powertrain control system software verification and validation.

Robot Framework [11] usage in software automated testing will be shown in [18], while [32] uses Robot Framework in an embedded environment.

2 Continuous Integration in Embedded Systems Development Process

This chapter introduces Continuous Integration, different testing types as also their usage in embedded systems developments process. Also current CI process in the RIOT project will be described.

2.1 Continuous Integration Overview

Continuous Integration (CI) is a development practice used in software engineering that requires developers to integrate code into a shared repository several times a day. Each integration will be verified by an automated build that allows an early error detection.

Basically, a CI process goes through the following stages: unit, integration, and system testing [1].

The unit tests ensure that specific units and components of the software are fully functional. These components will be tested isolated from each other. The aim of this stage is to check that each component correctly implements its design and is ready to be integrated into a system of components.

Now that the separate components are tested, they will be integrated and checked whether they can work together in a system. The integration tests are designed to find defects on the interface level between the modules/functions.

The system tests analyze the application as a whole to verify that all its acquirement are met.

Embedded systems or as they are sometimes called cyber-physical systems have a unique characteristic that they directly interact with the physical world. Therefore, they need inputs and outputs that cannot always be generated or checked easily when testing [36].

Providing such an environment for the automated CI process is not always possible or feasible. To avoid the necessity of manipulating hardware the embedded systems will be tested in simulated environments. Sensors and actuators will be replaced with a computer that provides the exact signals the sensor would send under the desired physical condition. The same applies to the actors where a computer is connected and translates the outgoing signals to a hypothetical physical action. This technique is called Hardware-in-the-Loop (HIL) simulation. It provides system-level testing of embedded systems in a comprehensive, cost-effective, and repeatable manner [19]. Figure 2.1 demonstrates a high-level view of an example HIL simulation where an embedded system or System under Test (SUT) produces input for the real-time simulation and consumes its output in the form of sensor data or operator commands.

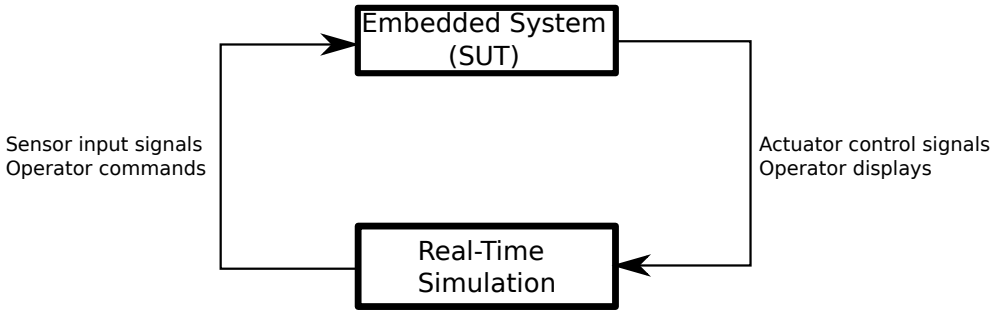


Figure 2.1: HIL Example [19]

Figure 2.2 shows a basic CI process tailored for embedded systems and containing all the above-mentioned test types.

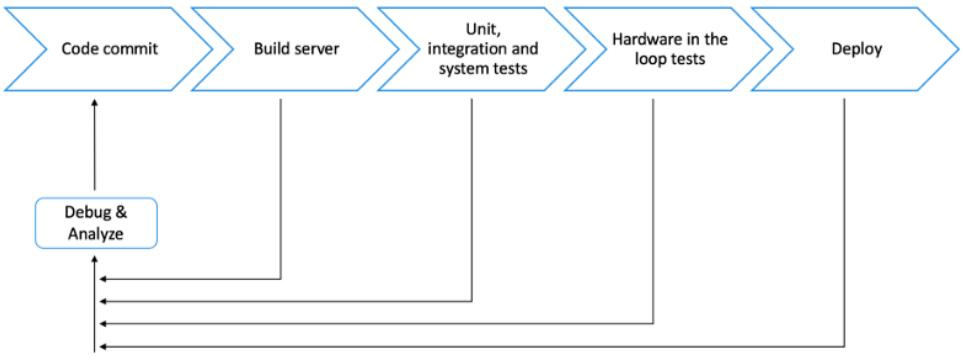


Figure 2.2: Continuous Integration Process for Embedded Systems[23]

This work focuses on the peripheral driver testing. Therefore, to embed the Device under Test (DUT) into a HIL testing, a computer providing all necessary interfaces is needed.

Normally such a host does not provide all these interfaces or the low-level access and special functionality are not available. Adding such interfaces through for example USB converter would affect the timing accuracy. [21] suggests to use a mocking hardware for the HIL testing. The usage of a commercial testing hardware or the development of a custom hardware means a considerable investment. Provided, such a device can be reused for multiple projects or where the cost of human/manual testing is very high, or a high level of reliability is an absolute must, the costs are justified.

Figure 2.3 demonstrates a setup where CI Server, on the one hand, uploads firmware through a JTAG interface to the DUT and executes tests using UART and Ethernet interfaces, on the other, controls a Testing Coprocessor simulating GPIO and SPI slaves over the USB interface.

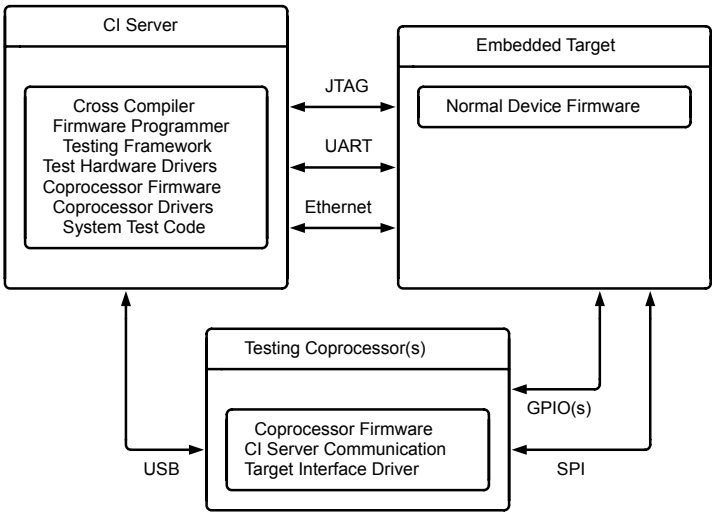


Figure 2.3: HIL Testing with a Mocking Hardware [21]

2.2 RIOT Development Process

The RIOT project is hosted on GitHub platform and takes advantage of its services like version control, issue tracker, wiki as also integration with Travis CI service. According to RIOT's documentation [1], the following steps are needed in order to get the developer's contribution merged:

[1] https://github.com/RIOT-OS/RIOT/wiki/Contributing-to-RIOT

1. Fork the RIOT git repository (if you have not done this already).
2. Create a branch.
3. Make commits.
4. Make sure your code is in compliance with RIOTs coding conventions.
5. Push this branch to your fork on GitHub.
6. Do a pull request (Use the labels).
7. Other RIOT members will provide feedback.
8. Address this feedback.
9. Your code is merged in RIOT master branch.

A pull request triggers Travis CI to perform a number of tests including static source code analysis like cppcheck and Coccinelle, Python style guide enforcement checks like flake8 etc.

As soon as the maintainers see the PR mature for build tests, a special label "CI: ready for build" will be applied. This step invokes the Murdock server. Murdock is a simple CI (continuous integration) server written in Python developed specifically for RIOT OS. It will be used both for pull request validation as also for the nightly builds testing. Basically, Murdock will run a number of static tests i.e. build tests, static code analysis tests, etc. But it can also perform tests on a native port and a number of registered boards (samr21-xpro was the only available board at the time of writing). For now, only the tests that do not require externally attached hardware can be run.

GitHub invokes Murdock via a webhook feature to perform CI tasks, using disque based work queue (dwq) Murdock distributes the tasks to available dwq worker slaves running inside Docker containers.

The heart of the Murdock is disque based work queue. Dwq is a tool that can be used to distribute jobs on git repositories across multiple machines. So Murdock master mode distributes CI tasks based on per board/app basis to the registered dwq worker slaves reducing the duration of CI session.

The results from all used worker slaves are combined and provided back to the GitHub.

3 Test Platform for HIL in RIOT

3.1 PHiLIP

The Primitive Hardware In the Loop Integration Product (PHiLIP) project [37] is a combination of hardware and software that RIOT will use in a CI process but is not limited to it. The project uses affordable hardware like Blue Pill (STM32F103C8) and STM32 Nucleo-64 development (STM32F103RB) boards. The Blue Pill boards can be purchased for less than $2. This makes it usable for a lot of developers needing a tool with qualified firmware to test their embedded systems at various development stages. Figure 3.1 a shows Blue Pill board mounted on top of a Raspberry Pi using a specially designed adapter board (the green layer between Blue Pill and Raspberry Pi). Together these devices form a HIL node for RIOT's CI.

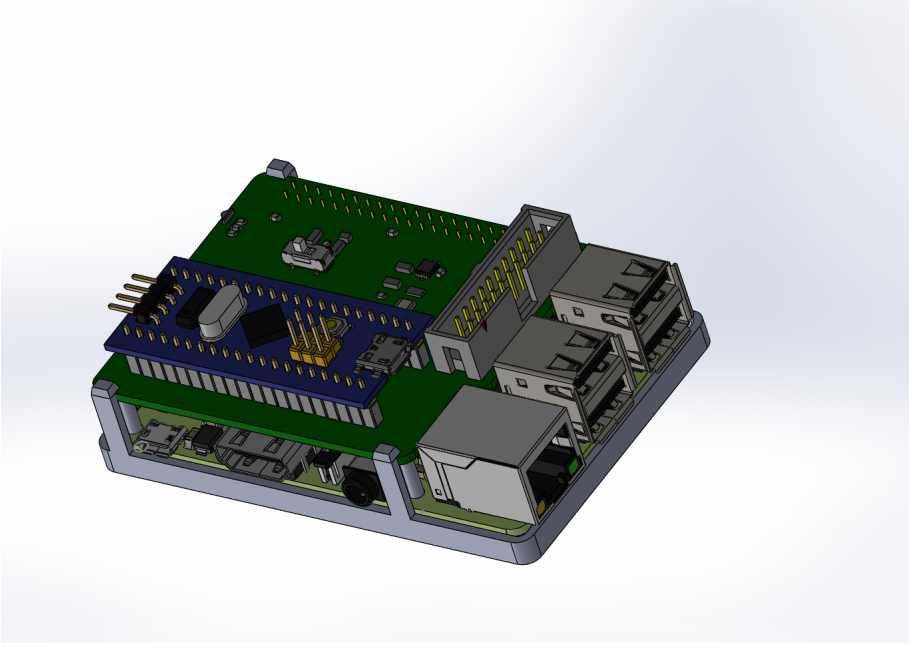


Figure 3.1: HIL Node for CI

An MCU solution was chosen because specialized testing hardware is expensive and has proprietary firmware, so that not every required scenario can be covered. Besides MCU provides low-level hardware access enabling exact test control. For example, NRT's NanoBoard supports only I2C, SPI, GPIO, and analog signals and already costs $59 [33]. Some related interfaces like UART, CAN, etc. are missing and the API does not provide access to the controller's registers.

PHiLIP covers such interfaces as I2C, UART, GPIO, SPI etc. Figure 3.2 shows Blue Pill board pin configuration. Serial interface UART (TX1, RX1) is used to control the firmware.

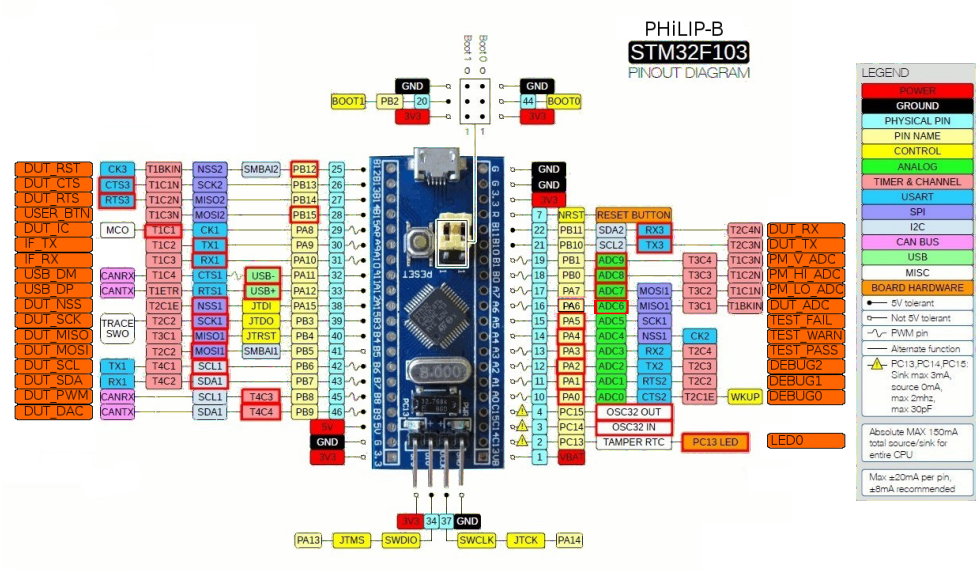


Figure 3.2: PHiLIP-B Pinout [37]

PHiLIP implements a set of virtual registers. One set of registers controls slave behavior and also gathers statistics like sent/received bytes or packets, error flags, a slave state, etc. This way the user can detect and analyze DUT issues. The other set of registers lets both the user and DUT read and write arbitrary data and simulate real slave devices.

