D5.1

Platform baseline benchmarks

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.1
Deliverable name	Platform baseline benchmarks
Date	December 2018
Dissemination level	public
Workpackage and task	5.1
Author	Mateusz Maciaś (PIAP)
Contributors	Adam Dąbrowski (PIAP), Tomasz Kołcon (PIAP), Tomasz Plaskota (PIAP)
Keywords	Microcontroller, NuttX, Benchmarking
Abstract	This deliverable summarizes work performed in task 5.1: Reference values for HW + OS and shows results of platform baseline benchmarking based on defined use cases. It is also a general introduction to benchmarking that will continue in tasks 5.2: Benchmarking of whole stack and task 5.3: Benchmark tooling for application developers.

Contents

1 Summary

This deliverable summarizes work performed in task 5.1 Reference values for HW + OS and shows results of platform baseline benchmarking based on defined use cases. It is also a general introduction to benchmarking that will be continued in tasks 5.2 Benchmarking of whole stack and task 5.3 Benchmark tooling for application developers.

This document starts with introduction to goals, concepts and limitations important for benchmarking. This is followed by description of what we wanted to measure, further expanded in how we measured relevant values. In this section, all the tools employed in the process will be introduced. Then, the setup used for benchmarking is detailed. Subsequently, results are presented for each benchmark along with comments. Finally, conclusions are presented.

2 Acronyms and keywords

Term	Definition
ROS	Robot Operating System
SWO	Single Wire Output
SWD	Serial Wire Debug
SWV	Serial Wire Viewer
JTAG	Joint Test Action Group (also name of interface)
ITM	Instrumentation (or Instruction) Trace Macrocell
ETB	Embedded Trace Buffer
MTB	Micro Trace Buffer
DWT	Debug, Watchpoint and Trace
TPIU	Trace Port Interface Unit
OS	Operating System
RTOS	Real Time Operating System
HW	HardWare
IP	Internet Protocol
TCP	Transmission Control Protocol
RAM	Random Access Memory
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks.
I/O	Input/Output
ETM	Embedded Trace Macrocell

3 Methodology

The work presented in this document is build on top of preliminary platform assessment detailed in D2.1: Report on the reference hardware development platforms. For example, this report pro-

vides power consumption values for NuttX on Olimex and Discovery boards, compared for different modes of NuttX power manager. Such preliminary measurements will be expanded upon, while other types of benchmarks have no precursors in earlier deliverables. Results described in this document will in turn provide a baseline for further micro-ROS benchmarking.

It is important to note, that this document is only first part of benchmarking process, and will be improved on further when scenarios and micro-ROS codebase progresses. With final delivery of benchmarking results planned in D5.2 MicroROS benchmarks and D5.3 MicroROS benchmarking and validation tools.

Goals of task 5.3 have to be taken into consideration. As part of our work, publicly available toolkit will be released. This is why all tooling used for benchmarking is looked on, to check their availability for wider community. For example we don't want to use expensive proprietary JTAG/SWD based analysis software and hardware tools, as it is hard to expect that academic and hobbyist part of ROS community would adopt them. Note that this goal will not be reached in scope of this document (but at a point when D5.3 MicroROS benchmarking and validation tools is released). This document will state requirements for this tool set, based on real needs.

3.1 Goals for benchmarking

Benchmarking is a method of quantifying behavior of system. In embedded system this is always focused on resource usage (clock cycles, memory, power, etc…). In benchmarking methodologies are developed to get consisting and representative results, but there is always a trade off, between complication of benchmark and how easy it is to use and understand. Developing benchmarks is also problematic because it requires to imagine possible way users might use system/code/library later on.

As benchmarking goals vary, from manufacturers of chips trying to show how far ahead they are from competition, to comparing specific implementation of communication library to in specific use case [1]. In this document, we are focused on baseline benchmarking, ahead of micro-ROS development, looking int platform characteristics, and simplest possible scenarios.

3.2 A short introduction to vocabulary

Real-time performance can be understood as the speed with which an RTOS can complete its functions in response to an event. **Real-time constraint** states that the system needs to guarantee response within specified time constraints, which are also called deadlines. Aside from the guaranteed response time, useful characteristics include typical and expected response times, which may be significantly shorter. Real-time systems are classified by the consequence of missing a deadline, where **hard real-time** systems can cause total failure if deadline is missed, while **soft real-time** only cause a drop in quality. Real-time performance can be context-sensitive and proper evaluation needs to include **stress** testing, which consists of measuring characteristics while the system is performing other tasks, e.g resulting in a heavy I/O use.

Synthetic benchmarks are isolated tests designed to yield repeatable results and to test individual parts with well-defined operations. **Real world benchmarks** give the entire system a workload that resembles real tasks and focus on holistic results that are easy to relate to intended use of the system.

3.3 Limitations

- The method used to determine time is limited by available clock resolution and precision, as well as any overhead introduced by measurement. It is the most important limitation for accuracy and resolution of all temporal metrics.
- Limited context space to search (can't test all possible contexts). It's important to guess which contexts are crucial. This process of guessing can be gradually improved using conclusions from previous results and knowledge of the system.
- Only limited combinations can be tested (HW + RTOS, including firmware version, RTOS version, compiler).
- There are limits to drawing conclusions from stochastic behavior testing of the system - it can't determine guarantees with certainty, instead it can make weaker, probabilistic statements (such and such occurrence was not observed so it can be determined that it's probability is not higher than X in exactly the same circumstances).

3.4 The benchmarking setup overview

The reference platforms are Olimex STM32-E407 and STM32 L1 Discovery as described in OFERA deliverable D2.1. In scope of this documents those will be called Olimex and Discovery boards. These platforms have already been validated in the scope of that deliverable as suitable for the project's needs, which have been estimated through requirements gathering, a first comprehensive iteration on architecture design, and footprints of existing parts or components such as Micro RTPS. The benchmarking process for the HW+RTOS part aims to deliver a baseline for further benchmarks of the micro-ROS stack, supplying a comparison to the raw state and enabling measurements of various overheads.

Since both platforms are supported by NuttX and because of reasoning presented in the D2.1, NuttX RTOS will be the choice for benchmarking setup as well. The NuttX power manager enables a stand-by mode, with some results obtained in D2.1 showing that it can save about 30% of the power compared to the normal operation mode on the Olimex board.

Olimex is intended to support the full micro-ROS stack while the STM32 L1 Discovery will support a sensible subset of micro-ROS modules with limited data, processing and communication footprints.

3.5 Preliminary benchmarking

ALR has conducted some benchmarking to determine the validity of base platform choices. For both Olimex LTD STM32-E407 and STM32L1 Discovery, power consumption and memory usage were measured in several configurations. These benchmarks were done with the NuttX RTOS deployed. Characteristics were measured with different setups (e.g. with 6LoWPAN) and settings (e.g. NuttX Power Manager StandBy mode). More details are available in deliverable D2.1 Report on the reference hardware development platforms. Our benchmarks concerned with power consumption and memory use will validate and extend these findings.

4 Tools used

4.1 Performance and resources use monitoring support in NuttX

4.1.1 CPU load monitoring

CPU load monitoring can be enabled through RTOS Features > Performance Monitoring > Enable CPU load monitoring (config SCHED_CPULOAD). When enabled, timer interruptions will coarsely determine the percentage of time when CPU is idle or busy and the statistics over time can be a useful metrics.

Depending on configuration:

- SCHED_CPULOAD_EXTCLK (default n): sampling is done by system clock (if N) or by external clock (if Y).
- SCHED_CPULOAD_TICKSPERSEC (default 100): a rate of interrupts for external clock. Better to change it from the default.
- CPULOAD_ONESHOT (default n): use MCU-specific oneshot timer as external clock. CPULOAD_ENTROPY controls randomness and range of intervals.
- CPULOAD_PERIOD (default n): use MCU-specific period timer as external clock.

4.1.2 System performance monitor hooks

System performance monitor hooks can be enabled through menuconfig in RTOS Features > Performance Monitoring > System performance monitor hooks (config SCHED_INSTRUMENTATION). This enables instrumentation in scheduler to monitor system performance. Board-specific logic must provide a series of sched_note_* functions. These are internal OS interfaces called at critical locations, thus not much can be done there (no syslog output too).

If that instrumentation is enabled, there are several configuration of interest (turned off by default):

- SCHED_INSTRUMENTATION_PREEMPTION enables additional hooks for changes to preemption state.
- SCHED_INSTRUMENTATION_CSECTION enables additional hooks for entry and exit for critical sections.
- SCHED_INSTRUMENTATION_SPINLOCKS enables additional hooks for spinlock state (active wait locks).

SCHED_INSTRUMENTATION_BUFFER gives a memory buffer to capture scheduler instrumentation data. This makes sched_note_* interfaces not needed, instead the circlular buffer catches all of them. Also, bufferring minimizes the impact of the instrumentation on the behavior of the system. A function ssize_t sched_note_get(FAR uint8_t *buffer, size_t buflen) must be called and dispose of information quickly. The buffer size can be set through SCHED_INSTRUMENTATION_BUFFER. However, if critical section or spinlock instrumentation is turned on, the sched_note_get interaface can't be used (calls would fill the buffer more than remove from it). An important header for all this is include/nuttx/sched_note.h

4.1.3 Interrupts monitoring

Interrupts monitoring can be enabled through RTOS Features > Performance Monitoring > Enable IRQ monitoring (config SCHED_IRQMONITOR). This will result in counting of interrupts from all sources. Counts are available through procfs file system (file irqs).

4.1.4 Memory

When more features are enabled (such as 6LoWPAN), the available heap size is decreased as static memory usage increases. Thus, depending on the configuration, the total available memory on NuttX will vary. /proc/meminfo is the simplest way to probe the use of memory. Also, a tool is available (tools/showsize.sh) to show top 10 consumers of FLASH and SRAM. Command line arm-none-eabi –print-size can be used to show that as well.

Memory use of selected configurations was shown in D2.1 deliverable.

4.1.5 TCP/IP stack:

NuttX has some documentation, and offers some commands for testing of the stack performance[2][3].

4.2 External tools for benchmarking

This chapter discusses tools used to perform external measurement. This is necessary supplement to facilities build into operating system and hardware platforms, that allows obtaining more accurate results and having less impact on measured system. Following measurement will be discussed:

- Power consumption measurement by external equipment,
- Using some I/Os to monitor state,
- Use debugging features (JTAG/SWD) build into processor, but not accessible from NuttX [4][5],
- Measurement of radio emission in some scenarios.

4.2.1 Tools used

To measure power consumption external hardware is used. This measurement could be done on board, or by small peace of hardware connected to board. For example external circuit for measuring voltage and current can be connected to microcontroller analog to digital converter input. This would have impact on software running, and object being benchmarked. Also accuracy of such measurement could not be ideal.

Note that we want to measure power consumed by benchmarked board, but this has to be measured in time. Also some analysis has to be performed on board schematic, to look into efficiently of power conversion system (if present) and what is consuming power.

Possible tool for measuring power in time is oscilloscope, with some additional circuitry for measuring current (series resistor). Some simpler alternatives will be investigated further in D5.3.

External I/O monitoring might less influence execution than using NuttX features, or be used to check how NuttX internal mechanism are affecting benchmarked system. For monitoring I/O, also oscilloscope can be used. Traditional oscilloscope might be limited in number of channels available, so more feasible option might to use logic analyser. Also mixed signal devices (also called mixed signal oscilloscopes) are available, and might solve this issue.

For radio communication scenario, it would be useful to know when message actually leaves radio transceiver. Regular oscilloscope, probably will not have enough bandwidth, and more specialized equipment has to be used. With spectrum analyser being as expensive as they are, we are leaning using some of software defined radio approach, in which relatively cheap radio frontend is connected to computer for processing, and most of this processing is done in software.