3.1.1 Firmware Controlling Protocol

PHiLIP will be controlled using a simple ASCII protocol. The protocol supports four commands shown in Table 3.1 on the following page allowing the user to read and write

registers of the memory map and reset PHiLIP's MCU. These commands can be typed in a terminal software like HyperTerminal or minicom to manually control PHiLIP.

Writing to a register will only alter the memory map. To apply settings, for example, to change UART baud rate one needs to issue ex command after changing the related register.

Command	Parameter	Description
rr	index, size	Read register
wr	index, data, size	Write to register
ex	n.a.	Commit changes
mcu_rst	n.a.	Reset MCU

Table 3.1: PHiLIP Protocol

3.1.2 RIOT PAL

riot_pal is a Python package abstracting away and unifying shell based commands in RIOT and for bare metal memory map access. This way both DUT and PHiLIP can be controlled using the same interface.

Low Level Memory Map Interface (LLMemMapIf) is the core component interfacing with a memory map. It parses the map from a CSV file and allows fine-grained access to the registers compared to ASCII protocol. In ASCII protocol each register is addressed with a numerical offset. riot_pal allows hierarchical symbolic register names comparable with the structure field access in C programming language. For example, reading "uart.mode" register will access mode field of the uart structure in PHiLIP. The interface provides the following basic functions:

- read_struct - reads a set of registers defined by the memory map.
- read_reg - read a register defined by the memory map.
- write_reg - writes a register defined by the memory map.

3.2 Test Infrastructure in RIOT

RIOT's peripheral tests are collected under `tests` folder prefixed with `periph_`. Such tests provide a small application that will be flashed onto DUT. This program provides a shell interface allowing the user to initialize the peripheral in question and perform actions according to the API. All interactions run via the default UART. For example `main.c` from `periph_spi` allows the user to setup SPI modes, frequency etc. and transfer data to some slave.

UART peripheral test application implements the following commands:

- `init` - initialize a UART device with a given baud rate.
- `send` - send a string through given UART device.

The incoming data will be stored in a buffer associated with a given UART and as soon as character `\n` will be received the content of the buffer will be printed.

3.2.1 Robot Framework

Robot Framework is a generic test automation framework written in Python. It makes use of tabular test data syntax and utilizes the keyword-driven testing approach. The keywords act as programming functions or methods. The users can use the keywords provided by the standard libraries to create their own higher-level keywords.

This approach is demonstrated in Listing 3.1 taken from the official Robot Framework documentation [11]. The "Settings" section provides test suite description and also includes additional higher-level keywords from `resource.txt` file. The test case "Valid Login" performs the followings steps: opens a login page in a browser, inputs demo and mode as login credentials, submits them and checks whether the page could be opened with the submitted credentials. In the end, the browser is closed. A keyword, for example, "Submit Credentials" has only one character spaces between the words and corresponds to a Python function `submit_credentials`. A parameter is provided using a tabular like in user name input keyword.

```
*** Settings ***
Documentation     A test suite with a single test for valid login.

                  This test has a workflow that is created using keywords in
```

```
                  the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
        Open Browser To Login Page
        Input Username    demo
        Input Password    mode
        Submit Credentials
        Welcome Page Should Be Open
        [Teardown]    Close Browser
```

Listing 3.1: Robot Framework Example [11]

Robot framework is independent of operating system and application. Figure 3.3 shows the framework architecture as it is described in its documentation. The framework receives the test data and utilizes test libraries to communicate with the system under test [32]. Usually, this communication is direct but some testing libraries require lower level test tools as drivers. In the case of RIOT, a test library relies on RIOT Protocol Abstraction Layer as a driver to handle communication with both PHiLIP and the DUT.

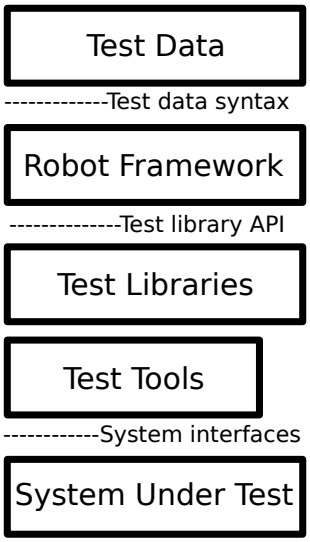


Figure 3.3: Robot Framework Architecture [11]

Robot Framework generates thorough reports and logs in XML format and hence, can be easily integrated into web-based CI systems.

RobotFW-tests project maintains various HIL tests [20]. It uses git submodules to integrate RIOT source tree and so provide a full build environment.

Folder `dist/robotframework/res` provides keywords for common modules used by all tests:

- `api_shell.keywords.txt` - keywords for shell API calls.
- `philip.keywords.txt` - keywords to reset PHiLIP and DUT.
- `riot_base.keywords.txt` - keywords to reset RIOT application.

Folder `dist/robotframework/lib` provides a Python module `PhilipAPI.py`. This module abstracts the communication with PHiLIP using RIOT PAL.

A test will be invoked using make command:

```
BOARD=<name> make -C tests/<test name> flash robot-test
```

4 UART

4.1 UART Overview

The Universal Asynchronous Receiver Transmitter (UART) is an integrated circuit (IC) providing asynchronous serial communication capabilities. Universal designation stands for configurability of both data format and transmission speeds. Also the actual electric signaling levels and methods (like differential signaling etc.) typically are handled by a special driver circuit external to the UART [2].

UART internal structure consists of following elements: transmitter, receiver, baud rate generator and a First-In-First-Out (FIFO). Figure 4.1 on the next page shows such a structure based on the Exar's ST16C550 UART. The transmitter converts parallel data into a stream of bits, which the receiver then samples and converts into parallel data. This conversion is performed by a shift register. The FIFO is a special buffer used both as a transmitter and receiver that stores high speed incoming data to prevent data loss and increase throughput for the outgoing data. Baud rate generator provides clock to transmitter, receiver and FIFO.

In UART communication two devices are interconnected by two pins RX (receive) and TX (transmit). RX pin of one device is connected to TX pin of the other and vice versa. This way a bidirectional communication is possible. Two UARTs can be interconnected directly using TTL signals, for a communication with devices in a field physical standards like RS-232, RS-422, RS-485 and others are used.

Transmission timing UARTs communicate with each other using the asynchronous methodology. Instead of using a clock signal to indicate the beginning and the end of a byte, start and stop bits are used. Hence, both sides must use the same baud rate because a large deviation will cause wrong bit sampling. To allow larger frequency deviation many

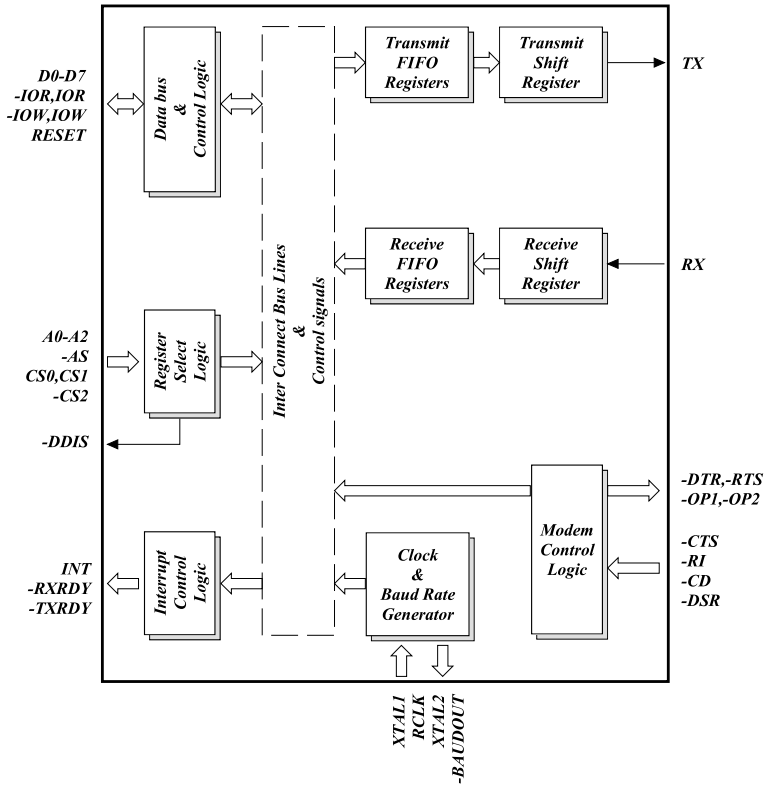


Figure 4.1: ST16C550 UART Internal Structure [9]

UARTs over-sample the incoming data at 16 times the baud rate to detect the near center of the start bit [28].

Aside from baud rate setting issues, there are also other factors that can affect the timing of a UART transmission: send jitter problem, baud rate drift, clock offset and signal runtime [8]. But as the software UART drivers do not influence these issues, this section concentrates on the arithmetic error in baud rate setting.

Standalone UARTs like 8250 and its derivatives use an external clock source. So normally it is chosen so that it is possible to get the standard baud rates like 50, 300, 600, 2400, 4800, 9600, 19.2 k, 38.4 k, 57.6 k and 115.2 k exactly.

Following example shows how a baud rate can be programmed into a ST16C550 UART with a crystal or oscillator running at 14.7456 MHz. This UART uses a 16-bit divisor together with the over-sampling rate to divide the clock and get the required baud rate. Two registers DLM and DLL hold the most and less significant bytes of the divisor, respectively. Equation (4.1) on the following page calculates the baud rate from UART

clock, over-sampling rate and divisor. Table 4.1 on the next page shows the obtained baud rate divisors, actual baud rates and an error between a desired baud rate and actual baud rate. Because of the specifically selected clock source, the error is zero.

$$baud\ rate = \frac{clock}{16 \times divsor} \tag{4.1}$$

As for MCUs, the situation is different because UART is an integral part of the MCU itself and thus uses one of the system clocks. To show the difference between standalone and integrated UARTs, the baud rate programming is performed for STM32 family UARTs.

These UARTs have a more flexible configuration. The divisor value programmed into the baud rate register USART_BRR has an integer part 12-bits and a fractional part 4-bits. The over-sampling rate is configurable and allows both 8x and 16x rates. The OVER8 bit controls the UART to sample with 8x over-sampling rate if set and 16x if cleared. A prescaler specifies the clock for the UART. According to the data sheet it ranges from 8 to 84 MHz. This example takes 8 MHz clock to show the calculation error. Equation (4.2) will be used to calculate a baud rate for STM32. Hence setting OVER8 to 0 converts the formula to the same equation as for ST16C550, but the problematic part is splitting the divisor into mantissa and a fraction.

$$baud\ rate = \frac{clock}{8 \times (2 - OVER8) \times divisor} \tag{4.2}$$

Calculation error is demonstrated by determining divisor for 115200 b/s:

1. Using the modified equation (4.1) and rounding to two digits after a decimal point a divisor for 115200 b/s is 4.34
2. Fractional part for the USART_BRR register is calculated as $16 \times 4.34 = 5.44$. After rounding it becomes 5. So USART_BRR value is 0x45
3. To determine the actual divisor the USART_BRR provides the fractional part and is calculate as follows: $12 : 5 = 0.3125$. In the end the divisor used by the UART equals 4.3125 instead of 4.34 resulting in an error of $\frac{115942 \times 100}{115200} = 0.64\%$

Table 4.2 on the following page taken from [30] shows baud rate calculation errors for the same standard baud rates using the above mentioned approach. As can be seen from the table the resulting baud rates differ from the desired ones.

Desired baud rate	Divisor	Actual baud rate	% Error
115200	8	115200	0
230400	16	230400	0
460800	32	460800	0

Table 4.1: Baud rate table for ST16C550

Desired baud rate	Divisor	Actual baud rate	% Error
115200	4.3125	115942	0.64
230400	2.1875	228571	0.79
460800	1.0625	470588	2.12

Table 4.2: Baud rate table for STM32

In general, a UART in 16x oversampling mode allows about ±5 % of a baud rate deviation considering a byte consisting of a start bit, 8 data bits and one stop bit [28]. But depending on the implementation each UART model and even its revision can have its own tolerance when receiving a character.

For example according to [28] SC16CXXXB UARTs with date code prior to 0443 tolerate baud rate deviation of -0.5 % if the incoming baud rate is less than programmed one, and 5.6 % if the incoming baud rate is more than programmed one. Later revisions have increased their tolerance for slower baud rates to -4.4 %.

STM32 UARTs tolerate deviation from 3.03% to 4.375% depending on the selected mode.

Data framing UART wraps each character into a packet when sending on the wire. Table 4.3 shows a serial frame and its components. It begins with a start bit and ends with one or two stop bits. Between these bits resides a character itself and optionally a parity bit. The receive UART samples this packet and checks if every bit can be detected according to the programmed settings. If a stop bit cannot be detected at its fixed position then a framing error is detected and a related status bit is set.

1 start bit	5 to 9 data bits	0 to 1 parity bits	1 to 2 stop bits

Table 4.3: UART Packet

Parity The line noise can corrupt transmitted bytes. To detect this, the UARTs can use an added bit to ensure that the total number of 1-bits in a serial frame is odd or even. Therefore, there are two parity bit types used for error detection: odd and even. The odd parity bit is set when the number of 1-bits is odd, and the even parity bit is set when the number of 1-bits is even, respectively. Table 4.4 shows how serial frames look with either parity enabled.

Character	Binary	Number of ones	Serial frame 8-O-1	Serial frame 8-E-1
a	0110 0001	odd	0-10000110-1-1	0-10000110-0-1
c	0110 0011	even	0-11000110-0-1	0-11000110-1-1

Table 4.4: Odd an Even Parity Examples

There are also other parity types that do not have an error detection function. Mark and space belong to stick parity. With mark parity, the parity bit is always 1, and with stick parity, the parity bit is always 0. Some 9-bit networks use it to detect whether a UART frame contains an address or data [5].

Modem signals A full-featured UART like 8250 and its derivatives implements additional six signals used to control a modem: RTS, CTS, DTR, DSR, DCD, and RI. These signals handle hardware handshake, device and line status as also incoming calls. If the UART is not used with a modem, then mostly used signals are RTS/CTS.

Request To Send (RTS) and Clear To Send (CTS) signals are cross-coupled between two communicating devices. Each device uses its RTS to signal if it is ready to accept new data and check CTS to see if it is allowed to send data [17].

4.2 Supported UARTs in RIOT

RIOT operating system supports various microcontrollers (MCUs). These MCUs provide one or several UARTs. Table 4.5 on the next page shows a non-exhaustive list of UARTs supported in RIOT and their features. As can be seen from the table MCUs implement only a small subset of the UART capabilities compared to the 8250 family.

The MCUs provide only built-in RTS/CTS handshake, there is no modem control/status register equivalent. So if these signals should be controlled by software such a functionality will have to implemented using signals in GPIO mode.

Only a small subset of the MCUs supports stick parity. Data bits support also differs from MCU to MCU.

Hardware support for RS485 mode is available only by a small number of MCUs. In these MCUs RTS signal will be used to control the RS485 transmitter. Other controllers must use a free GPIO and shift register empty interrupt in order to determine when the last bit went on wire to disable the transmitter.

MCU	Modem Control Signals	Data Bits	Parity	RS485 Support
ATmega328 [35]	none	5 to 9	even, odd	no
CC2538 [14]	RTS, CTS	5 to 8	even, odd, stick	no
MSP430x1xx [15]	none	8 to 9	even, odd	no
PIC32 [34]	RTS, CTS	8 to 9	even, odd	no
SAM3X [4]	RTS, CTS	5 to 9	even, odd, stick	yes
STM32F04x [29]	RTS, CTS	7 to 9	even, odd	yes
STM32F446xx [30]	RTS, CTS	8 to 9	even, odd	no
ESP32 [31]	RTS, CTS	5 to 8	even, odd	no
nRF51 [27]	RTS, CTS	8	even	no

Table 4.5: UARTs Supported in RIOT

4.3 UART API in RIOT

RIOT's low-level UART peripheral driver was designed with the simplicity in mind to allow for easy implementation and maximum portability. So it uses the common 8-N-1 format of the serial port i.e. 8 data bits, no parity and one stop bit. The only parameter a user can change is the baud rate.

Initialization `uart_init` routine takes care of the UART initialization (see its definition below).

```
int uart_init (uart_t uart,
               uint32_t baudrate,
```

```
                uart_rx_cb_t rx_cb,
                void * arg
                )
```

It accepts following parameters:

- uart - UART device to initialize
- baudrate - desired baudrate in baud/s
- rx_cb - receive callback, executed in interrupt context once for every byte that is received (RX buffer filled), set to NULL for TX only mode
- arg - optional context passed to the callback functions

```
void uart_write (uart_t uart,
                 const uint8_t * data,
                 size_t len
                )
```

uart_write routine sends len characters from the given buffer data. The function is blocking and will only return if len bytes are sent.

```
typedef void(* uart_rx_cb_t) (void *arg, uint8_t data)
```

Character reception is implemented using an interrupt callback function, that was given to the uart_init routine.

4.4 UART Test Overview

UART API allows a user to configure UART and send/receive serial data. All UARTs from Table 4.5 on the preceding page support serial frame configuration. The majority of devices also provide RTS/CTS signals. Automatic RS485 transmitter control is only supported by a small number of MCUs and thus will not be handled further.

Together with PHiLIP as mocking hardware the following test types can be made with a HIL simulation:

- Configuration tests.
- Data transmission tests.
- Modem signal tests.

Configuration tests take care of UART initialization, baud rate calculation and also serial data frame configuration (data bits, parity and stop bits). Data frame configuration requires an API extension that is also part of this work.

Data transmission tests check that the previously applied configuration works correctly. A successful API call only means that a setting could be applied but its effect can be only checked in real data transmission.

Modem signal tests check whether RTS/CTS pin configuration was correctly applied

Configuration and data transmission tests were implemented in PHiLIP and also integrated into RIOT CI environment. Modem signal tests were only implemented in PHiLIP as runtime hardware handshake configuration API is still in evaluation.

5 Extending UART API

Currently, UART API does not allow to configure the serial interface to any other mode as 8-N-1 at runtime. This makes RIOT difficult to be used with the serial protocols/devices requiring other settings. The author of a pull request "UART: setting baudrate, stopbits, and parity on runtime" [1] suggests that UART API should be extended to support such serial settings as a number of data and stop bits as also parity.

5.1 STM32 UART

As a proof of concept STM32 platform was chosen. STM32 UARTs have following capabilities: 7, 8 and 9 data bits, odd and even parity as also 1 and 2 stop bits.

Not all devices share the same functionality. As for a number of supported data bits, there are two STM32 UART flavors: L1, F1, F2, F4 series support only 8 and 9 bits and the devices belonging to the F0, F3, F7, L0, L4 generally support 7, 8 and 9 bits. But some of the devices from the second group do not support 7 bits [29].

The configuration takes place in the control registers USART_CR1 and USART_CR2.

STM32 UART's number of data bits depends on the parity settings, i.e. if a frame should have 8 data bits and a parity then a 9 bit mode will have to be setup. Table 5.1 on the next page shows all possible combinations (the two last combinations are not available to the UARTs that do not support 7 data bits). M bits are USART_CR1_M, USART_CR1_M0 and USART_CR1_M1 enabling 9 and 7 data bits accordingly. PCE or USART_CR1_PCE bit enables parity control. USART_CR1_PS bit configures odd or even parity.

USART_CR2_STOP_1 bit enables 2 stop bits.

[1] https://github.com/RIOT-OS/RIOT/pull/5899

M bits	PCE bit	Frame
00	0	|start bit|8-bit data|stop bit|
00	1	|start bit|7-bit data|parity bit|stop bit|
01	0	|start bit|9-bit data|stop bit|
01	1	|start bit|8-bit data|parity bit|stop bit|
10	0	|start bit|7-bit data|stop bit|
10	1	|start bit|6-bit data|parity bit|stop bit|

Table 5.1: Frame Formats

USART_CR1_UE bit on the UARTs supporting 7-bit mode must be unset prior to changing serial settings.

The incoming data will be stored in the USART_RDR_RDR register together with the parity bit. Hence, when read into the software buffer, parity bit becomes part of the data byte. This is rather uncommon approach as usually the UARTs discard start, parity and stop bits and do not pass them to the host [10].

5.2 Configuration Routine

New routine uart_mode extends the API. Listing 5.1 shows function definition.

```
int uart_mode (uart_t uart,
               uart_data_bits_t data_bits,
               uart_parity_t parity,
               uart_stop_bits_t stop_bits
              )
```

Listing 5.1: Definition of the uart_mode Function

- uart - UART device to configure
- data_bits - number of data bits in a UART frame
- parity - parity mode
- stop_bits - number of stop bits in a UART frame

Three new enumeration types map parity, data and stop bits. The idea behind these enumerations is to specify a register configuration bits required to enable serial settings. These values can then be just assigned to the register without checking for the exact value in a switch/case or an if clause (see Listing A.1 on page 56 lines 40 and 41). Each platform overrides the common enumerations with related values and also marks not supported modes. The values defined as zero specify default values, i.e. uart_mode(uart, 0, 0, 0) configures UART to 8-N-1 mode.

Listing 5.2 demonstrates how it is implemented for the parity. Even parity is activated with parity control enable bit, and the odd parity requires both parity control enable and parity select bits. STM32 does not support mark and space parities, therefore, these modes are marked as invalid. The enumeration for stop bits is simpler as all defined modes are supported (refer to Listing 5.3).

```
typedef enum {
    UART_PARITY_NONE = 0,                                   /**< no parity */
    UART_PARITY_EVEN = USART_CR1_PCE,                       /**< even parity */
    UART_PARITY_ODD = (USART_CR1_PCE | USART_CR1_PS),       /**< odd parity */
    UART_PARITY_MARK = UART_INVALID_MODE | 4,               /**< not supported */
    UART_PARITY_SPACE = UART_INVALID_MODE  | 5              /**< not supported */
} uart_parity_t;
```

Listing 5.2: Enumeration for the parity

```
typedef enum {
   UART_STOP_BITS_1 = 0,                    /**< 1 stop bit */
   UART_STOP_BITS_2 = USART_CR2_STOP_1,     /**< 2 stop bits *
} uart_stop_bits_t;
```

Listing 5.3: Enumeration for the number of stop bits

As already mentioned the number of supported data bits cannot be detected based on the CPU family. The solution is to use USART_CR1_M1 macro that is only defined for CPUs implementing this feature. Listing 5.4 shows 7-bit mode being enabled only if USART_CR1_M1 is defined.

```
typedef enum {
    UART_DATA_BITS_5 = UART_INVALID_MODE | 1,    /**< not supported */
    UART_DATA_BITS_6 = UART_INVALID_MODE | 2,    /**< not supported unless
        parity is set */
#if defined(USART_CR1_M1)
```

```
    UART_DATA_BITS_7 = USART_CR1_M1,            /**< 7 data bits */
#else
    UART_DATA_BITS_7 = UART_INVALID_MODE | 3,   /**< not supported unless
        parity is set */
#endif
    UART_DATA_BITS_8 = 0,                       /**< 8 data bits */
} uart_data_bits_t;
```

Listing 5.4: Enumeration for the number of data bits

Another challenge is the MSB parity bit during the data reception. This issue arises only when the number of data bits is smaller than 8 and parity is enabled. For example receiving '0' or 0x30 character with 7-E-1 mode, will result in reading 00110000 from the USART_RDR_RDR register, so the character will be correctly passed to the user space. But receiving '1' or 0x31 character, will result in reading 10110001, hence delivering a wrong character i.e. 0xb1 instead of 0x31.

The MSB has to be masked as the receive callback is to return the data bits. The first and the last place where the UART driver can handle the incoming data is the interrupt handler. The mask will be defined in an interrupt context structure as shown in Listing 5.5. This way the handler does not need to spend extra time to determine what mask to use (see Listing A.2 on page 57 lines 11 and 23).

```
/**
 * @brief   Allocate memory to store the callback functions
 *
 * Extend standard uart_isr_ctx_t with data_mask field. This is needed
 * in order to mask parity bit.
 */
static struct {
    uart_rx_cb_t rx_cb;   /**< data received interrupt callback */
    void *arg;            /**< argument to both callback routines */
    uint8_t data_mask;    /**< mask applied to the data register */
} isr_ctx[UART_NUMOF];
```

Listing 5.5: STM32 Interrupt Context Structure

The function uart_mode is so far only provided for STM32 platform. Until all platforms implement it, this code is guarded with MODULE_PERIPH_UART_MODECFG macro and exposed to the applications as a periph_uart_modecfg feature. As soon as all

platforms add it to their UART drivers, both macro and a feature can be removed and new platforms will have to provide uart_mode's implementation at once.

5.3 Verification

During the implementation of the uart_mode routine, several testing techniques were used.

The first verification occurred using a terminal emulator and RIOT's testing firmware, i.e. tests/periph_uart/main.c. This manual method was sufficient to check whether STM32 could correctly send bytes with changed parity and data bits. Though this only checked that no framing errors occurred during the transmission. Terminal software ignores parity errors so that the characters will be displayed correctly even if one side was sending with even parity and the other side was receiving with odd parity.

The opposite direction was even more challenging because of the parity bit inclusion into the data read register. So even if the parity was correctly set on both sides, the characters with parity bit set were not the expected ones. It was also hard to check whether two stop bits were actually sent. Both STM32 and FTDI only check for the first stop bit to assure the correct framing. The second stop bit will be ignored. So using an oscilloscope as shown in Section 7.3 on page 45 was required to verify that all serial frames were correctly sent.

As soon as all settings were applied correctly and other implementation aspects were changing, PHiLIP was used to perform verification automatically.

6 Integrating UART Tests into RIOT

Adding UART testing to the RIOT CI process required changes to several projects. First of all PHiLIP's firmware has to be extended to support UART, then RIOT peripheral testing infrastructure has to be reworked to perform tests with PHiLIP.

6.1 Preparing PHiLIP for UART Testing

PHiLIP is using STM32 UART and hence has following capabilities:

- 8 to 9 data bits.
- None, even and odd parities.
- Hardware flow control on RTC/CTS pins.
- 1 to 2 stop bits.
- Overrun, noise, frame and parity error detection.

Though compared to Table 4.5 on page 19 PHiLIP covers not all UART features, but most common settings can be tested. The qualified firmware was extended to allow the following: receive/transmit tests at different baud rates, none, even and odd parity tests as also automatic hardware flow control tests.

Figure 6.1 on the next page shows a development setup for a PHiLIP device. Host PC connects to both UARTs using FTDI based USB to serial adapters.