If various equipment is used, proper care must be taken to ensure that timing between them is synchronized. Simple scenario would be to use single I/O to mark start of benchmarking, and connect it to all equipment used.

Tools:

- Oscilloscope: Rigol DS1074Z:
  - It has mixed signal option, but our has only 4 channels,
  - Current probe for indirect current measurements,
  - PC software tool for collecting data [6],
  - Script for displaying measurements in human readable format.
- Mixed signal scope: Saleae Logic Pro 8.
- Logic analyzer: LA1010, with sigrog software [7].
- Software defined radio: USRP B200.
- JTAG/SWD:
  - SWD build into Discovery,
  - Olimex ARM-USB-OCD-H,
  - ST-LINK / V2.
- Custom shunt resistor adapter (mounted instead of 2.54mm jumper).

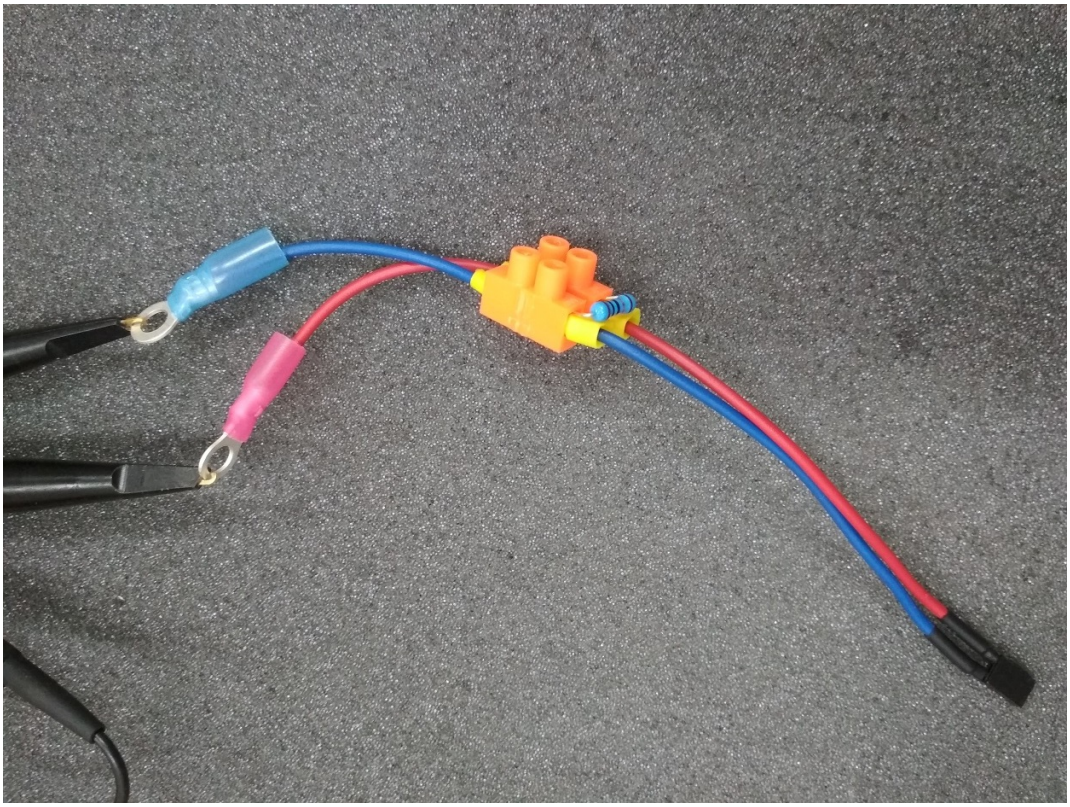

Figure 1: Custom shunt resistor adapter

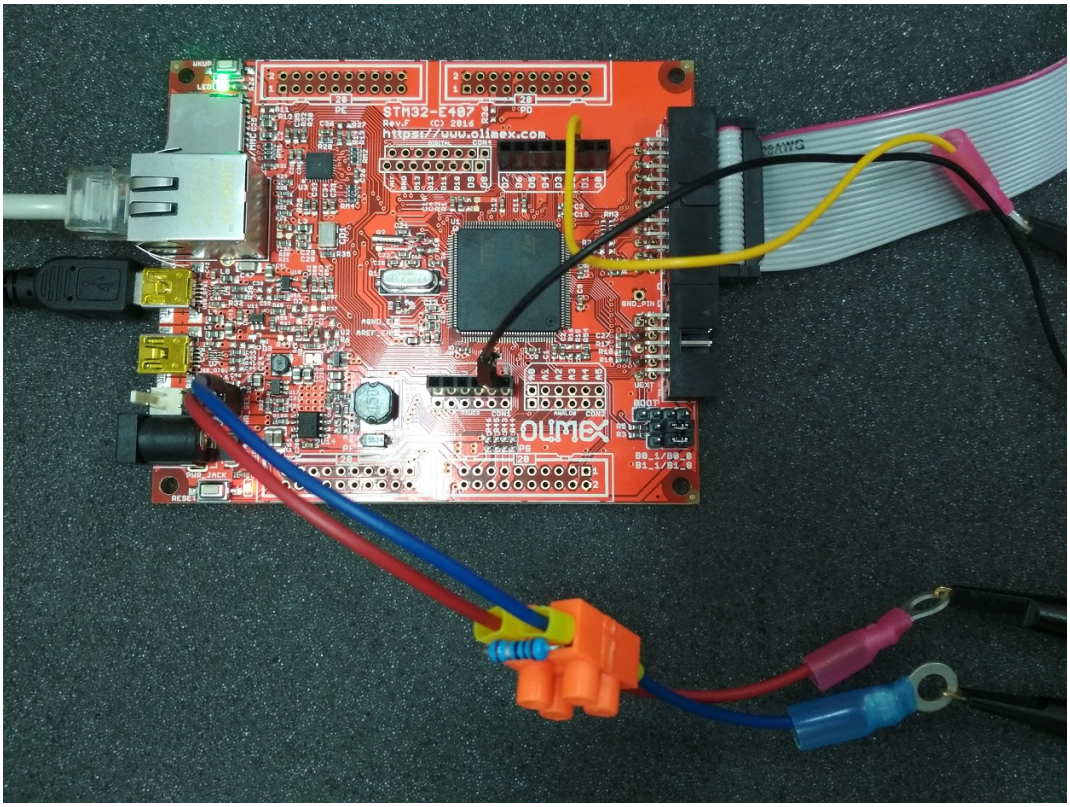

Figure 2: Olimex STM32-E407 measurements setup

For measuring power, EMBC in their ULPMark Measurement Methodology uses ST X-NUCLEO-LPM01A, that has following parameters[8][9]:

- Programmable voltage source from 1.8 V to 3.3 V
- Static current measurement from 1 nA to 200 mA
- Dynamic measurements:
  - 100 kHz bandwidth, 3.2 Msps sampling rate
  - Current from 100 nA to 50 mA
  - Power measurement from 180 nW to 165 mW

4.2.2 STM32-E407 Power supply

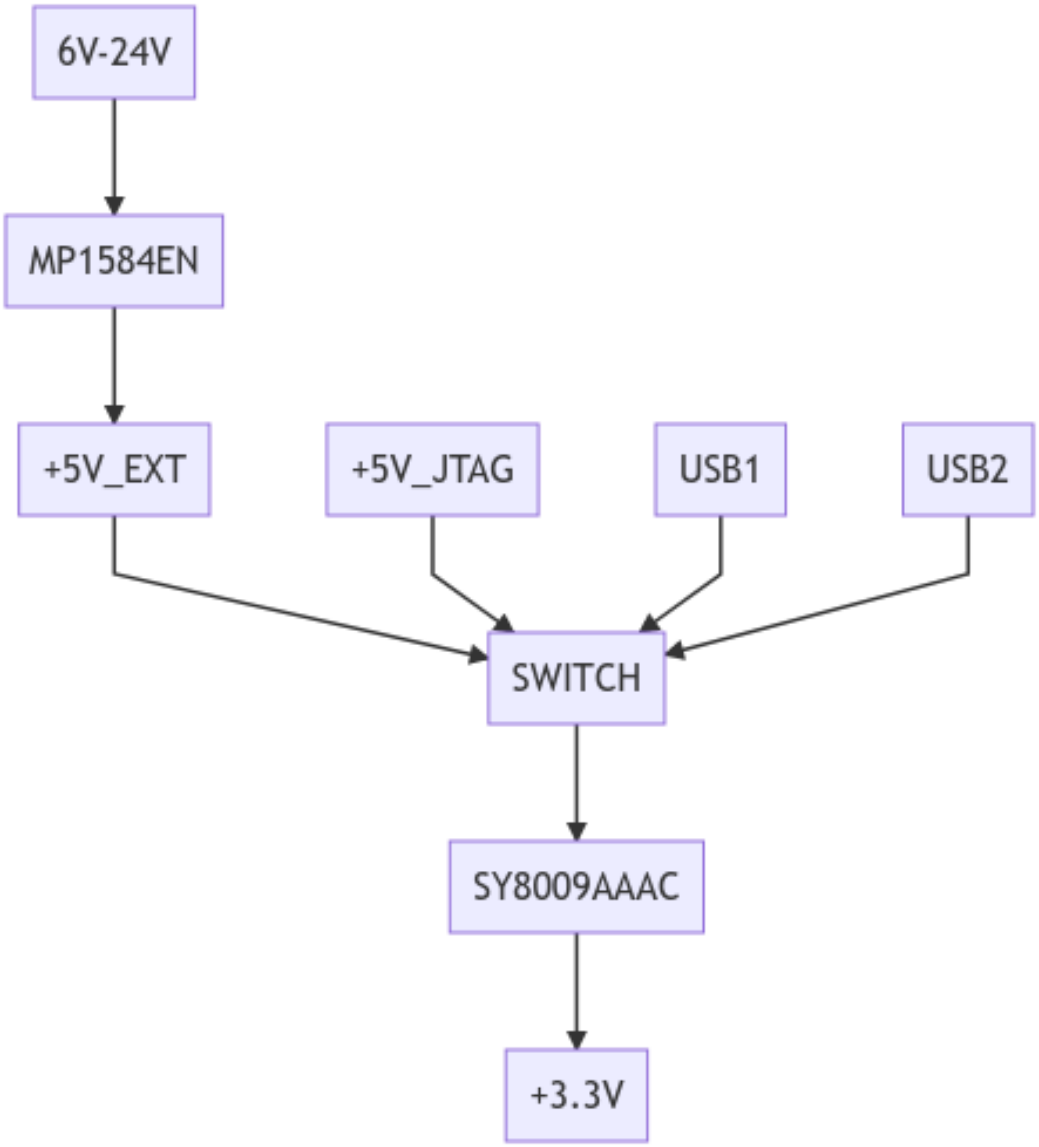

Figure 3: Simplified Diagram of Olimex STM32-E407 power supply circuit.

- MP1584EN - this is DC to DC switching converter, has efficiency roughly between 70 and 90% (depending on load current and input voltage)..
- SY8009AAAC - this is DC to DC switching converter, has efficiency between 82 and 95%.

On this board there is no easy way of measuring current powering microcontroller only, as there is no infrastructure for that. Easiest place to measure is on SWITCH itself (before second DC/DC converter). Measuring there means that current consumed by entire board is measured (including ethernet PHY, LED's, pull-up resistors and everything connected to board), and DC/DC converter efficiency has to be taken into consideration.