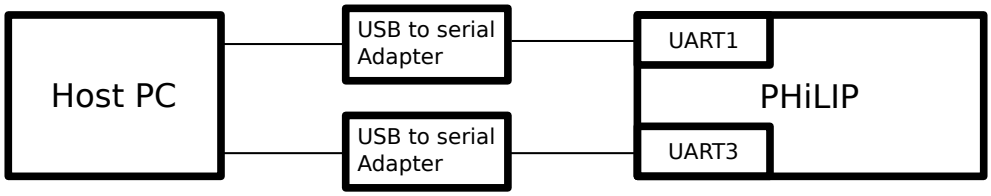


Figure 6.1: PHiLIP Development Setup

6.1.1 Software Structure

The software structure corresponds to PHiLIP's git repository [37] at the time of commit 864fd153e69eb7b8de91920ca85bbf31a75dcd92.

UART test logic is tightly interwoven with the serial communication interface used to control the PHiLIP application. This code resides in `port_uart.c` source file. The UART test specific code resides in `port_dut_uart.c` and handles UART settings like baud rate, parity, data and stop bits as also setting/clearing of the RTS signal.

PHiLIP's serial communication is working in an input mode that is similar to libc's canonical mode, i.e. the string will be returned only if the \n character is received. So PHiLIP is waiting for \n and as soon as it is received it starts with command parsing.

The register map was extended with `uart_t` structure that holds UART related settings. UART behavior will be defined by the mode field. There are four modes:

- MODE_ECHO - received data will be returned as is, i.e. simulating virtual loopback.
- MODE_ECHO_EXT - this is a data changing mode. All received data bytes will be incremented by one.
- MODE_REG - register access mode. In this mode UART is working as the serial controlling link.
- MODE_TX - special mode that permanently sends data.

baud holds UART's baud rate define as an integer. `rx_count` and `tx_count` hold number of bytes sent to and received from the DUT. `ctrl` structure will be used to configure parity, stop bits as also toggle the RTS pin. `status` structure indicates CTS state.

```
typedef union uart_t_TAG {
        struct {
                /* Test mode */
                uint8_t mode;
                /* Baudrate */
                uint32_t baud;
                /* Number of received bytes */
                uint16_t rx_count;
                /* Number of transmitted bytes */
                uint16_t tx_count;
                /* UART control register */
                uart_ctrl_t ctrl;
                /* UART status register */
                uart_status_t status;
                /* Reserved bytes */
                uint8_t res[5];
        };
        uint8_t data8[16];
} uart_t;

typedef struct uart_ctrl_t_TAG {
        /* Number of stop bits */
        uint8_t stop_bits : 1;
        /* Parity */
        uint8_t parity : 2;
        /* RTS pin state */
        uint8_t rts : 1;
        /* Number of data bits */
        uint8_t data_bits : 1;
} uart_ctrl_t;

typedef struct uart_status_t_TAG {
        /* CTS pin state */
        uint8_t cts : 1;
        /* Parity error */
        uint8_t pe : 1;
        /* Framing error */
        uint8_t fe : 1;
        /* Noise detected flag */
        uint8_t nf : 1;
        /* Overrun error */
        uint8_t ore : 1;
} uart_status_t;
```

6.1.2 Receive and Transmit

As pointed in Section 4.1 on page 14 UART's clock source can have a great impact on sending and receiving serial data. Not only the external clock but also the clock for the UART subsystem in an MCU can affect sampling accuracy and the resulting transmit baud rate. So it is important to make sure an embedded system's UART supports target baud rate. PHiLIP provides various modes to test the DUT's transmit/receive behavior.

STM32 UART is capable of detecting following receive errors:

- Overrun error is detected when data cannot be transferred from the shift register to the RDR register.
- Noise error is detected when at least one of the samplings shows 2 of 3 sampled bits are at 0.
- Framing error is detected when the stop bit is not recognized on reception at the expected time.
- Parity error is detected when the parity bit does not match the number of expected 1-bits.

Echo or virtual loopback mode receives a string ending with \n character and sends it back to DUT. Such a test can be performed at any baud rate that both devices support.

As soon as DUT does not receive the same string as was sent `uart_status_t` register can be checked for the possible cause of transmission issue. If either noise or framing errors are set then PHiLIP could not reliable receive the data. If no such errors were set then DUT could not receive the data. Both `rx_count` and `tx_count` should be checked to determine how many characters where received and sent by PHiLIP.

6.1.3 Parity

PHiLIP supports none, odd and even parities modes. Taking 8 and 9-bit data into account following test combinations with parity bit are possible:

- 7 bit data and even parity.

- 7 bit data and odd parity.
- 8 bit data and even parity.
- 8 bit data and odd parity.

During the UART API extension, a special feature of STM32 UART was revealed. I.e. that the parity bit together with the serial data is placed into the data register. As long as the data reading is interrupt driven the byte can be masked in the interrupt routine. But as PHiLIP configures UART in DMA mode, the data has to be masked when reading from DMA buffer. Listing A.3 on page 58 shows `_rx_str` routine that handles incoming data.

6.1.4 Hardware Handshake

MCU's in RIOT support automatic hardware handshake on RTS/CTS pins if these dedicated pins are provided at all. The test should check whether the automatic hardware handshake was configured correctly i.e. no data is to be sent when CTS is read as high and RTS is to be raised when the UART's hardware handshake trigger level is reached.

If RTS/CTS mode is activated RTS cannot be toggled in software. This is also true for STM32 used in PHiLIP. To overcome this limitation pins RTS/CTS will be configured as GPIOs. This way RTS signal can be toggled and CTS signal change can generate an interrupt.

Hardware handshake test will be executed in two steps. The first step checks whether the DUT's transmitter would react to its CTS signal change and stop sending. Via serial control interface PHiLIP will be instructed to set RTS signal, then DUT will try to send a test data. If after a defined timeout no data will be received on DUT side, the test is positive. Otherwise, the handshake is not working or configured properly.

The second step is more complicated. DUT's UART is in automatic hardware handshake mode and cannot toggle its RTS signal, so PHiLIP has to bring the other UART to set RTS. This can be achieved by sending enough data to fill the Rx FIFO so that UART will have to set its RTS signal.

MODE_TX was developed to continuously send data and thus force the DUT to raise its RTS signal. Enabling it will make PHiLIP sending character 'a' continuously till this mode is changed. The interrupt handler for the CTS pin catches the signal change on

rising edge (see Listing 6.1) and saves it into cts field of the uart_status_t structure (see Listing 6.2).

```
/* Configure GPIO pin : DUT_CTS_Pin */
GPIO_InitStruct.Pin = DUT_CTS_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_PULLDOWN;
HAL_GPIO_Init(DUT_CTS_GPIO_Port, &GPIO_InitStruct);
```

Listing 6.1: CTS Interrupt Configuration

```
void EXTI15_10_IRQHandler(void)
{
        /* USER CODE BEGIN EXTI15_10_IRQn 0 */
        uint8_t status;
        read_regs(offsetof(map_t, uart.status), (uint8_t *)&status, sizeof(((
            uart_t *)0)->status));
        status |= 0x01;
        write_regs(offsetof(map_t, uart.status), (uint8_t *)&status,  sizeof
            (((uart_t *)0)->status), IF_ACCESS);
        #ifdef BLUEPILL
        HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
        #endif
        #ifdef NUCLEOF103RB
        HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_10);
        #endif
        /* USER CODE END EXTI15_10_IRQn 1 */
}
```

Listing 6.2: CTS Interrupt Handler

Figure 6.2 on the following page shows two signals: PHiLIP's TX transmitting multiple 'a' characters and DUT's RTS going high.

6.1.5 Validating PHiLIP Firmware

At the moment of writing, only I2C and UART slaves were implemented. Hence, the PHiLIP project is still in active development and requires constant testing. The qualified firmware can be released only if the whole functionality is verified.

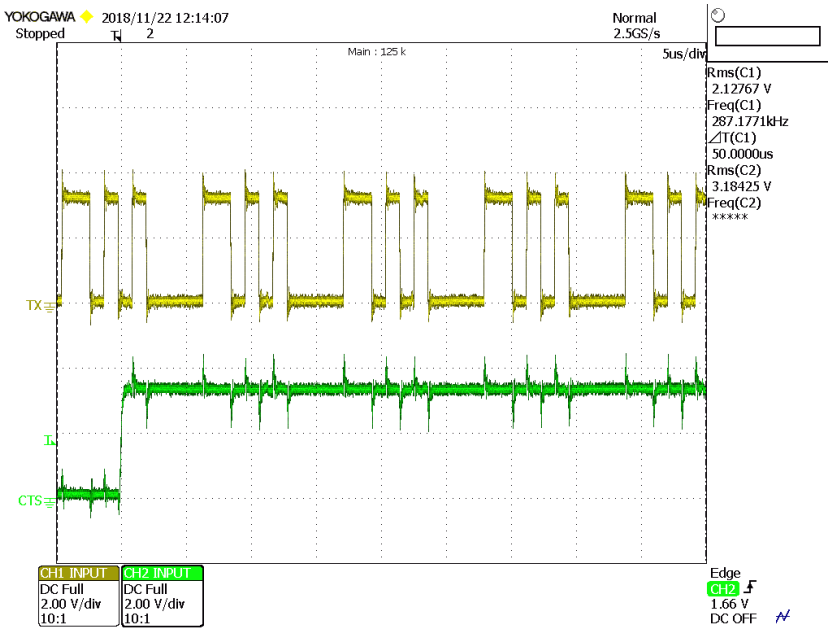


Figure 6.2: CTS Assertion

The test will be handled with a Python script TEST/uart_module_test.py. It accepts three parameters:

- port - serial device connected to PHiLIP's control port.
- dutport - serial device connected to PHiLIP's DUT port.
- loglevel - script's verbosity.

6.2 Preparing RIOT for UART Testing

The testing process should ensure that both newly introduced boards/platforms are correctly implemented and that refactoring, bug fixing, etc. do not break the already working configuration.

Echo tests with different baud rates cover the UART initialization, baud rate calculation and data reception/transmission aspects. Using the current UART API calls these tests check that the UARTs initialization does not freeze (proper IRQ locking sequence), the correct UART clock is selected, the baud rate is set without returning an error and baud rate calculation error is within a tolerance range. It is necessary to perform the baud rate

tests on the most used mode settings (such as 8-N-1) as this is the only mode supported by all platforms. The short echo tests with predefined characters check for baud rate accuracy and the tests with random data stress the DUT and check that different patterns can be correctly sampled.

The advanced UART configuration tests check not only whether the UART can set the modes correctly, but that uart_mode calls can be executed one after another without the need to completely reinitialize the UART using uart_init call. The negative tests ensure that the required settings were effectively applied.

6.2.1 PHiLIP Keywords

First of all tests require an abstraction of the RIOT PAL protocol to get meaningful keywords for UART configuration. dist/robotframework/lib/PhilipAPI.py implements PhilipAPI class. It provides the following methods reset_dut, setup_uart, get_counters and get_error_flags.

reset_dut resets DUT using a dedicated GPIO signal. This step is necessary in order to bring the DUT into an initial state and thus establish the same condition for all tests.

setup_uart provides one method where all UART related settings can be set. This method changes PHiLIP's memory map and applies these changes.

get_counters reads receive and transmit counter fields from PHiLIP's memory map.

get_error_flags reads error flags from PHiLIP's memory map.

PhilipAPI will be imported via philip.keywords.txt file as PHILIP object.

```
*** Settings ***
Library             PhilipAPI  port=%{PHILIP_PORT}  baudrate=${115200}  WITH
    NAME  PHILIP

Resource            riot_base.keywords.txt

*** Keywords ***
Reset DUT and PHILIP
[Documentation]     Reset the device under test and the PHILIP tester.
Reset Application
PHILIP.Reset MCU
```

```
PHILIP.Reset DuT

Show PHILIP Statistics
[Documentation]     Show rx/tx counters and error flags.
${rx}   ${tx} =  PHILIP.Get counters
${pe}   ${fe}   ${nf}   ${ore} =  PHILIP.Get error flags
Set Test Message   RX: ${rx['data']}, TX: ${tx['data']}, PE: ${pe['data']},
    FE: ${fe['data']}, NF: ${nf['data']}, ORE: ${ore['data']}

Get PHILIP Statistics
[Documentation]     Return rx/tx counters and error flags.
${rx}   ${tx}=                    PHILIP.Get counters
${pe}   ${fe}   ${nf}   ${ore}=   PHILIP.Get error flags
[return]                          RX: ${rx['data']}, TX: ${tx['data']}, PE: $
    {pe['data']}, FE: ${fe['data']}, NF: ${nf['data']}, ORE: ${ore['data']}
```

Listing 6.3: PHiLIP Keywords

6.2.2 Transmit and Receive

The tests begin with an echo mode where a string "t111" will be sent and received at 115200b/s and 8-N-1 configuration. In the case of an error, this predefined pattern can be checked on an oscilloscope.

After checking the basic UART functionality a test with a long string consisting of randomly chosen digits will be made.

The extended echo tests i.e. data changing tests use the same patterns.

Register access tests simulate real slave communication using a defined protocol.

Baud rate tests take a number of standard speeds and perform transmissions in echo mode. As a negative test both PHiLIP and DUT perform transmission at different baud rates.

6.2.3 Data Bits, Parity and Stop Bits

The newly introduced support for changing advanced serial settings at runtime is so far only available for STM32 UARTs. Therefore, the tests should be able to detect

support for uart_mode command at runtime. All CPUs must support 8-N-1 configuration. Hence, if this command fails, the CPU in question does not provide support for uart_mode and this test must be skipped.