4.2.3 STM32L1 Discovery Power supply

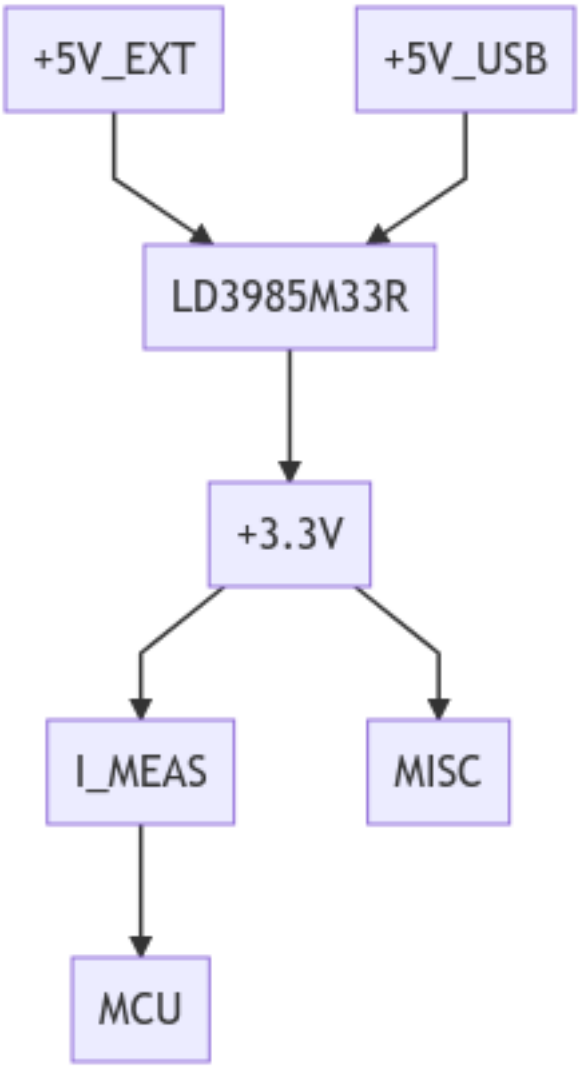

Figure 4: Simplified Diagram of STM32L1 Discovery power supply circuit.

- LD3985M33R - this is linear voltage regulator. Loss is proportional to current passed.

This board has build in current measurement circuit (with option of being bypassed). It measures current consumed by microcontroller itself. This point also allows easy connection for external measurement.

5 Configurations

5.1 Communication configuration specification

Base on D1.7 Reference Scenarios and Technical System Requirements Definition, three basic simple communication configuration were derived. Those are used as basic configuration for platform benchmarking. Benchmarking scenarios focus on benchmarking many of aspects of communication from/to chosen platforms. Generally at the point of getting baseline we do not try to simulate use-cases, but only to keep them in mind while choosing parameters for benchmarking scenarios, such as frequency, power consumption, data size etc.

5.1.1 Serial setup

First communication setup benchmarked is serial protocol, which will be most extensively used in Domestic Outdoor Robots (DOM) use-case. Other scenarios might choose to utilize this protocol as well, due to simplicity, however usage there will be much more limited and therefore configuration will focus on this particular use case.

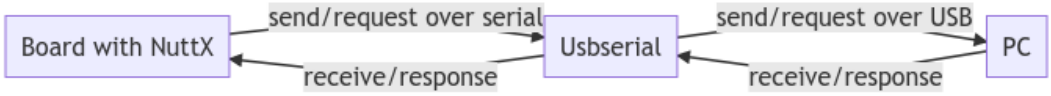

Figure 5: Graph of serial configuration.

5.1.2 TCP/IP setup

This setup applies only to Olimex STM32-E407 board. Benchmarking here is focused on Modular Arm (MA) use-case.

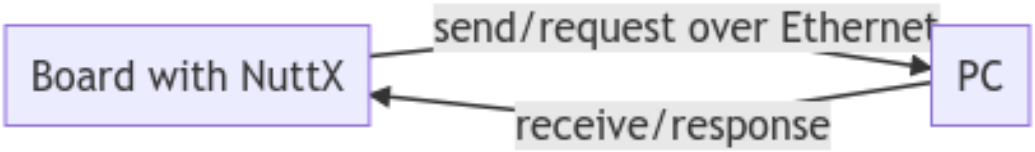

Figure 6: Graph of TCP/IP configuration.

5.1.3 6LoWPAN radio setup

This is the main focus of communication for other two use-cases, being Smart Warehouse (SW) and Autopilot Drone (AD). These use-cases will likely utilize other communication protocols due to more

actors present in these use-cases, however 6LoWPAN radio is most critical communication protocol used.

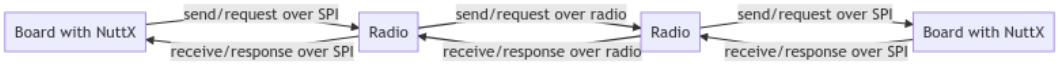

Figure 7: Graph of 6LoWPAN configuration.

6 Detailed benchmarking scenarios

Benchmarks are partially performed inside software, which adds overhead on top of some metrics. This is a subject treated with care and we make sure to limit it to reasonable amount. During scenario benchmark data is only collected. Analyzing and displaying afterwards to make it to not affect benchmark. Some parts might not have overhead at all, but results like total time of running scenario are definitely affected.

For each scenario simplest possible application were written, and NuttX code was augmented with benchmarking code. This provided information such as this (example from serial test):

```
nsh> serial /dev/ttyS0 512
configured gpio
Reset memory usage - total: 11168
usage diff: 0.05, idle usage diff: 0.05
Working with /dev/ttyS0
Received 512
Recieved echo from server after: 18347 ticks, 73388 us
func app_write: 27000us
func fwrite: 26992us
func app_read: 46384us
func uart_read: 36us
func fread: 42164us
func lib_fread: 40648us
func lib_take_sem: 708us
func lib_wrflush: 2008us
func _NX_READ: 34140us
func nx_read: 1400us
func lib_fwrite: 26988us
func lib_take_sem: 0us
func lib_rdflush: 8us
func lib_fflush: 26916us
func uart_recv: 164us
usage diff: 2.63, idle usage diff: 2.62
start, stop, suspend, resume, preemplock, preempresume: 0, 0, 107, 108, 0, 0
Received 512
Recieved echo from server after: 18340 ticks, 73360 us
```

What is happening under the hood is shown on diagrams in following chapters.

6.1 Scenario for serial

Details on internal implementation of serial are shown in next two chapter (divided into sending and receiving part). This was done based on code analysis. Note that this is blocking implementation.

6.1.1 Sending data

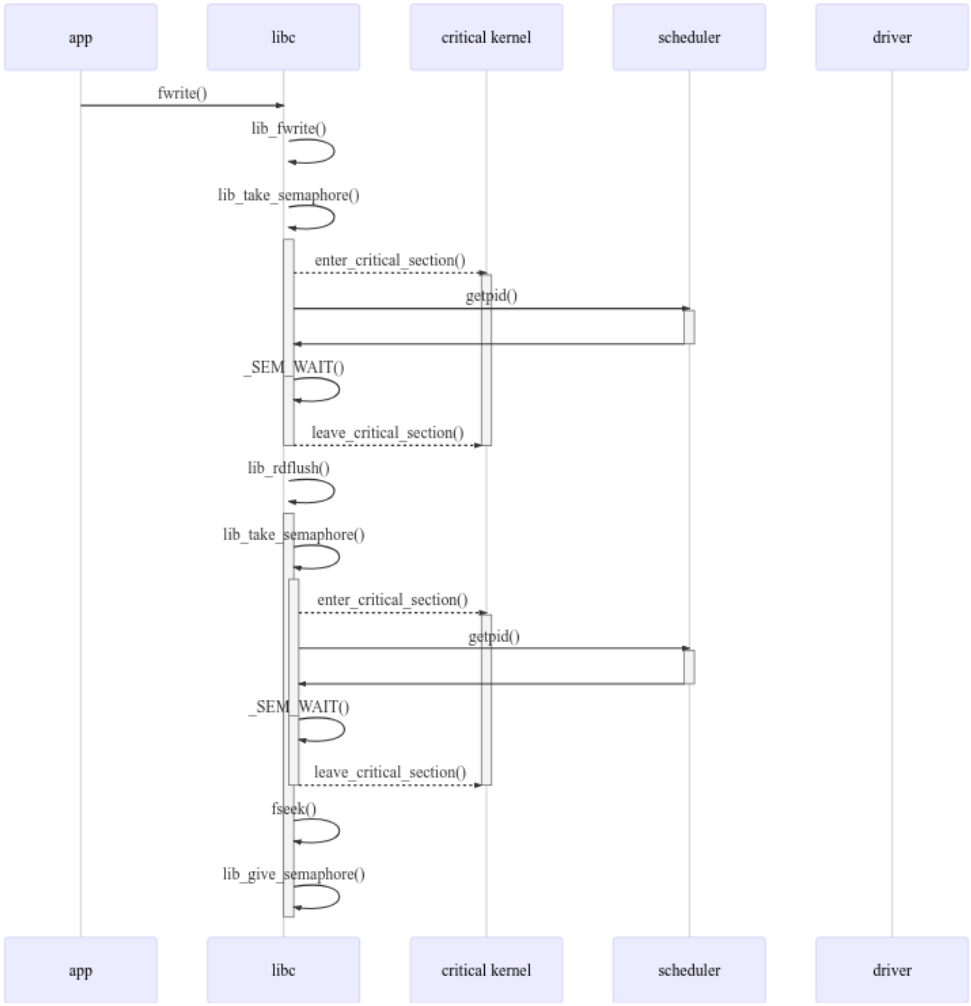

Figure 8: Diagram of serial send implementation.

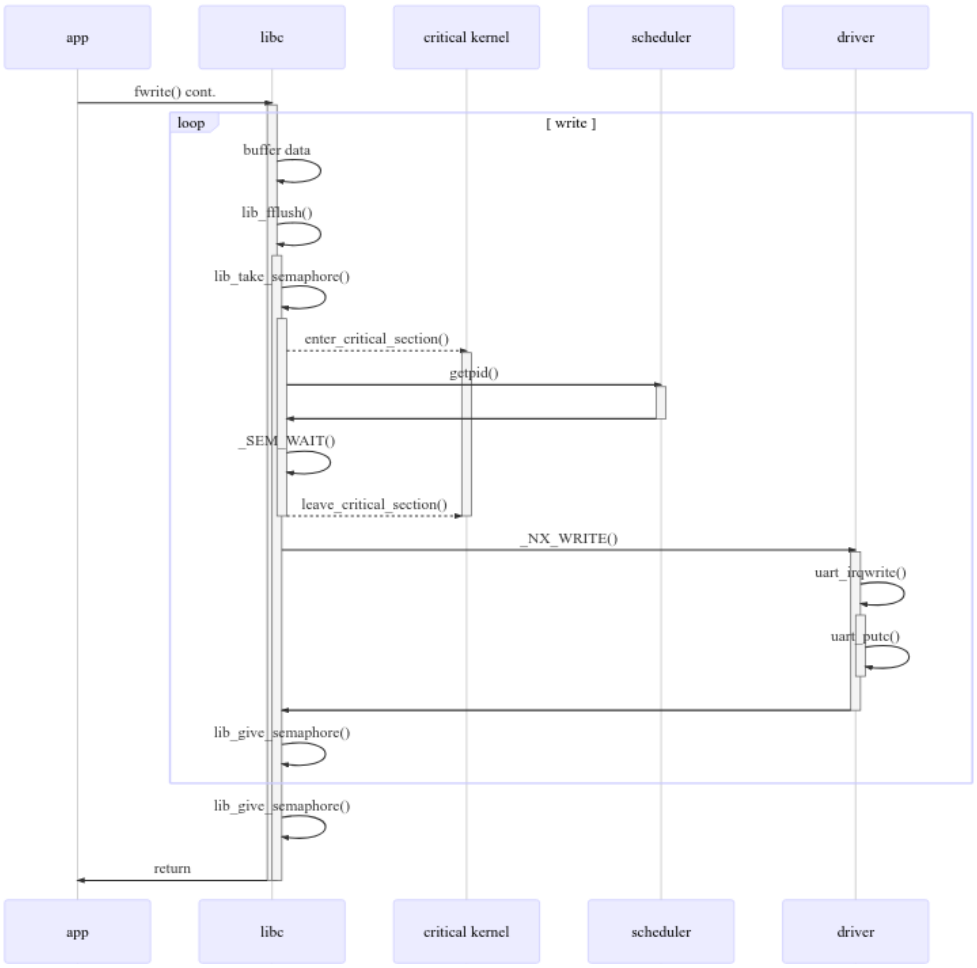

Figure 9: Diagram of serial send implementation (continued).