The parity tests will be performed first for 8 data bits and then for 7 data bits. Each parity test will be made along with a negative test where the opposite parity will be set. Listing 6.4 shows these tests.

```
Even Parity 8 Bits
DUT UART mode should exist  dev=1
PHILIP.Setup Uart           parity=${UART_PARITY_EVEN}
API Call Should Succeed     Uart Init
API Call Should Succeed     Uart Mode  data_bits=8   parity="E"   stop_bits=1
DUT Should Match String     1  ${SHORT_TEST_STRING}  ${SHORT_TEST_STRING}
API Call Should Succeed     Uart Mode  data_bits=8   parity="O"   stop_bits=1
DUT Should Not Match String or Timeout     1   ${SHORT_TEST_STRING}   ${
    SHORT_TEST_STRING}
Show PHILIP Statistics
```

Listing 6.4: Parity and Data Bits Test

The stop bits test is challenging because modern UARTs do not check for the second stop bit i.e. as soon as the first stop bit is detected, the frame is valid. Hence, the negative test when DUT sends with two stop bits and PHiLIP receives with one stop bit will fail. As a consequence, the test cannot really check whether DUT can work with two stop bits. As Figure 6.3a on the following page shows, two 't' characters have no delay between them. Therefore, a receiver relying on two stop bits will fail to correctly receive the data.

The solution is to configure PHiLIP in such a mode that it will be forced to verify all the bits from DUT. Following example demonstrates how the test works with PHiLIP configured in 8-O-1 and DUT in 8-N-2.

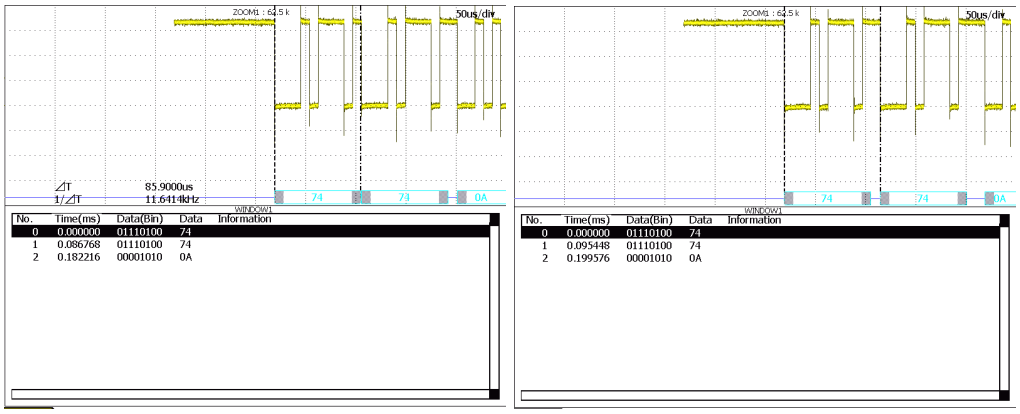
DUT sends 't' character or 0x74. Translated into a serial frame it looks like 0-00101110-11. PHiLIP interprets this character as 0-00101110-1-1 i.e the same number of bits. But in the first case both '1' at the end of the frame are stop bits and in the second case one '1' is a parity bit and the other is a stop bit. Therefore, PHiLIP will definitely check the full frame.

The 8-O-1 mode is chosen because every test string has to be terminated with 'NL' character or 0x0a. This character has an odd number of '1' bits. Hence, all characters in a test string must have an odd number of bits. Figure 6.3b demonstrates such a test string.

Listing 6.5 shows both the positive and negative tests. At first a string will be sent and received with 8-N-2 configuration. Then serial parameters will be changed to 8-N-1 and the test times out because PHiLIP gets error.

```
Two Stop Bits
DUT UART mode should exist  dev=1
PHILIP.Setup Uart           parity=${UART_PARITY_ODD}
API Call Should Succeed     Uart Init
API Call Should Succeed     Uart Mode   data_bits=8   parity="N"   stop_bits
    =2
DUT Should Match String     1  ${TEST_STRING_FOR_STOP_BITS}  ${
    TEST_STRING_FOR_STOP_BITS}
API Call Should Succeed     Uart Mode   data_bits=8   parity="N"   stop_bits
    =1
DUT Should Not Match String or Timeout     1   ${TEST_STRING_FOR_STOP_BITS}
      ${TEST_STRING_FOR_STOP_BITS}
Show PHILIP Statistics
```

Listing 6.5: Two Bits Test



(a) Test String at 8-N-1 (b) Test String at 8-N-2 or 8-O-1

Figure 6.3: Stop Bits Test Strings

6.2.4 Test Reports

Robot Framework provides extensive logging and reporting in XML format that will then be automatically converted to HTML format. Figure 6.4 on the next page shows the main test report including all test suits. Total statistics like the number of tests, duration, and pass to fail ratio. Test failures were artificially provoked to demonstrate how error reporting was implemented for UART tests.

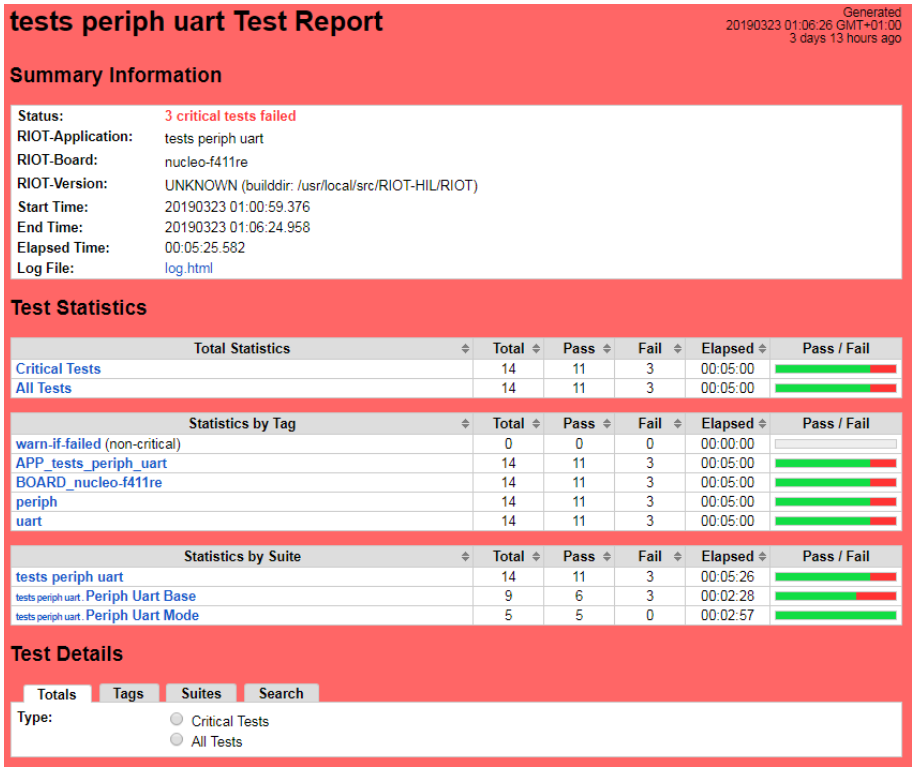
Figure 6.5 on the following page and Figure 6.6 on page 40 provide details for individual test suits. This time every test including its statistics is visible. "Message" field is used to show test related information both in the case of successful execution and failure. For UART tests this field shows PHiLIP's statics i.e. rx/tx counters and error flags. Such data helps to understand the cause of a failure as described in Section 6.1.2 on page 30. Three tests from the Base test suite are marked as failed: Echo, Extended Echo, and Wrong Baud Rate Test.

Echo test sends a short string and expects the same string in return. "Test timed out: RX: 0, TX: 0, PE: 0, FE: 1, NF: 0, ORE: 0" is displayed in the "Message" field. To understand why the test timed out PHiLIP's statistics should be examined. RX: 0 means PHiLIP could not detect a string terminated with 'NL. FE: 1 points to a framing error that explains why the test timed out.

Extended Echo test sends a short string and expects in return all characters except 'NL' to be incremented by one. "Reference string does not match the received one: RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0" is displayed in the "Message" field. Both counters show that PHiLIP could receive and send the test string. Possible error cause could be that at least one character got damaged on a wire but the framing was correct.

Wrong Baud Rate test sends a short test string but at a different baud rate than PHiLIP is set to and thus expects a timeout. The test fails because the baud rate on the DUTs side was not changed.

Figure 6.7 on page 40 shows a successful and Figure 6.8 on page 41 a failed test with all their steps. Every step can be expanded to the most low-level keyword and the failed keyword appears in red.



tests periph uart Test Report

Generated
20190323 01:06:26 GMT+01:00
3 days 13 hours ago

Summary Information

Status:	3 critical tests failed
RIOT-Application:	tests periph uart
RIOT-Board:	nucleo-f411re
RIOT-Version:	UNKNOWN (builddir: /usr/local/src/RIOT-HIL/RIOT)
Start Time:	20190323 01:00:59.376
End Time:	20190323 01:06:24.958
Elapsed Time:	00:05:25.582
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	14	11	3	00:05:00	
All Tests	14	11	3	00:05:00	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
warn-if-failed (non-critical)	0	0	0	00:00:00	
APP_tests_periph_uart	14	11	3	00:05:00	
BOARD_nucleo-f411re	14	11	3	00:05:00	
periph	14	11	3	00:05:00	
uart	14	11	3	00:05:00	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
tests periph uart	14	11	3	00:05:26	
tests periph uart . Periph Uart Base	9	6	3	00:02:28	
tests periph uart . Periph Uart Mode	5	5	0	00:02:57	

Test Details

Totals | Tags | Suites | Search

Type: ○ Critical Tests ○ All Tests

Figure 6.4: Robot Framework Report Overview

Test Details LOG

Totals | Tags | Suites | Search

Name:	tests periph uart.Periph Uart Base
Status:	9 critical test, 6 passed, 3 failed 9 test total, 6 passed, 3 failed
Start / End Time:	20190323 01:00:59.592 / 20190323 01:03:27.410
Elapsed Time:	00:02:27.818
Log File:	log.html#s1-s1

Name	Documentation	Tags	Crit.	Status	Message	Elapsed	Start / End
tests periph uart . Periph Uart Base . Echo		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	FAIL	Test timed out: RX: 0, TX: 0, PE: 0, FE: 1, NF: 0, ORE: 0	00:00:33.092	20190323 01:01:13.272 20190323 01:01:46.364
tests periph uart . Periph Uart Base . Long Echo		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 44, TX: 44, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:13.016	20190323 01:01:46.374 20190323 01:01:59.390
tests periph uart . Periph Uart Base . Extended Echo		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	FAIL	Reference string doesn't match the received one: RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.043	20190323 01:01:59.394 20190323 01:02:11.437
tests periph uart . Periph Uart Base . Extended Long Echo		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 44, TX: 44, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.215	20190323 01:02:11.441 20190323 01:02:23.656
tests periph uart . Periph Uart Base . Register Access		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 17, TX: 9, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.215	20190323 01:02:23.659 20190323 01:02:35.874
tests periph uart . Periph Uart Base . Should Not Access Invalid Register		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 9, TX: 3, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.775	20190323 01:02:35.883 20190323 01:02:48.658
tests periph uart . Periph Uart Base . Wrong Baud Test		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	FAIL	No timeout occurred and the reference string does match the received one: RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.894	20190323 01:02:48.669 20190323 01:03:01.563
tests periph uart . Periph Uart Base . Baud Test 38400		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:12.428	20190323 01:03:01.566 20190323 01:03:13.994
tests periph uart . Periph Uart Base . Baud Test 9600		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0	00:00:13.385	20190323 01:03:13.998 20190323 01:03:27.383

Figure 6.5: Robot Framework Report UART Base Tests

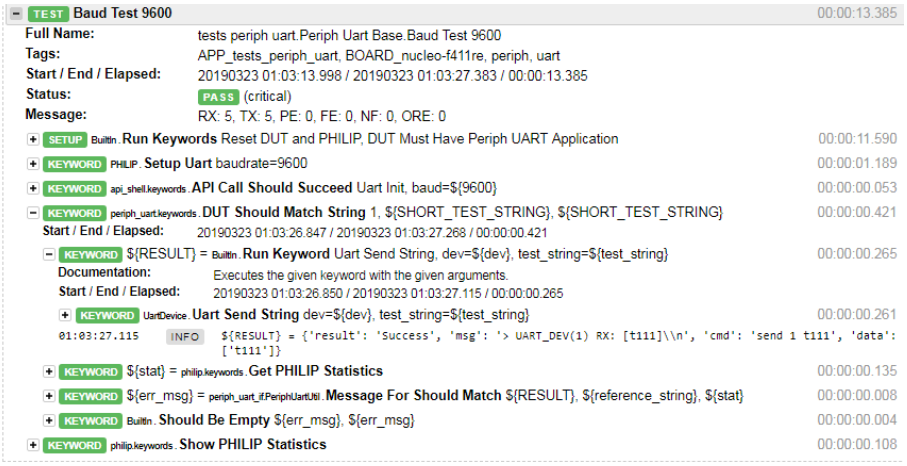
Test Details LOG

Totals | Tags | Suites | Search

Name: tests periph uart.Periph Uart Mode
Status: 5 critical test, 5 passed, 0 failed
5 test total, 5 passed, 0 failed
Start / End Time: 20190323 01:03:27.500 / 20190323 01:06:24.857
Elapsed Time: 00:02:57.357
Log File: log.html#s1-s2

Name	Documentation	Tags	Crit.	Status	Message	Elapsed	Start / End
tests periph uart . Periph Uart Mode . Even Parity 8 Bits		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 1, FE: 0, NF: 0, ORE: 0	00:00:32.930	20190323 01:03:38.563 20190323 01:04:11.493
tests periph uart . Periph Uart Mode . Odd Parity 8 Bits		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 1, FE: 0, NF: 0, ORE: 0	00:00:32.878	20190323 01:04:11.497 20190323 01:04:44.375
tests periph uart . Periph Uart Mode . Even Parity 7 Bits		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 1, FE: 0, NF: 0, ORE: 0	00:00:33.779	20190323 01:04:44.393 20190323 01:05:18.172
tests periph uart . Periph Uart Mode . Odd Parity 7 Bits		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 1, FE: 0, NF: 0, ORE: 0	00:00:32.909	20190323 01:05:18.189 20190323 01:05:51.098
tests periph uart . Periph Uart Mode . Two Stop Bits		APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart	yes	PASS	RX: 5, TX: 5, PE: 0, FE: 1, NF: 0, ORE: 0	00:00:33.724	20190323 01:05:51.100 20190323 01:06:24.824

Figure 6.6: Robot Framework Report UART Mode Tests



TEST **Baud Test 9600** 00:00:13.385

Full Name: tests periph uart.Periph Uart Base.Baud Test 9600
Tags: APP_tests_periph_uart, BOARD_nucleo-f411re, periph, uart
Start / End / Elapsed: 20190323 01:03:13.998 / 20190323 01:03:27.383 / 00:00:13.385
Status: PASS (critical)
Message: RX: 5, TX: 5, PE: 0, FE: 0, NF: 0, ORE: 0

SETUP BuiltIn . **Run Keywords** Reset DUT and PHILIP, DUT Must Have Periph UART Application 00:00:11.590
KEYWORD PHILIP . **Setup Uart** baudrate=9600 00:00:01.189
KEYWORD api_shell.keywords . **API Call Should Succeed** Uart Init, baud=${9600} 00:00:00.053
KEYWORD periph_uart.keywords . **DUT Should Match String** 1, ${SHORT_TEST_STRING}, ${SHORT_TEST_STRING} 00:00:00.421
Start / End / Elapsed: 20190323 01:03:26.847 / 20190323 01:03:27.268 / 00:00:00.421
KEYWORD ${RESULT} = BuiltIn . **Run Keyword** Uart Send String, dev=${dev}, test_string=${test_string} 00:00:00.265
Documentation: Executes the given keyword with the given arguments.
Start / End / Elapsed: 20190323 01:03:26.850 / 20190323 01:03:27.115 / 00:00:00.265
KEYWORD UartDevice . **Uart Send String** dev=${dev}, test_string=${test_string} 00:00:00.261
01:03:27.115 INFO ${RESULT} = {'result': 'Success', 'msg': '> UART_DEV(1) RX: [t111]\\n', 'cmd': 'send 1 t111', 'data': ['t111']}
KEYWORD ${stat} = philip.keywords . **Get PHILIP Statistics** 00:00:00.135
KEYWORD ${err_msg} = periph_uart_if.PeriphUartUtil . **Message For Should Match** ${RESULT}, ${reference_string}, ${stat} 00:00:00.008
KEYWORD BuiltIn . **Should Be Empty** ${err_msg}, ${err_msg} 00:00:00.004
KEYWORD philip.keywords . **Show PHILIP Statistics** 00:00:00.108

Figure 6.7: Robot Framework Successful Baud Test at 9600 b/s

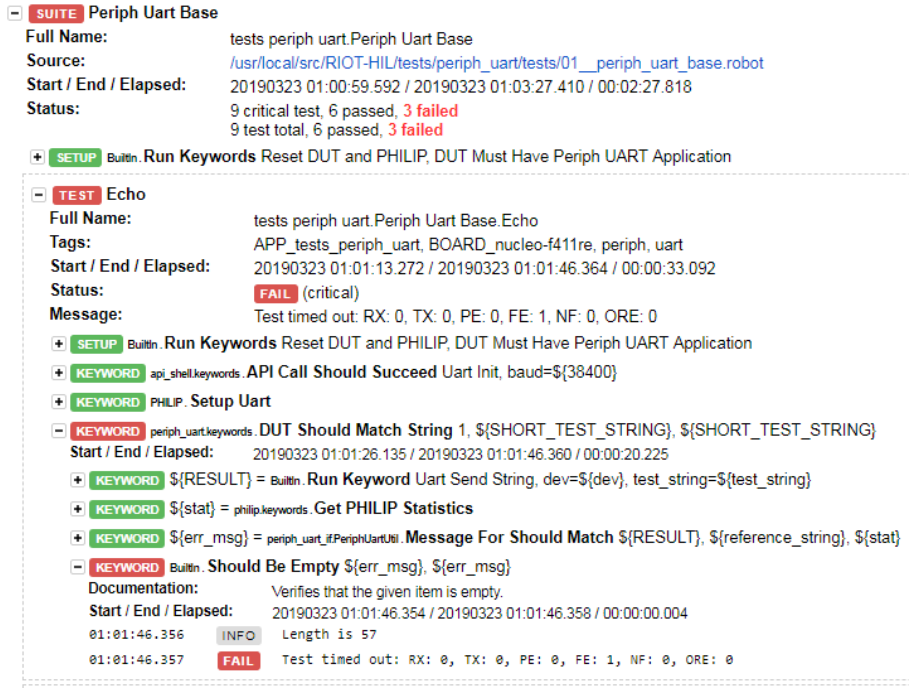


Figure 6.8: Robot Framework Failed Echo Test

6.3 Summary

Following goals were achieved during the implementation stage:

- PHiLIP project was extended to provide a UART slave functionality
- UART tests were integrated into the RIOT CI environment

PHiLIP's new functionality covers the features required for testing a UART as suggested in Section 4.4 on page 20. And it also provides various test statistics like rx/tx counters and error flags that help to analyze the test failures.

Robot Framework had a flat learning curve and due to its extensibility with Python, the tests could be easily integrated with RIOT PAL library. The keywords make the tests easy to understand even for persons not involved in the test development process.

Test reports provide a good overview of the executed test and their results. In the case of a test failure, UART tests show the failed keyword and statistics provided by PHiLIP what helps to quickly find the cause of the problem.

Testing two stop bits was challenging because most UARTs just ignore them. A special frame configuration had to be developed to force UARTs to check this setting. Parity handling required modifying the received data as STM32 UARTs leave it in the data byte. Aside from these issues, all other tests could be implemented based on UART specification.

Modem signal test was not implemented in RIOT because related API for setting hardware handshake is still missing from the UART driver. But PHiLIP already provides a functionality to test it.

7 Test Evaluation

7.1 Detected Issues in RIOT

After launching the CI service on PHiLIP basis one error with arduino-mega2560 was detected. According to issue 10517, a baud rate got truncated if set higher than 32767 b/s. For example, when set to 115200 b/s, the DUT sent data at 250 b/s.

Further investigation revealed that the problem was how `tests/periph_uart/main.c` parsed a baud rate parameter. During the parsing this parameter was parsed using `atoi()` function. For systems like STM32 where integer is 32-bit long, it was correct but for systems with 16-bit integer type, the value got truncated.

The solution was to use `strtol()` function returning a long integer.

7.2 Test with a Modbus Slave Simulator

Diagslave Modbus Slave Simulator [24] simulates a Modbus slave providing both network and serial versions of a protocol. The Modbus ASCII protocol used in this test supports 7 and 8 data bits, none, even, and odd parities as also 1 and 2 stop bits. These are exactly the settings that DUT (Nucleo-F411re board) supports.

A special program `modbus-riot-test` [38] was developed to send valid Modbus ASCII packets and check the correct responses.

This test checks following combinations at 115200b/s: 8-N-1, 8-E-1, 8-O-1, 7-E-1, 7-O-1. Two stop bits tests are omitted as both STM32 UART and FTDI FT232R used on the PC side ignore the second stop bit. For each configuration `modbus-riot-test` was recompiled with corresponding serial settings and flashed onto Nucleo board.

Modbus Slave Simulator was invoked with the following command line parameters for each test:

- 8-N-1: –m ascii –b 115200 –d 8 –s 1 –p none
- 8-E-1: –m ascii –b 115200 –d 8 –s 1 –p even
- 8-O-1: –m ascii –b 115200 –d 8 –s 1 –p odd
- 7-E-1: –m ascii –b 115200 –d 7 –s 1 –p even
- 7-O-1: –m ascii –b 115200 –d 7 –s 1 –p odd

Table 7.1 shows a frame structure and content that modbus−riot−test sends every second. The response of a Modbus slave is demonstrated in Table 7.2.

Name	Value(hex)	Description
Start	3A	Start of a frame: ":" character
Address	01	Slave address
Function	03	Read multiple holding registers
Register	0063	Read starting from the 100th register (0 is the first register)
Number	0005	Read 5 registers
LRC	94	Checksum
End	0D0A	End of the frame: carriage return and line feed

Table 7.1: Modbus Frame Request

Name	Value(hex)	Description
Start	3A	Start of a frame: ":" character
Address	01	Slave address
Function	03	Read multiple holding registers
Length	0A	Response contains 10 bytes of data (five 16-bit register values)
Data	00..00	All five registers have 0000 as data
LRC	F2	Checksum
End	0D0A	End of the frame: carriage return and line feed

Table 7.2: Modbus Frame Response

As soon as the Modbus slave detects a valid request it prints its summary on the screen:

```
Slave 1: readHoldingRegisters from 100, 5 references
```

For each serial frame configuration, the test makes 100 cycles. modbus−riot−test compares each received frame with a reference response and increases the packet counter. If a packet does not correspond to the reference one, an error counter is increased. At the end of each session the packet counter must be 100 and error counter 0:

```
RX: [:01030A00000000000000000000F20x0d]\n
Total message counter: 100
Error counter: 0
```

The tests for all serial configurations were successful.

7.3 Digital Oscilloscope Verification

Aside from tests against a PHiLIP device a digital oscilloscope with automatic UART signal parsing was used to verify that DUT (Nucleo-F411re board) correctly implements baud rate setup and the extended UART modes. With UART parsing mode enabled, oscilloscope acts like a UART and requires the same settings to correctly sample the serial data. The model used for evaluation can parse the following modes: 8-N-1, 8-O-1, 8-E-1, 7-O-1, 7-E-1. The tests were executed at 115200 b/s.

Figure 7.1 on the following page shows characters 't', 'u' and NL sent with 8-N-1 mode. The oscilloscope marks both the start and stop bits as gray rectangles. Between these rectangles, a hexadecimal value of a character is displayed. Two cursors around the first serial frame show the difference between the 8-N-1 and 8-N-2 modes. Though the oscilloscope does not support 8-N-2 mode directly it is possible to dissect such serial frames as Figure 7.2 on the next page demonstrates. The second stop bit appears just like a possible inter-character delay. Hence, the driver sets baud rate and the number of stop bits correctly.

Figure 7.3 on page 47 and Figure 7.4 on page 47 show the same characters but with enabled parity. Parity bit value also appears in the frame dissection. Figure 7.5 on page 48 and Figure 7.6 on page 48 show modes 7-O-1 and 7-E-1. The frames get shorter. Figure 7.7 on page 49 provides a negative example: the DUT sends data with 7-O-1 and the oscilloscope awaits 7-E-1. The character can still be decoded but parity error is reported.

The oscilloscope test confirmed the correctness of PHiLIP tests in regards to Nucleo-F411re board.