6.1.2 Receiving data

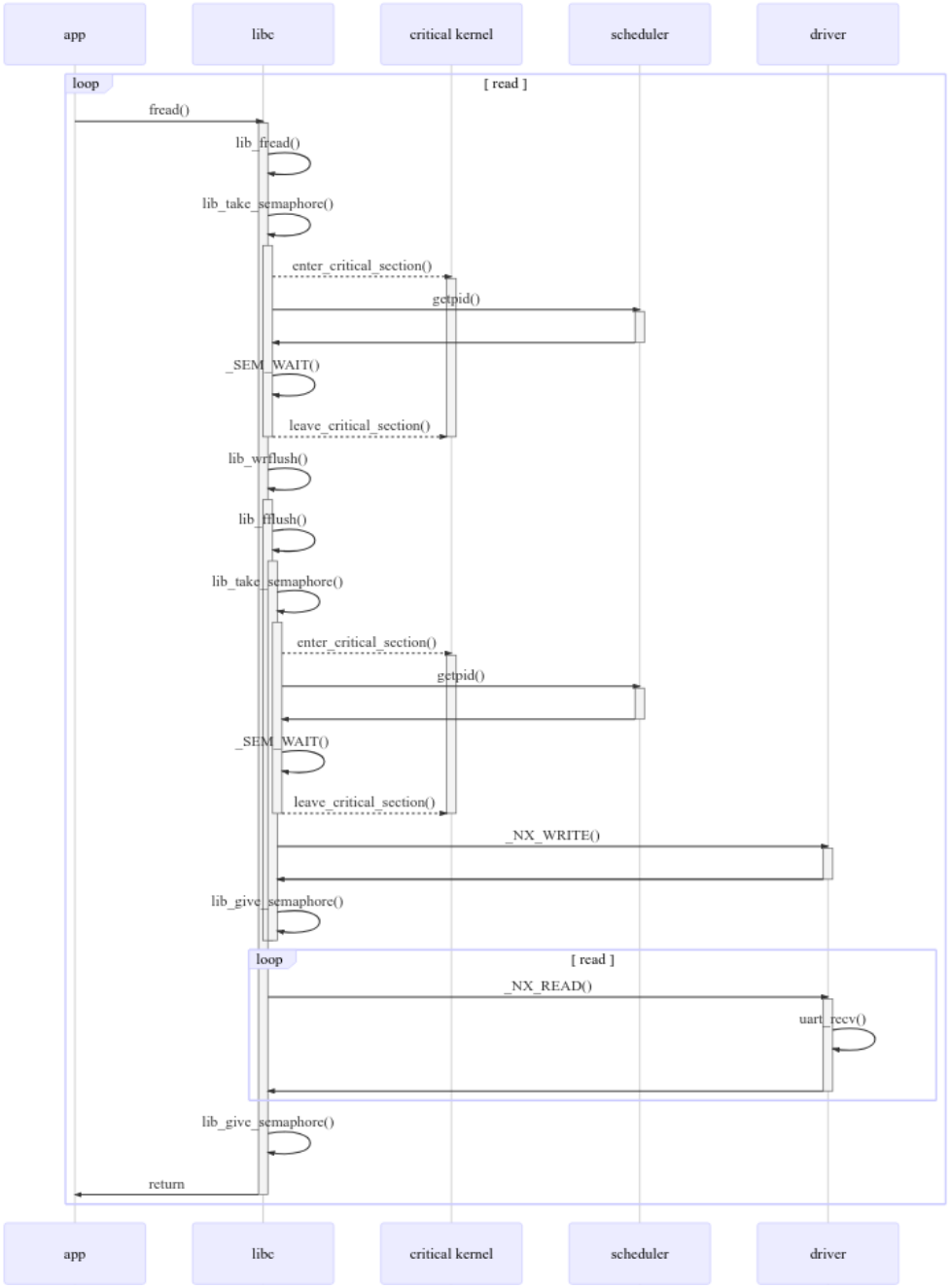

Figure 10: Diagram of serial receive implementation.

6.2 Scenario for tcp/ip

General benchmark for tcp/ip consists of 3 elements - sending data, polling for response and receiving response. Before measuring those elements, there is obviously initial setup performed including connecting to socket on server and preparing data to send, which is not part of results of benchmark.

Data collected here is (based on general characteristics):

- Latency for OS internals:
  - time taken for noticeable kernel/driver functions,
  - cpu_load for application and application+work_queue,
  - preemption count by scheduler from SCHED_INSTRUMENTATION.
- Latency for TCP/IP stack:
  - time taken for single roundabout message exchange,
  - time taken per element (send, poll, recv).

6.2.1 Sending data

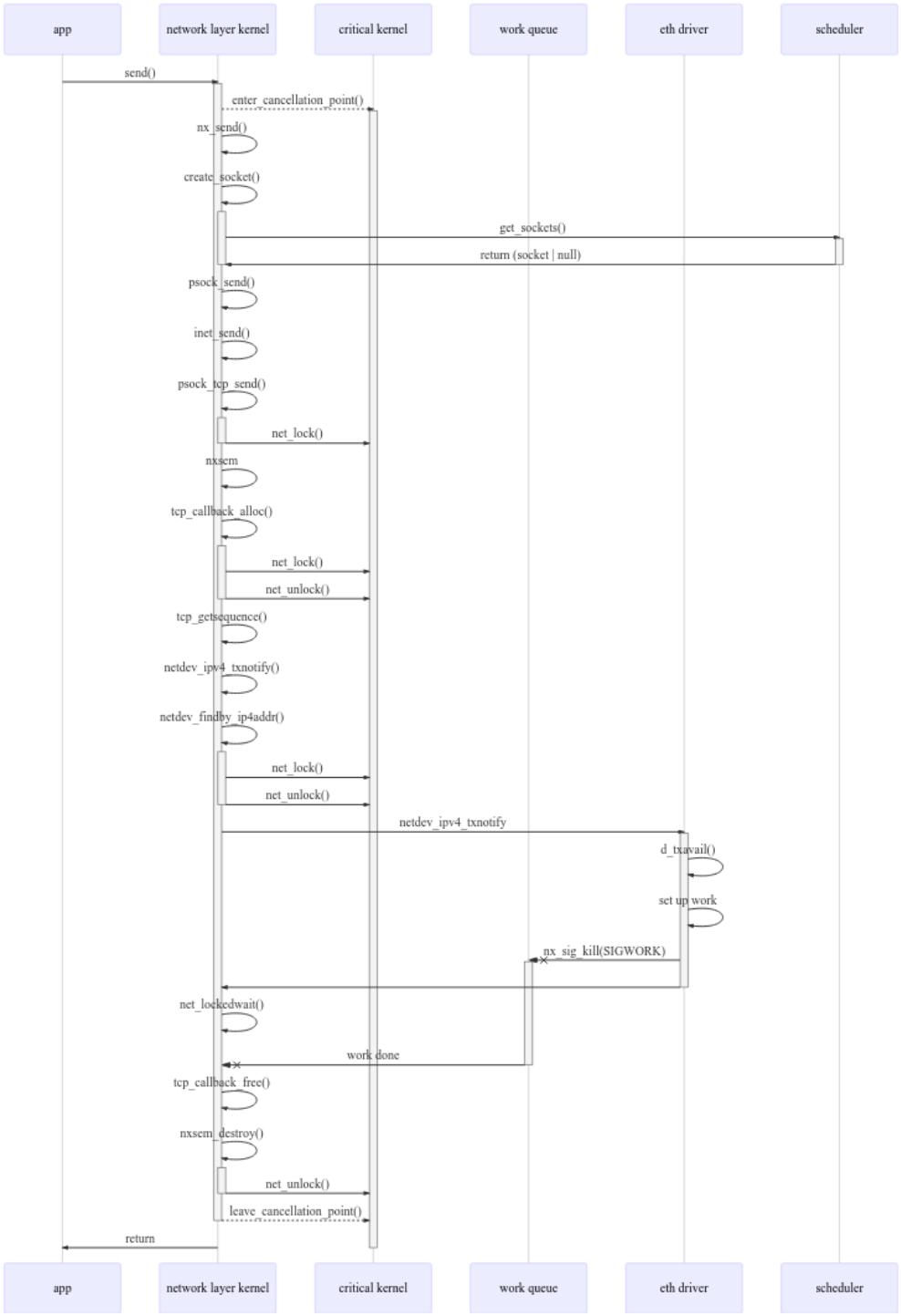

Figure 11: Diagram of TCP/IP send implementation.

6.2.2 Polling

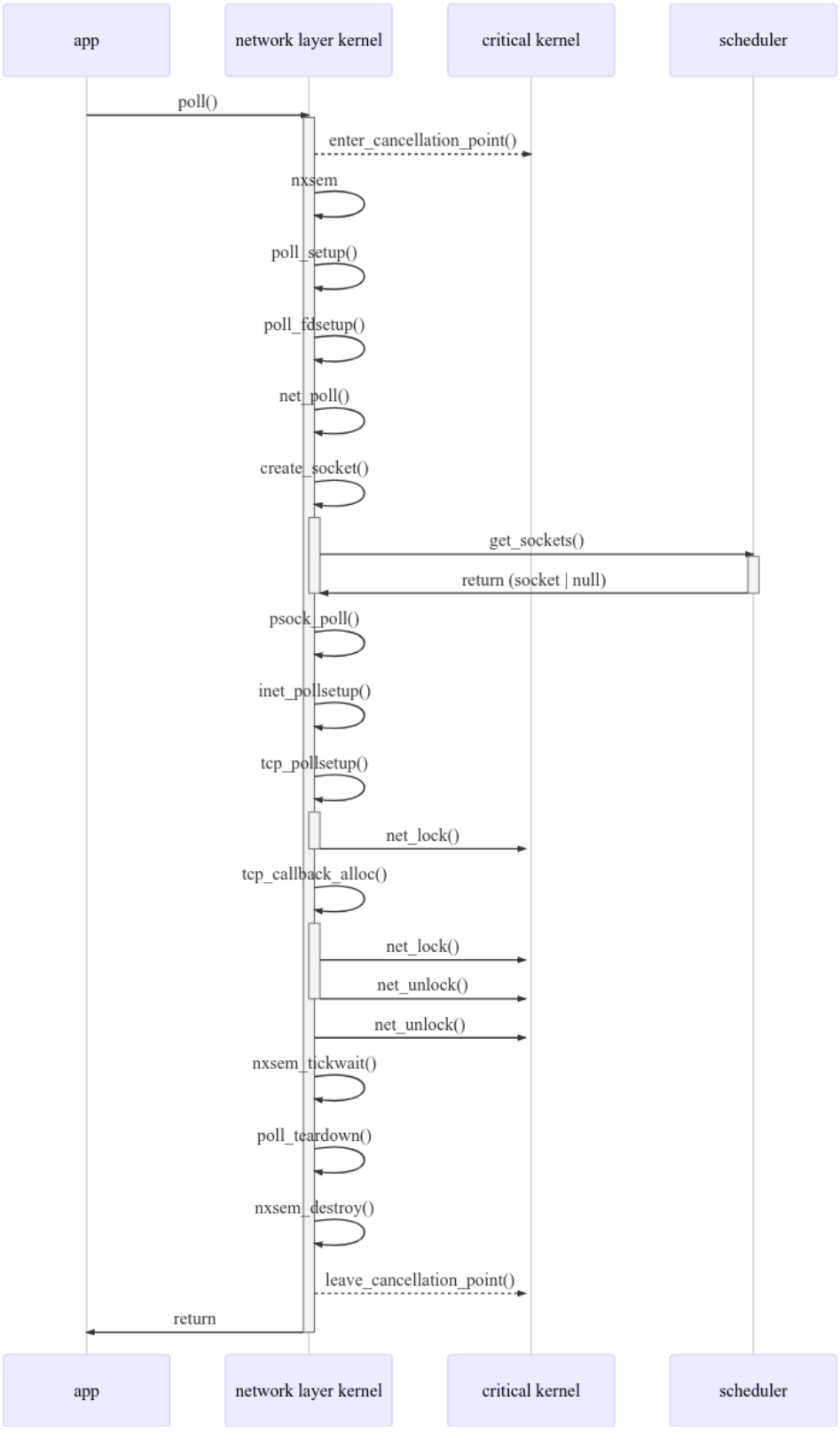

Figure 12: Diagram of TCP/IP polling implementation.

6.2.3 Recieving

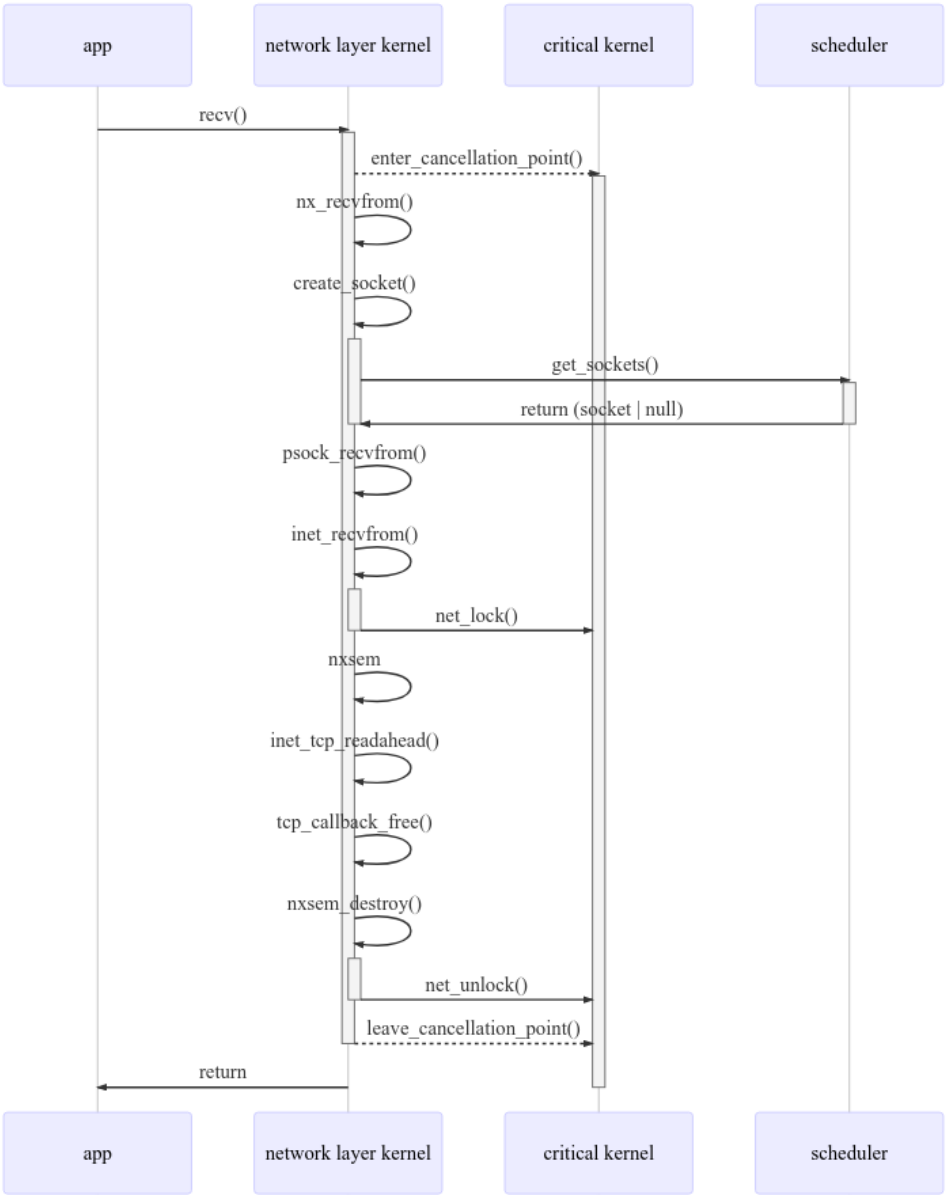

Figure 13: Diagram of TCP/IP receive implementation.

6.3 Scenario for 6LoWPAN

General benchmark for radio includes two end devices. We are using simplified example for udp client/server from NuttX examples. Baseline scenario is a client sending data to server, server receiving data, immediately sending bounceback packet and client receiving response. This means there are two elements to benchmark: sending data and receiving data. Initial setup of server/client is not a part of baseline benchmark. A ll data is collected from client, except there is additional data available from server which shows how long it took server between receive and sending back.

Data collected here is similar to one in TCP/IP scenario.

6.3.1 Sending data

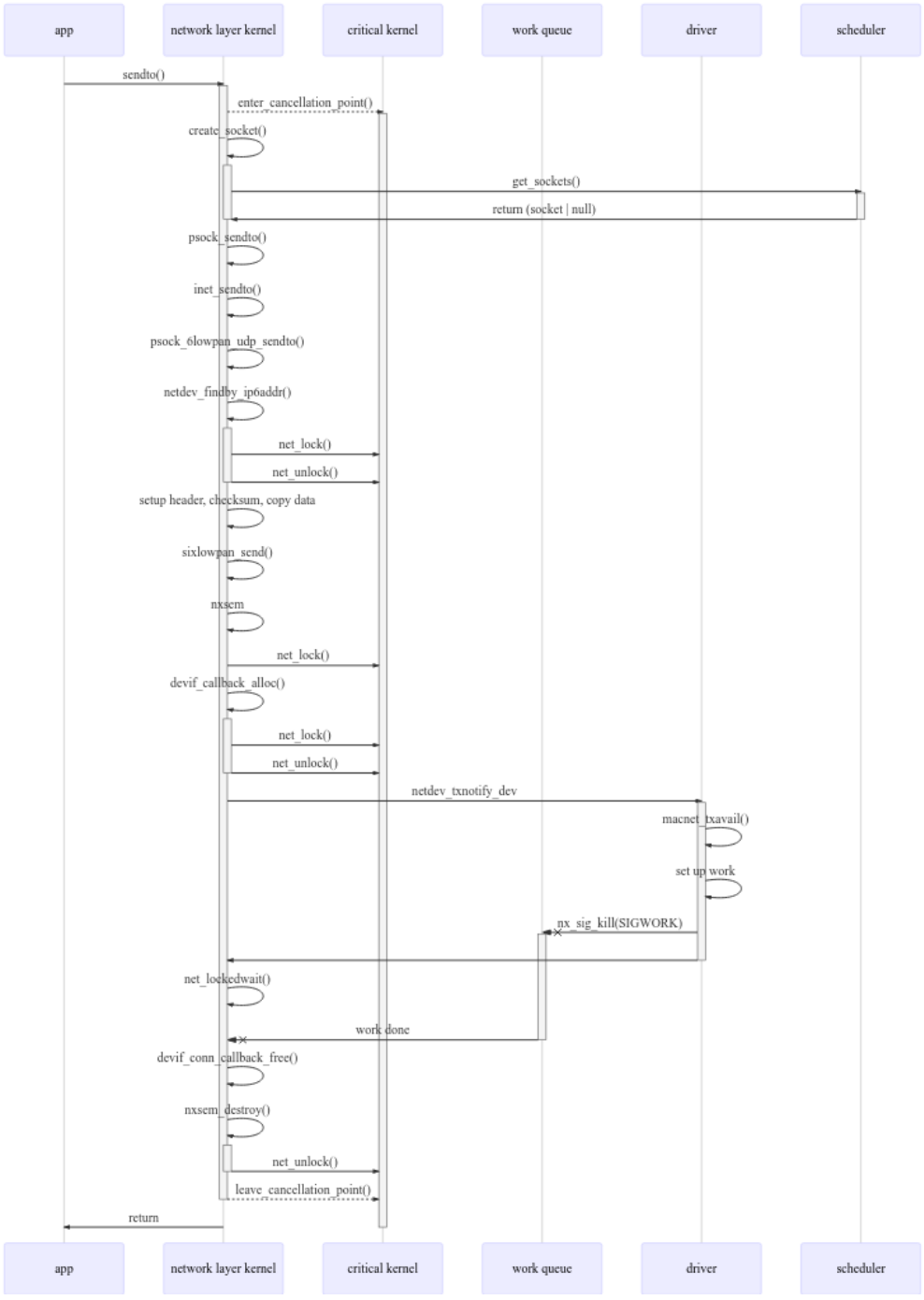

Figure 14: Diagram of 6LoWPAN send implementation.

6.3.2 Recieving data

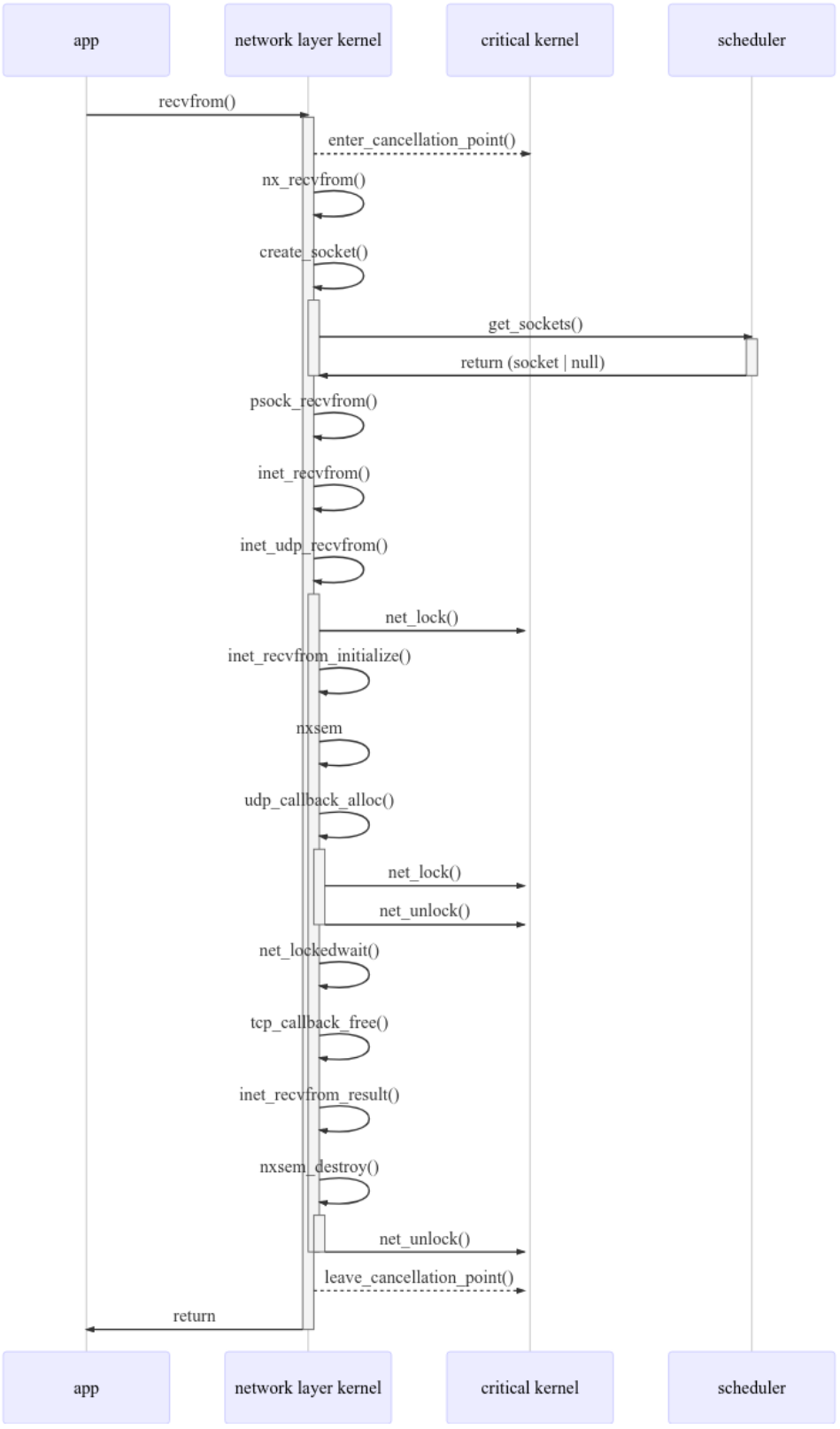

Figure 15: Diagram of 6LoWPAN receive implementation.