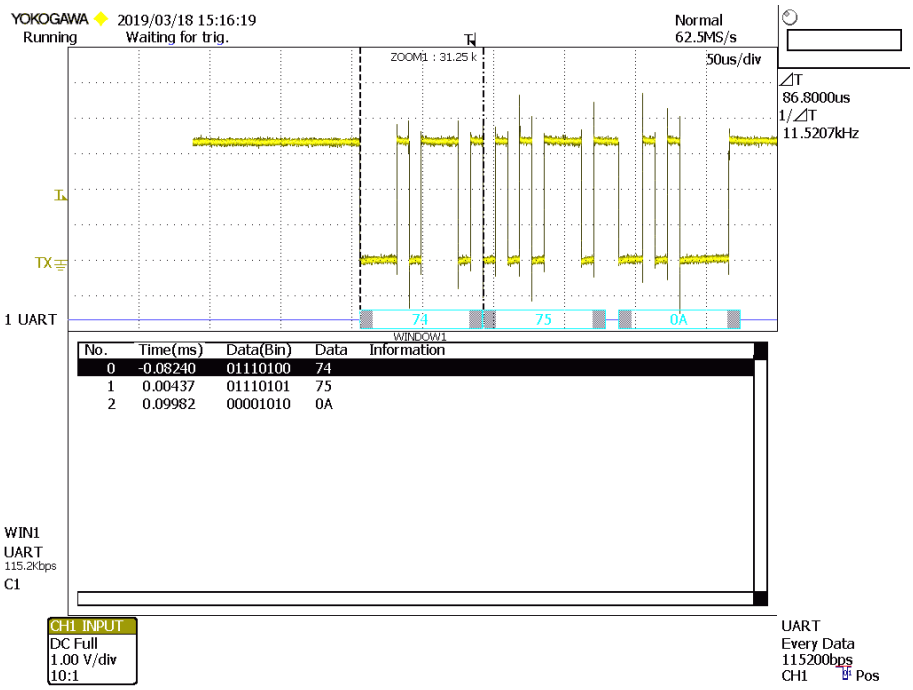


Figure 7.1: 8-N-1

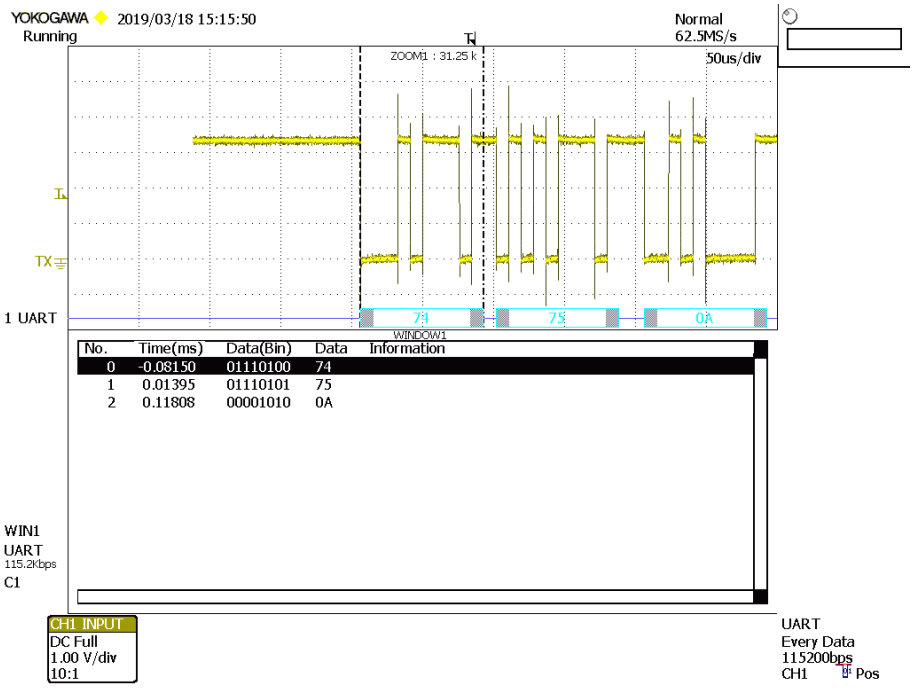


Figure 7.2: 8-N-2

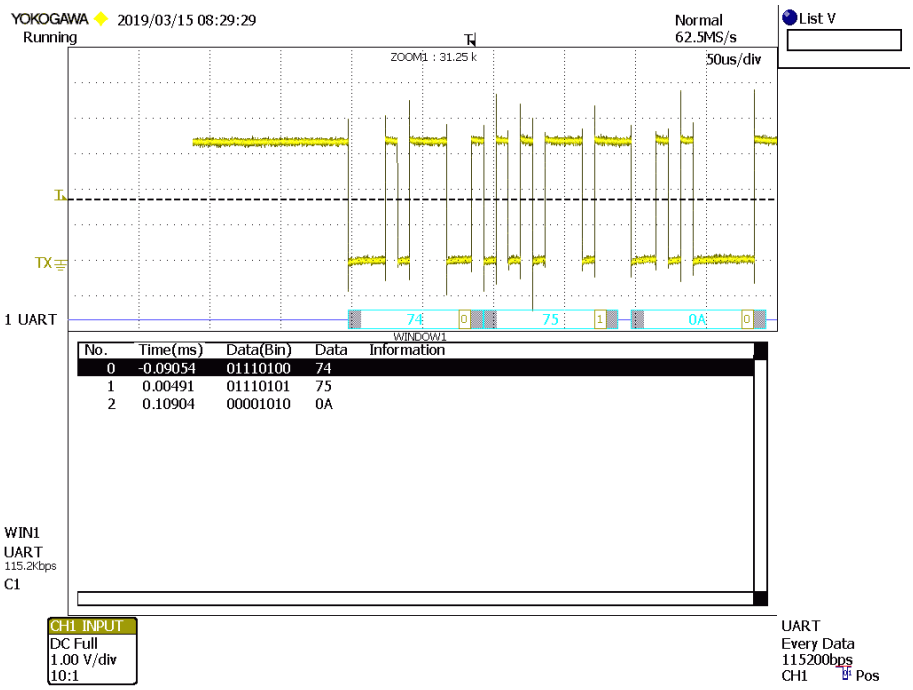


Figure 7.3: 8-O-1

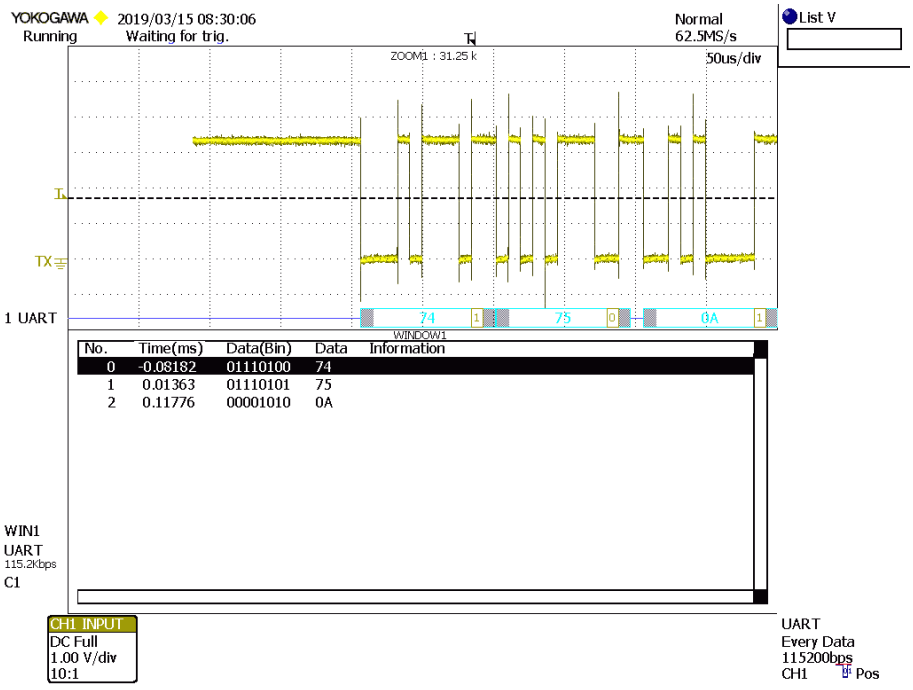


Figure 7.4: 8-E-1

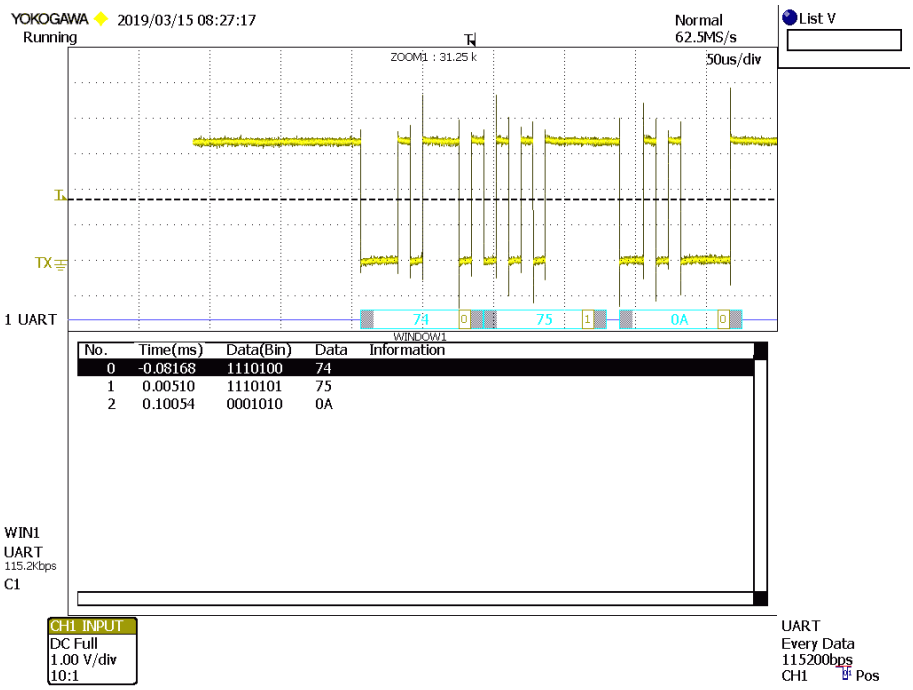


Figure 7.5: 7-O-1

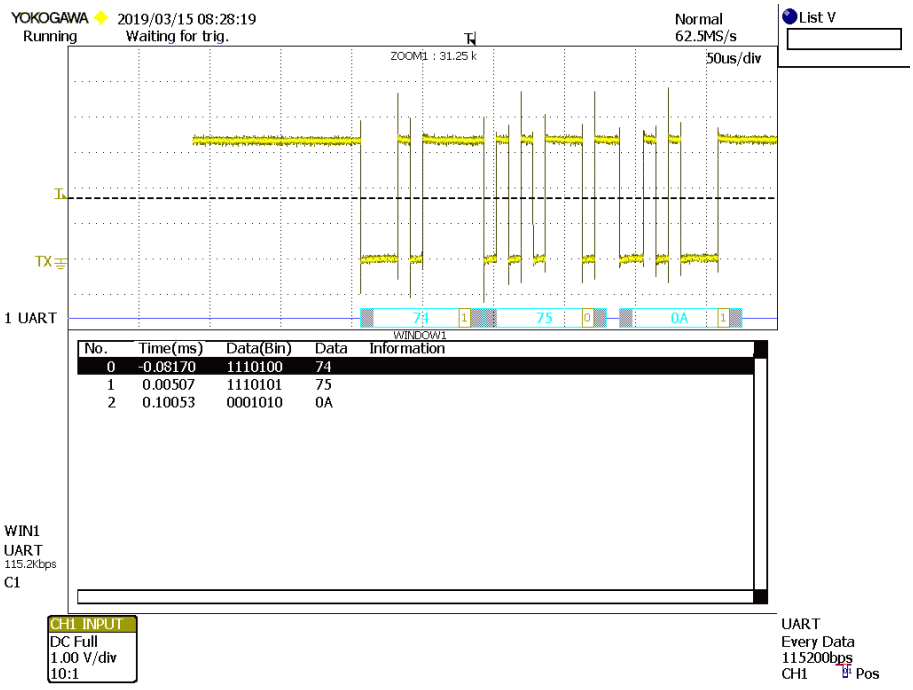


Figure 7.6: 7-E-1

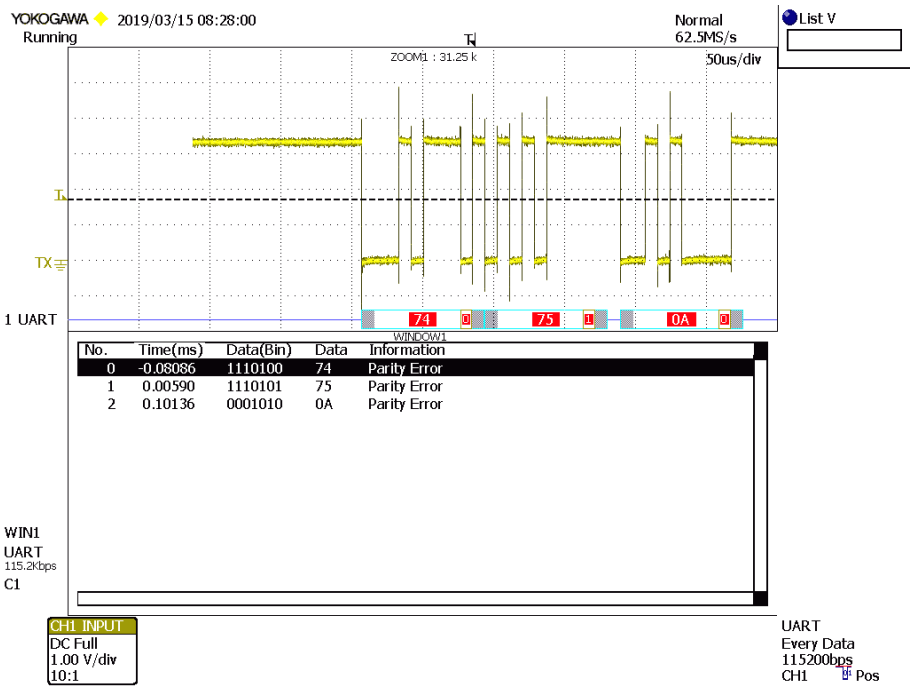


Figure 7.7: Wrong Parity Detection

8 Conclusion and Outlook

8.1 Conclusion

This thesis had two primary goals: make UART API more configurable and to integrate UART into the HIL testing infrastructure.

UART API now provides a new call that lets a user specify parity and the number of data and stop bits at runtime. This enhancement was asked by many RIOT OS community members and thus, is a long-awaited feature. The STM32 platform was the first to actually implement the new call and now other platforms can follow.

The PHiLIP project can now simulate a serial slave. It already proofed very useful during the UART API development. At first, it supported verification of the serial settings and then it helped during the refactoring stage.

UART tests are now a part of the continuous integration process. Since their inclusion, some bugs could be detected and fixed. The tests have not revealed more issues because UART interface currently provides very basic functionality and it is used extensively as a console interface. Hence, it is already tested well enough. But with more UART drivers getting new functionality or being refactored, the tests become more important.

The usage of Robot Framework as a testing tool made the tests readable and easy to understand due to a keyword-driven approach.

8.2 Outlook

Serial frame configuration is a feature that is available on all UARTs supported in RIOT OS. But there are a lot of other partly not common features that are supported on these MCUs. Hardware handshake, automatic RS485 transmitter control, polarity changing

for UART signals just to name a few. A flexible API needs to be developed to handle all these configurations.

Bibliography

[1] Dahlan Abdullah and Ilene Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., 2003. ISBN 0387951318.

[2] Ritesh Kumar Agrawal and Vivek Ranjan Mishra. The design of high speed UART. In *Information & Communication Technologies (ICT), 2013 IEEE Conference on*, pages 388–390. IEEE, 2013.