7 Benchmark results for Olimex board

This chapter shows results of performed benchmarking. Some immediate conclusions can be found near specific plots, in general those are gathered in next chapter.

7.1 Serial

7.1.1 512 bytes

This test consisted of sending and receiving 512 bytes of data using serial, each time repeated 100 times. Note that maximal sustainable frequency is 14Hz (due to 115200bps bitrate on serial link).

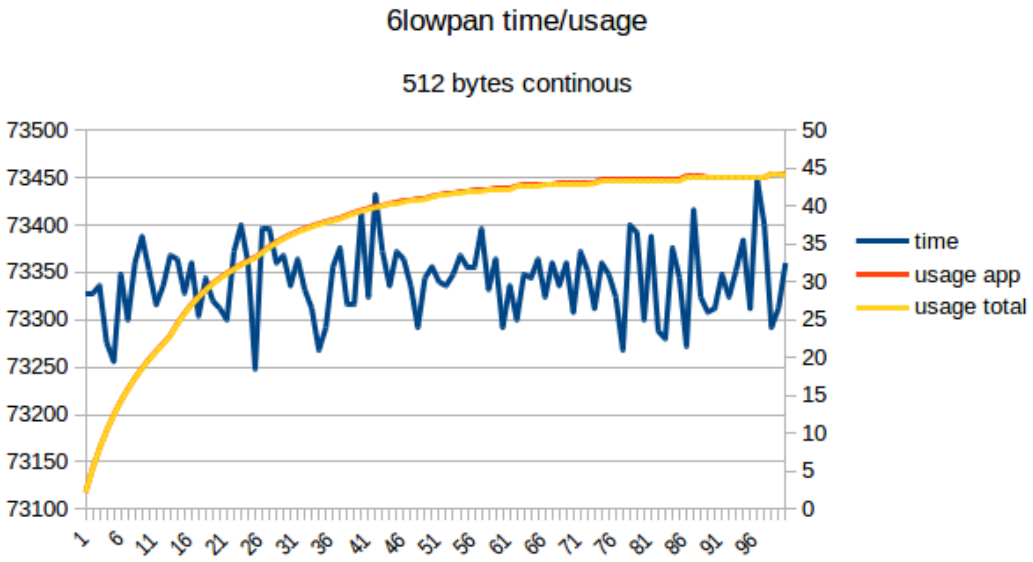

Figure 16: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send continuously over serial.

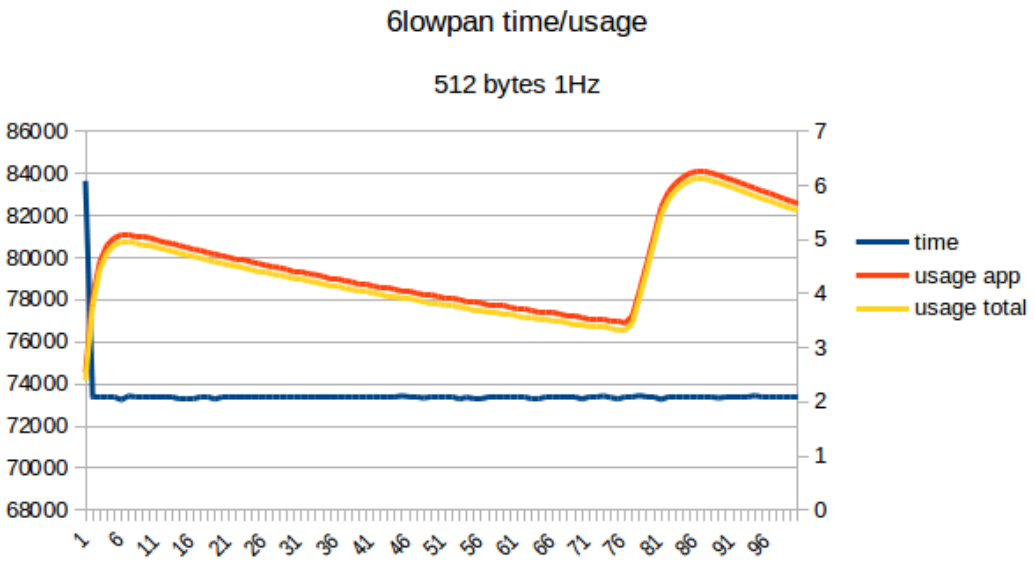

Figure 17: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 1Hz frequency over serial.

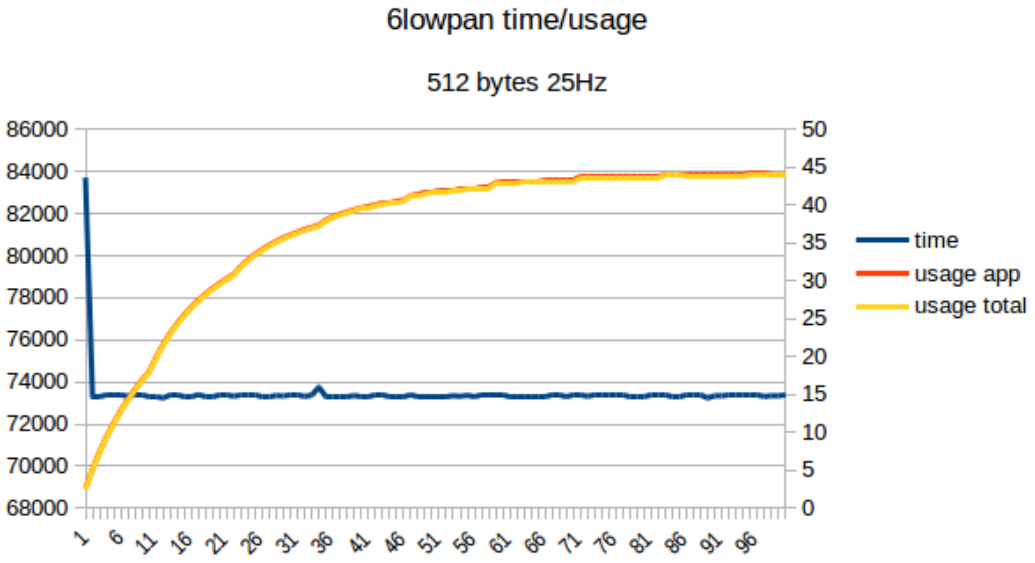

Figure 18: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 10Hz frequency over serial.

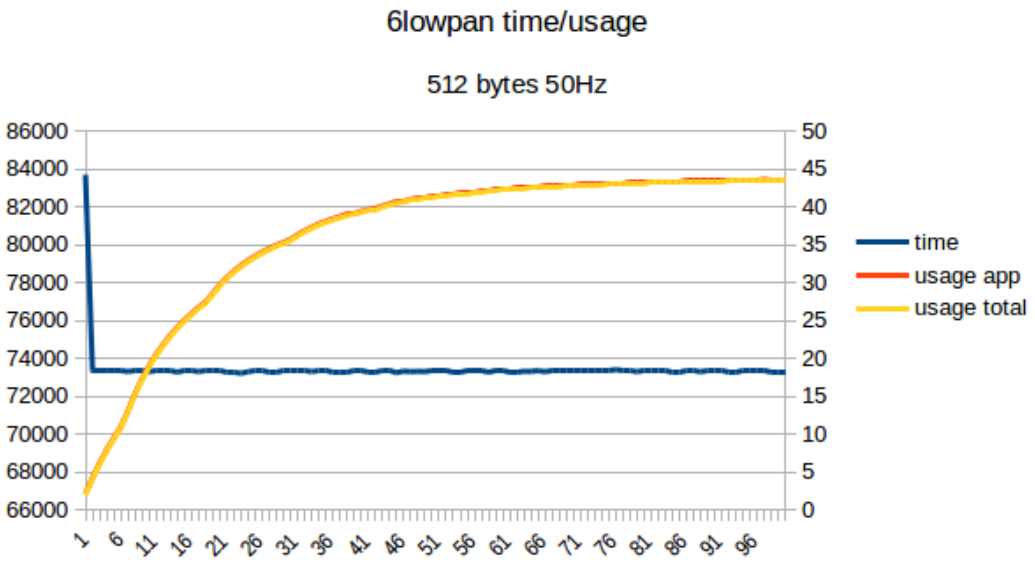

Figure 19: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 50Hz frequency over serial.

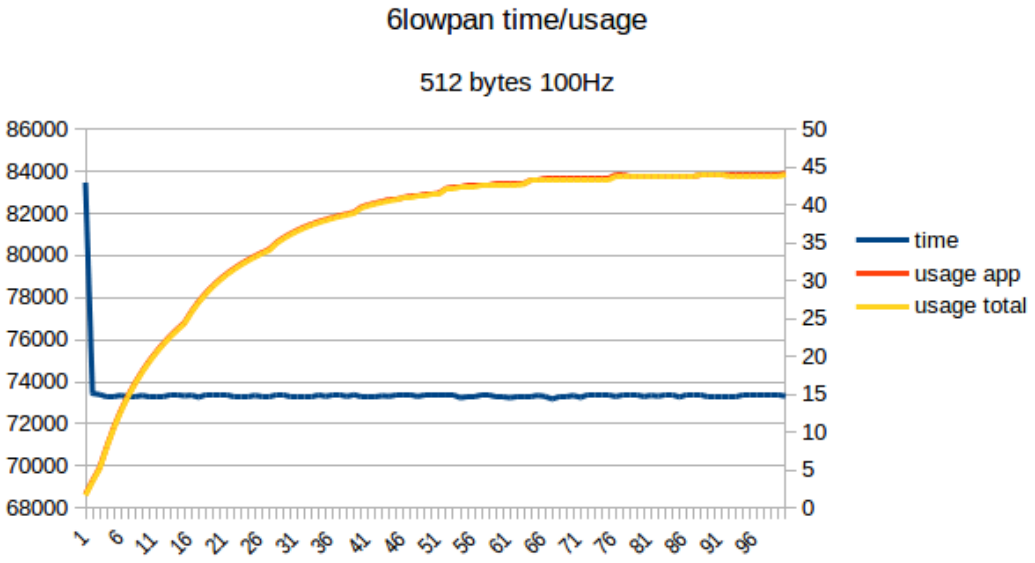

Figure 20: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 100Hz frequency over serial.