[3] Muhammad Shoaib Almas, Rujiroj Leelaruji, and Luigi Vanfretti. Over-current relay model implementation for real time simulation & hardware-in-the-loop (hil) validation. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 4789–4796. IEEE, 2012.

[4] Atmel. *SAM3X / SAM3A Series Datasheet*. Atmel Corporation, March 2015. URL http://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf.

[5] Jan Louise Axelson. *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links and Networks with Disk*. Lakeview Research, 1999. ISBN 0965081923.

[6] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal*, Dec 2018. ISSN 2327-4662. doi: 10.1109/JIOT.2018.2815038.

[7] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Wilfried Elmenreich and Martin Delvai. Time-triggered communication with UARTs. In *4th IEEE International Workshop on Factory Communication Systems*, pages 97–104, Aug 2002. doi: 10.1109/WFCS.2002.1159706.

[9] EXAR. *ST16C550 UART with 16-byte FIFO's*. EXAR Corporation, April 2005. URL http://www.mouser.com/ds/2/146/st16c550_501_040605-30673.pdf.

[10] Yi-yuan Fang and Xue-jun Chen. Design and simulation of UART serial communication module based on VHDL. In *2011 3rd International Workshop on Intelligent Systems and Applications*, pages 1–4. IEEE, 2011.

[11] Robot Framework Foundation. Robot framework, 2018. URL https://robotframework.org/. Last visited 2018-11-11.

[12] Martin Fowler. Continuous integration, 2006. URL https://www.martinfowler.com/articles/continuousIntegration.html. Last visited 2018-11-10.

[13] BM Hanson, MC Levesley, K Watterson, and PG Walker. Hardware-in-the-loop-simulation of the cardiovascular system, with assist device testing application. *Medical engineering & physics*, 29(3):367–374, 2007.

[14] Texas Instruments. *Texas Instruments CC2538 Family of Products User's Guide*. Texas Instruments Inc., May 2013. URL http://www.ti.com/lit/ug/swru319c/swru319c.pdf.

[15] Texas Instruments. *MSP430x1xx Family User's Guide*. Texas Instruments Inc., December 2016. URL http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[16] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher. Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA*, pages 1518 – 1532, 2007.

[17] Silicon Labs. *AN0059.0: UART Flow Control*. Silicon Labs Inc., January 2017. URL https://www.silabs.com/documents/public/application-notes/an0059.0-uart-flow-control.pdf.

[18] Mikael Laine. Design and implementation of a test environment for RFIC firmware. Bachelor's thesis, Dept. of Information Technology, Turku University of Applied Sciences, 2017.

[19] Jim A Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12:42–62, 1999.

[20] Sebastian Meiling. Robotfw-tests, 2019. URL https://github.com/RIOT-OS/RobotFW-tests. Last visited 2019-02-19.

[21] James Munns. CI for embedded systems, 2017. URL https://jamesmunns.com/blog/hardware-ci-overview/. Last visited 2018-11-11.

[22] Syed Nabi, Mahesh Balike, Jace Allen, and Kevin Rzemien. An overview of hardware-in-the-loop testing systems at Visteon. Technical report, SAE Technical Paper, 2004.

[23] Yaniv Nissenboim. How to shorten embedded software R&D time with continuous integration?, 2018. URL https://blog.jumper.io/continuous-integration-embedded-software-intro. Last visited 2018-11-11.

[24] proconX Pty. Diagslave modbus slave simulator, 2012. URL https://www.modbusdriver.com/diagslave.html. Last visited 2019-04-09.

[25] Philipp Rosenkranz, Matthias Wählisch, Emmanuel Baccelli, and Ludwig Ortmann. A distributed test system architecture for open-source IoT software. In *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*, pages 43–48. ACM, 2015.

[26] Marco AA Sanvido, Vaclav Cechticky, and Walter Schaufelberger. Testing embedded control systems using hardware-in-the-loop simulation and temporal logic. *IFAC Proceedings Volumes*, 35(1):453–458, 2002.

[27] Nordic Semiconductor. *nRF51 Series Reference Manual*. Nordic Semiconductor, October 2014. URL http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf.

[28] Philips Semiconductors. *SC16CXXXB baud rate deviation tolerance*. Philips Semiconductors, December 2004. URL https://www.nxp.com/docs/en/application-note/AN10333.pdf.

[29] STMicroelectronics. *STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs*. STMicroelectronics, January 2017. URL https://www.st.com/content/ccc/resource/technical/document/reference_manual/

c2/f8/8a/f2/18/e6/43/96/DM00031936.pdf/files/DM00031936.pdf/jcr:content/translations/en.DM00031936.pdf.

[30] STMicroelectronics. *STM32F446xx advanced Arm-based 32-bit MCUs*. STMicroelectronics, February 2018. URL https://www.st.com/content/ccc/resource/technical/document/reference_manual/4d/ed/bc/89/b5/70/40/dc/DM00135183.pdf/files/DM00135183.pdf/jcr:content/translations/en.DM00135183.pdf.

[31] Espressif Systems. *ESP32 Technical Reference Manual*. Espressif Systems, December 2018. URL https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[32] Phat Chau Tan. Automation testing with robot framework. Bachelor's thesis, Dept. of Information Technology, Helsinki Metropolia University of Applied Sciences, 2016.

[33] Nano River Technologies. Nanoboard, 2019. URL https://nanorivertech.com/nanoboard.html. Last visited 2019-02-19.

[34] Microchip Technology. *PIC32MX5XX/6XX/7XX Series Datasheet*. Microchip Technology Inc., June 2016. URL http://ww1.microchip.com/downloads/en/devicedoc/60001156j.pdf.

[35] Microchip Technology. *ATmega328PB Series Datasheet*. Microchip Technology Inc., February 2018. URL http://ww1.microchip.com/downloads/en/DeviceDoc/40001906C.pdf.

[36] Sebastian Vöst and Stefan Wagner. Towards continuous integration and continuous delivery in the automotive industry. *arXiv preprint arXiv:1612.04139*, 2016.

[37] Kevin Weiss. PHiLIP, 2018. URL https://github.com/riot-appstore/PHiLIP. Last visited 2019-02-19.

[38] Yegor Yefremov. modbus-riot-test, 2019. URL https://github.com/yegorich/modbus-riot-test. Last visited 2019-04-09.

A Appendix

```
1  int uart_mode(uart_t uart, uart_data_bits_t data_bits, uart_parity_t parity,
2  uart_stop_bits_t stop_bits)
3  {
4      assert(uart < UART_NUMOF);
5
6      isr_ctx[uart].data_mask = 0xFF;
7
8      if (parity) {
9          switch (data_bits) {
10             case UART_DATA_BITS_6:
11                 data_bits = UART_DATA_BITS_7;
12                 isr_ctx[uart].data_mask = 0x3F;
13                 break;
14             case UART_DATA_BITS_7:
15                 data_bits = UART_DATA_BITS_8;
16                 isr_ctx[uart].data_mask = 0x7F;
17                 break;
18             case UART_DATA_BITS_8:
19 #ifdef USART_CR1_M0
20                 data_bits = USART_CR1_M0;
21 #else
22                 data_bits = USART_CR1_M;
23 #endif
24                 break;
25             default:
26                 return UART_NOMODE;
27         }
28     }
29     if ((data_bits & UART_INVALID_MODE) || (parity & UART_INVALID_MODE)) {
30         return UART_NOMODE;
31     }
32
33 #ifdef USART_CR1_M1
34     if (!(dev(uart)->ISR & USART_ISR_TC)) {
35         return UART_INTERR;
```

```
36      }
37      dev(uart)->CR1 &= ~(USART_CR1_UE | USART_CR1_TE);
38  #endif
39
40      dev(uart)->CR2 &= ~USART_CR2_STOP;
41      dev(uart)->CR1 &= ~(USART_CR1_PS | USART_CR1_PCE | USART_CR1_M);
42
43      dev(uart)->CR2 |= stop_bits;
44      dev(uart)->CR1 |= (USART_CR1_UE | USART_CR1_TE | data_bits | parity);
45
46      return UART_OK;
47  }
```

Listing A.1: Source Code of the uart_mode Funtion

```
1   static inline void irq_handler(uart_t uart)
2   {
3   #if defined(CPU_FAM_STM32F0) || defined(CPU_FAM_STM32L0) \
4           || defined(CPU_FAM_STM32F3) || defined(CPU_FAM_STM32L4) \
5           || defined(CPU_FAM_STM32F7)
6
7           uint32_t status = dev(uart)->ISR;
8
9           if (status & USART_ISR_RXNE) {
10                  isr_ctx[uart].rx_cb(isr_ctx[uart].arg,
11                  (uint8_t)dev(uart)->RDR & isr_ctx[uart].data_mask);
12          }
13          if (status & USART_ISR_ORE) {
14                  dev(uart)->ICR |= USART_ICR_ORECF;    /* simply clear flag on
                        overrun */
15          }
16
17  #else
18
19          uint32_t status = dev(uart)->SR;
20
21          if (status & USART_SR_RXNE) {
22                  isr_ctx[uart].rx_cb(isr_ctx[uart].arg,
23                  (uint8_t)dev(uart)->DR & isr_ctx[uart].data_mask);
24          }
25          if (status & USART_SR_ORE) {
26                  /* ORE is cleared by reading SR and DR sequentially */
27                  dev(uart)->DR;
28          }
```

```
29
30 #endif
31
32         cortexm_isr_end();
33 }
```

Listing A.2: Source Code of the irq_handler Funtion

```
1  static error_t _rx_str(PORT_UART_t *port_uart) {
2          char *str = port_uart->str;
3          UART_HandleTypeDef *huart = port_uart->huart;
4
5          uint16_t rx_amount;
6          error_t err = ENOACTION;
7
8          rx_amount = _get_rx_amount(port_uart);
9          if (rx_amount >= 1) {
10                 if (port_uart->mask_msb) {
11                         for (int i = 0; i < strlen(str); i++) {
12                                 str[i] &= 0x7f;
13                         }
14                 }
15                 if (str[rx_amount - 1] == RX_END_CHAR
16                                 && _get_rx_amount(port_uart) != port_uart->
                                       size) {
17                         _update_rx_count(port_uart, strlen(str));
18                         HAL_UART_Abort(huart);
19                         HAL_UART_AbortReceive(huart);
20                         err = parse_input(port_uart->mode, str, port_uart->
                               size, port_uart->access);
21                         _update_tx_count(port_uart, strlen(str));
22                         HAL_UART_Transmit_DMA(huart, (uint8_t*) str, strlen(
                               str));
23
24                 }
25         }
26         return err;
27 }
```

Listing A.3: Source Code of the _rx_str Funtion

```
1 class PhilipAPI(LLMemMapIf):
2
```

```
3       ROBOT_LIBRARY_SCOPE = 'TEST SUITE'
4       ROBOT_LIBRARY_VERSION = get_version()
5
6       def __init__(self, port, baudrate):
7           super(PhilipAPI, self).__init__(PHILIP_MEM_MAP_PATH, 'serial', port,
                baudrate)
8
9       def reset_dut(self):
10          ret = list()
11          ret.append(self.write_reg('sys.cr', 0xff))
12          ret.append(self.execute_changes())
13          sleep(1)
14          ret.append(self.write_reg('sys.cr', 0x00))
15          ret.append(self.execute_changes())
16          sleep(1)
17          return ret
18
19      def setup_uart(self, mode=0, baudrate=115200,
20                     databits=serial.EIGHTBITS, parity=serial.PARITY_NONE,
21                     stopbits=serial.STOPBITS_ONE, rts=True):
22          ret = list()
23          ret.append(self.write_reg('uart.mode', int(mode)))
24
25          ret.append(self.write_reg('uart.baud', int(baudrate)))
26
27          if databits == serial.SEVENBITS:
28              ret.append(self.write_reg('uart.ctrl.data_bits', 1))
29          elif databits == serial.EIGHTBITS:
30              ret.append(self.write_reg('uart.ctrl.data_bits', 0))
31
32          if parity == serial.PARITY_NONE:
33              ret.append(self.write_reg('uart.ctrl.parity', 0))
34          elif parity == serial.PARITY_EVEN:
35              ret.append(self.write_reg('uart.ctrl.parity', 1))
36          elif parity == serial.PARITY_ODD:
37              ret.append(self.write_reg('uart.ctrl.parity', 2))
38
39          if stopbits == serial.STOPBITS_ONE:
40              ret.append(self.write_reg('uart.ctrl.stop_bits', 0))
41          elif stopbits == serial.STOPBITS_TWO:
42              ret.append(self.write_reg('uart.ctrl.stop_bits', 1))
43
44          # invert RTS level as it is a low active signal
45          if rts:
```

```
46              ret.append(self.write_reg('uart.ctrl.rts', 0))
47          else:
48              ret.append(self.write_reg('uart.ctrl.rts', 1))
49
50          # reset status register
51          ret.append(self.write_reg('uart.status', 0x00))
52
53          # apply changes
54          ret.append(self.execute_changes())
55          sleep(1)
56          return ret
57
58      def get_counters(self):
59          '''Get rx/tx counters.'''
60          ret = list()
61          ret.append(self.read_reg('uart.rx_count'))
62          ret.append(self.read_reg('uart.tx_count'))
63          return ret
64
65      def get_error_flags(self):
66          '''Get error flags.'''
67          ret = list()
68          ret.append(self.read_reg('uart.status.pe'))
69          ret.append(self.read_reg('uart.status.fe'))
70          ret.append(self.read_reg('uart.status.nf'))
71          ret.append(self.read_reg('uart.status.ore'))
72          return ret
```

Listing A.4: Source Code of PhilipAPI Class

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original