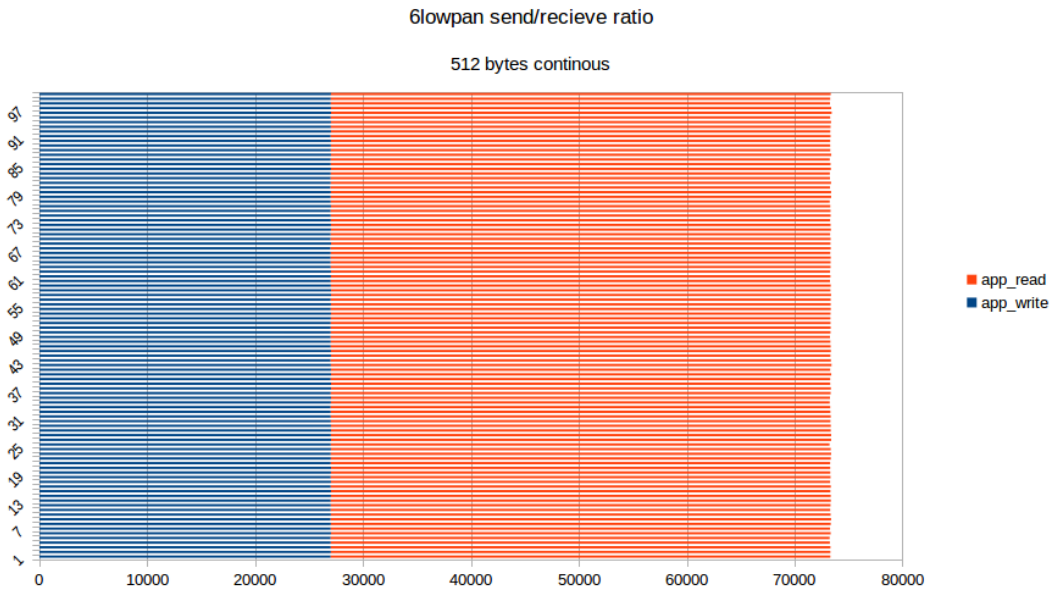

Figure 21: Ratio between time spend sending and receiving over serial. Note that receive time include time for other side to send data back.

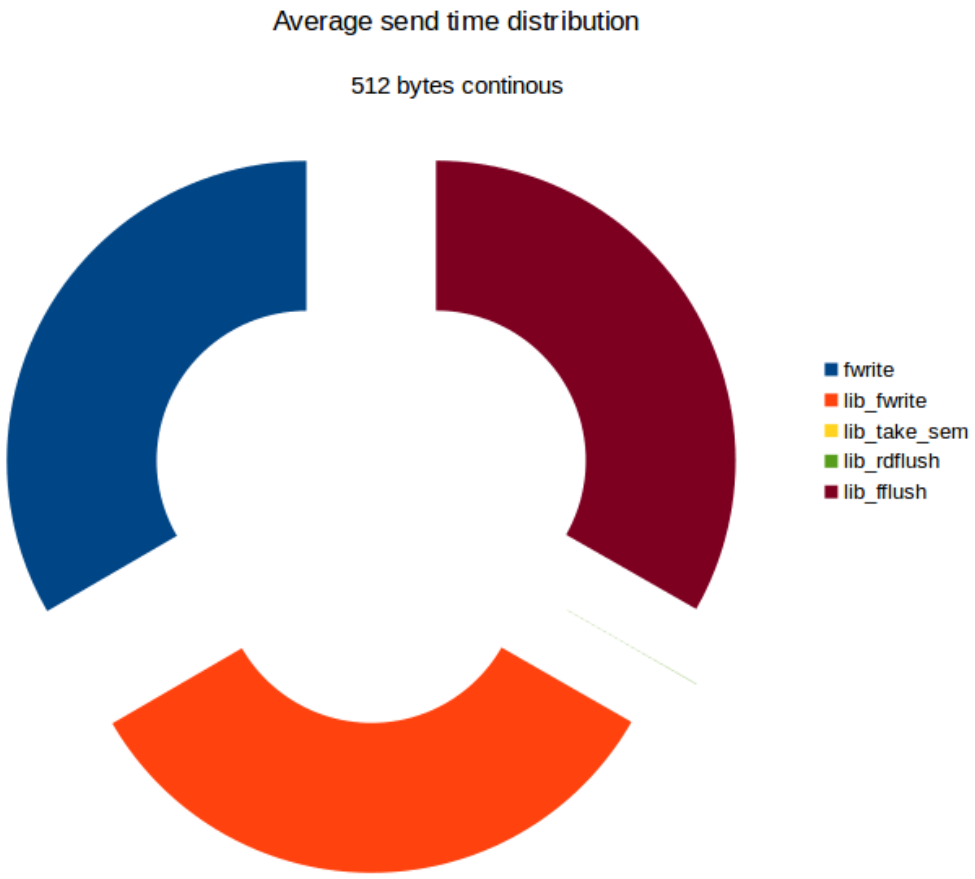

Figure 22: Comparison of time spend in various functions during send part of serial benchmark (note that this doesn't add to 100%).

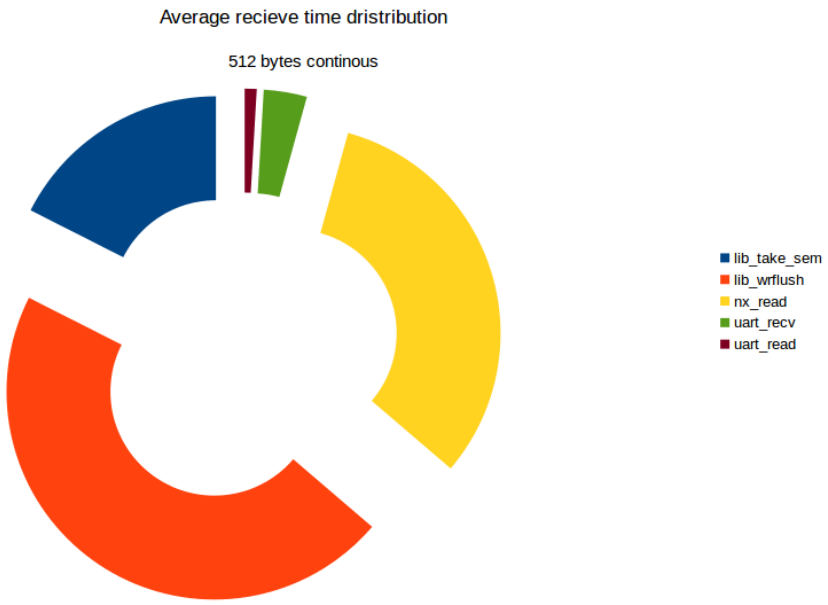

Figure 23: Comparison of time spend in various functions during receive part of serial benchmark (note that this doesn't add to 100%).

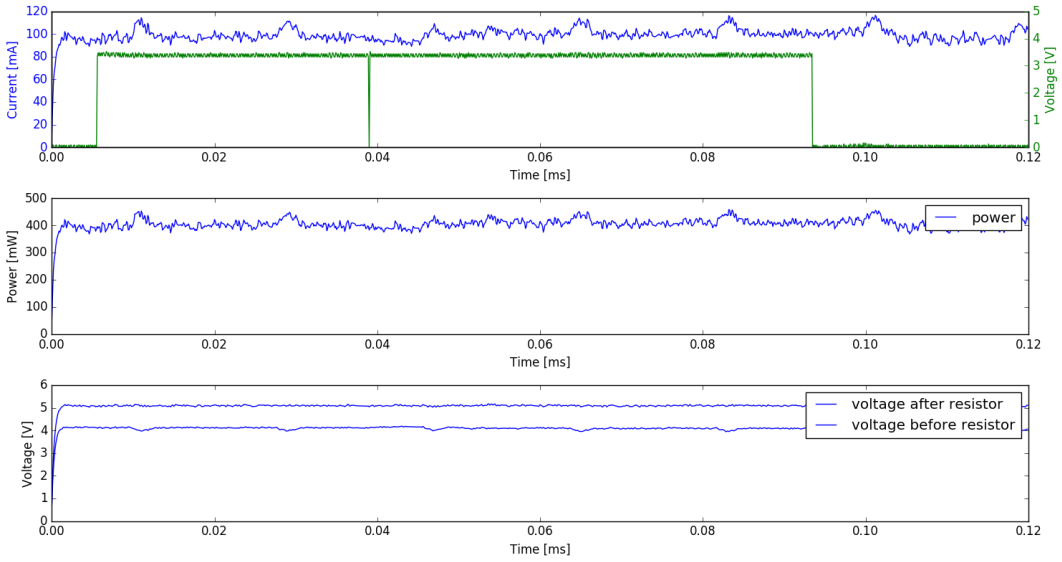

Figure 24: Current and power measurement during single send/receive cycle over serial.

7.2 TCP/IP

7.2.1 512 bytes

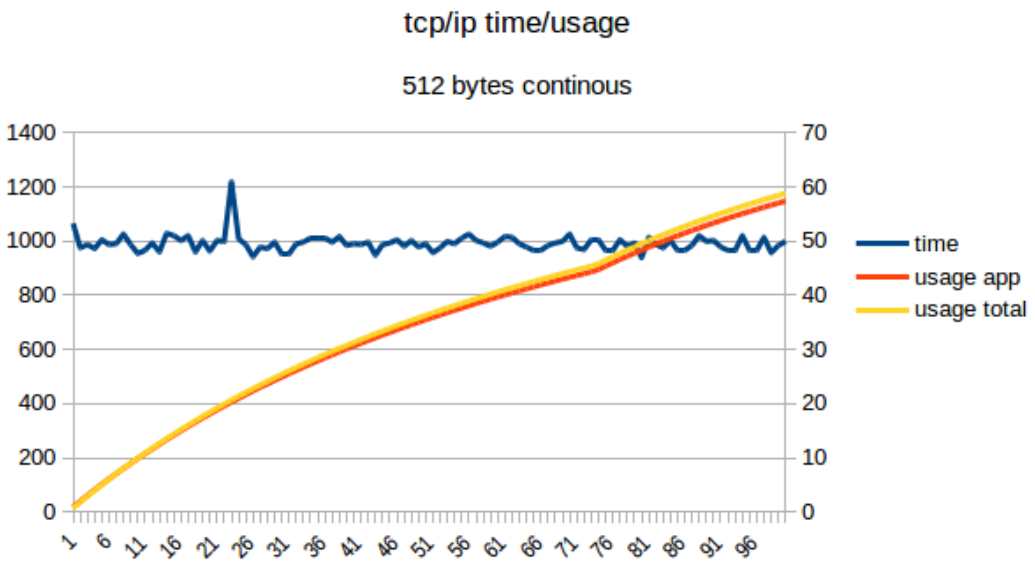

Figure 25: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send continuously.

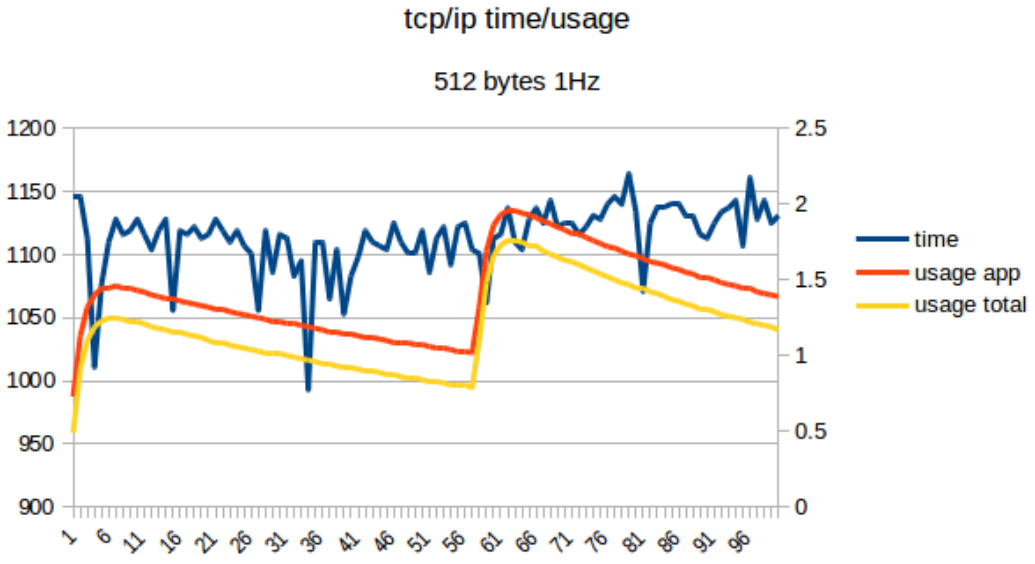

Figure 26: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 1Hz frequency.

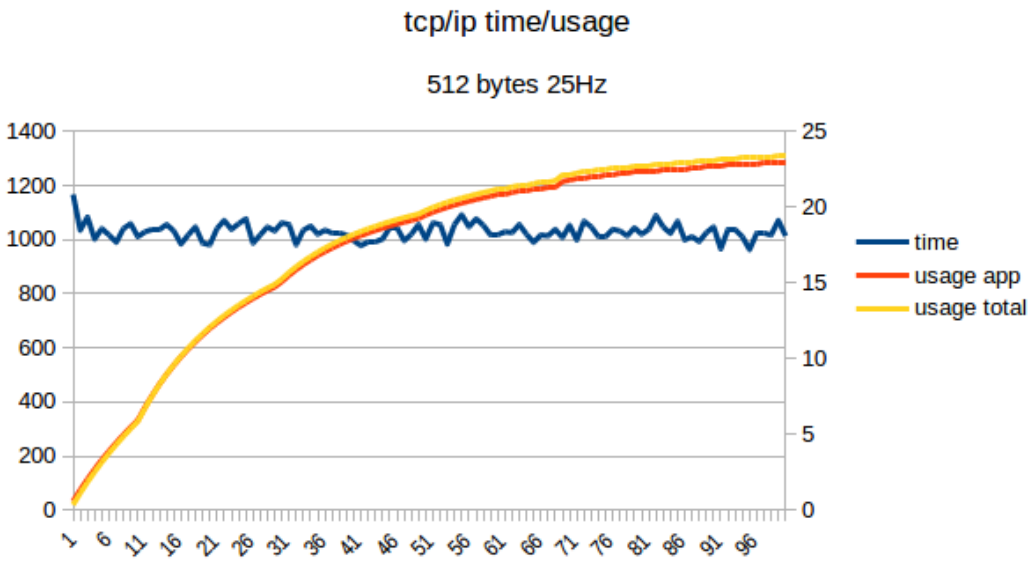

Figure 27: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 25Hz frequency.

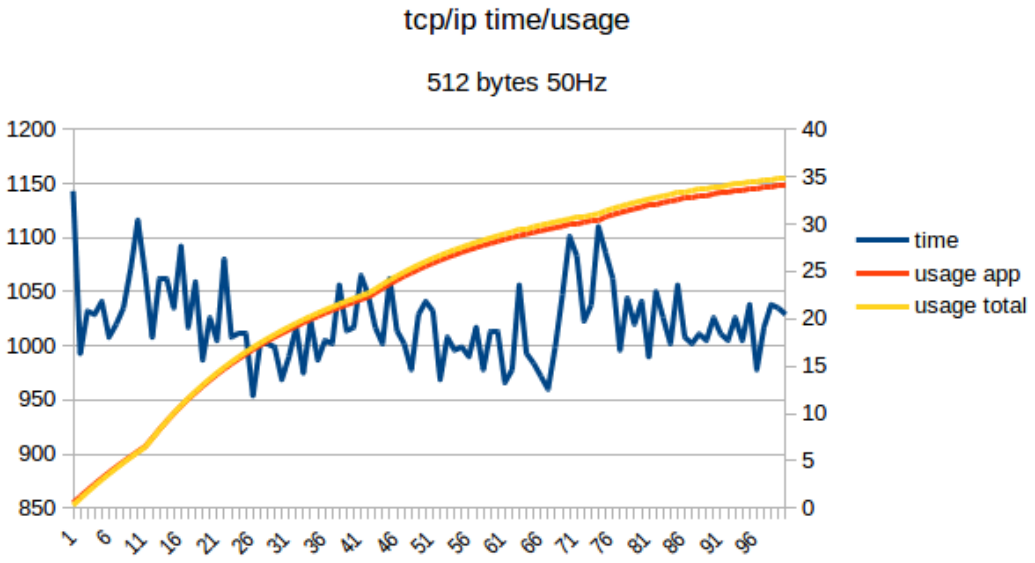

Figure 28: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 50Hz frequency.

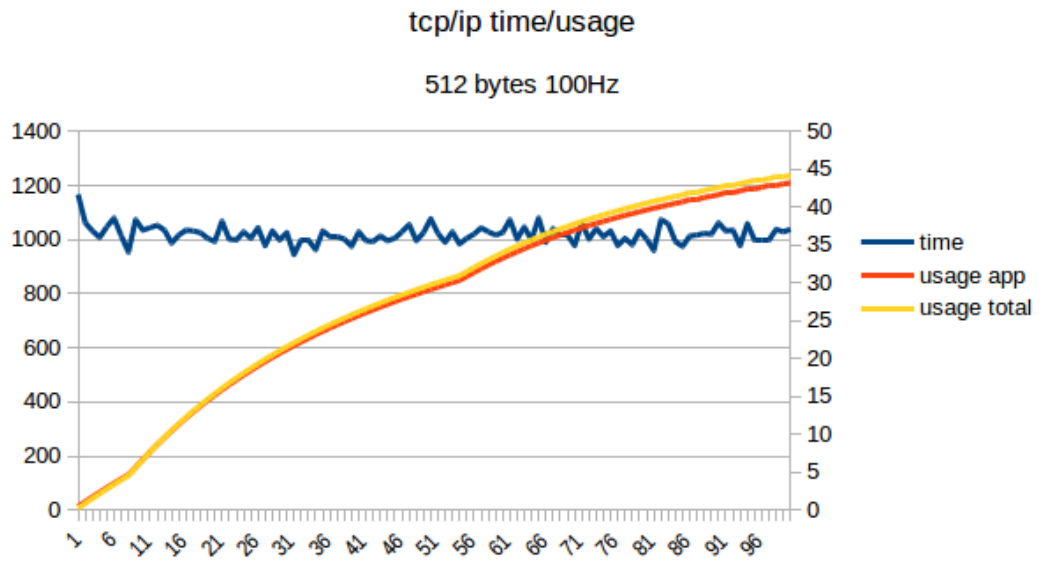

Figure 29: Time of send/receive cycle and cpu usage reported by OS, for 512 bytes packets send with 100Hz frequency.

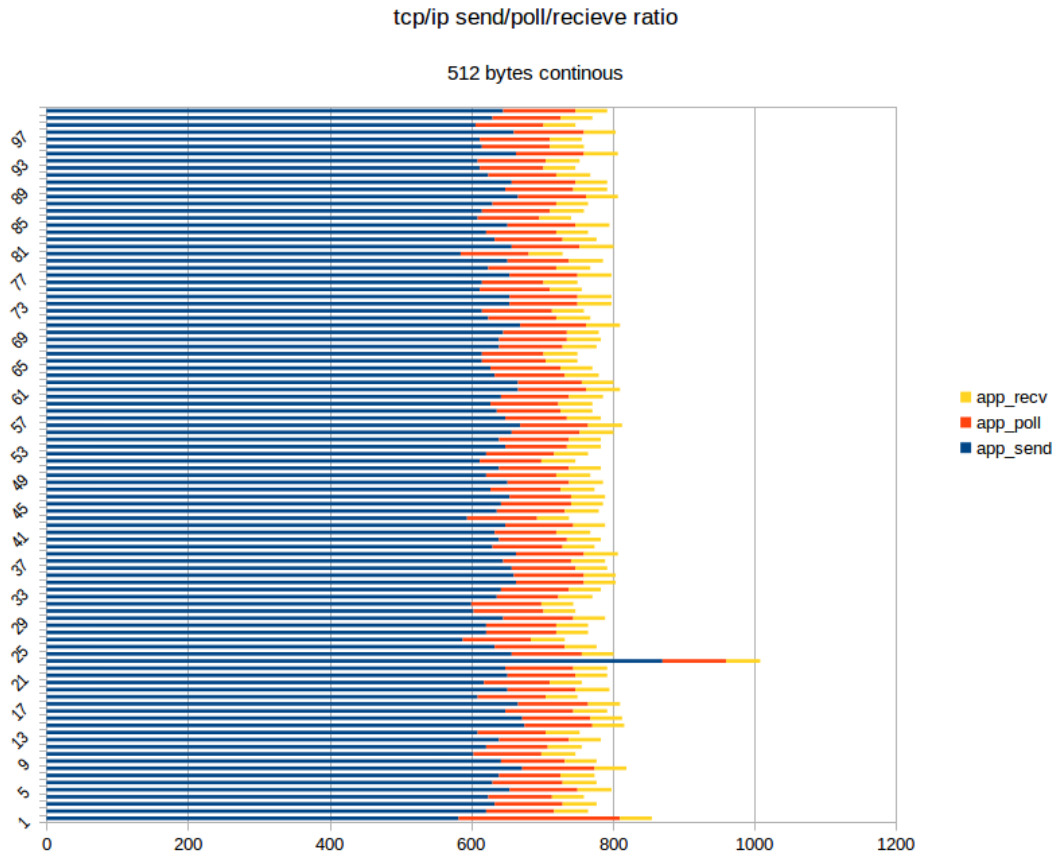

Figure 30: Ratio between time spend sending, receiving and polling.

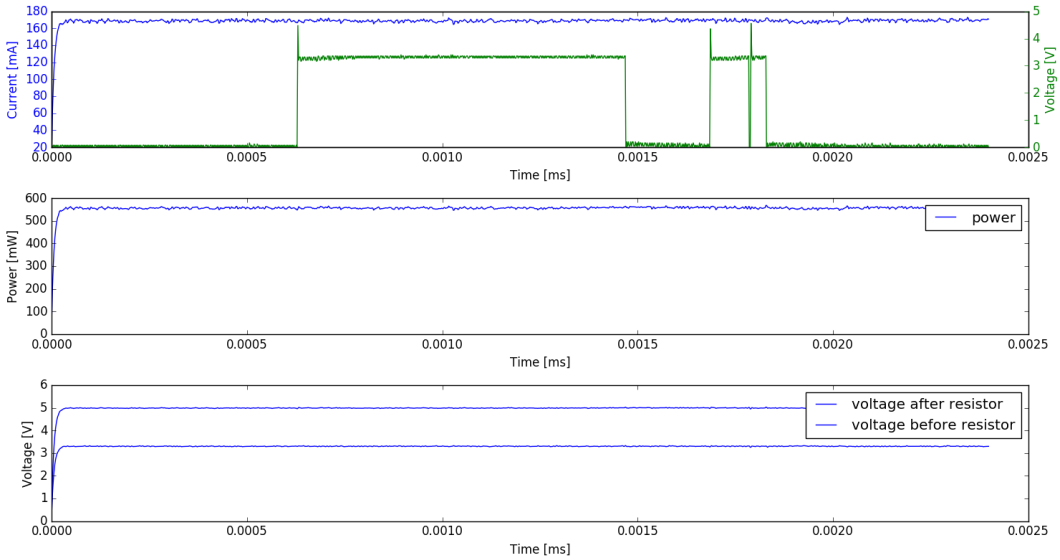

Figure 31: Current and power measurement during single send/receive cycle.

7.2.2 64 bytes

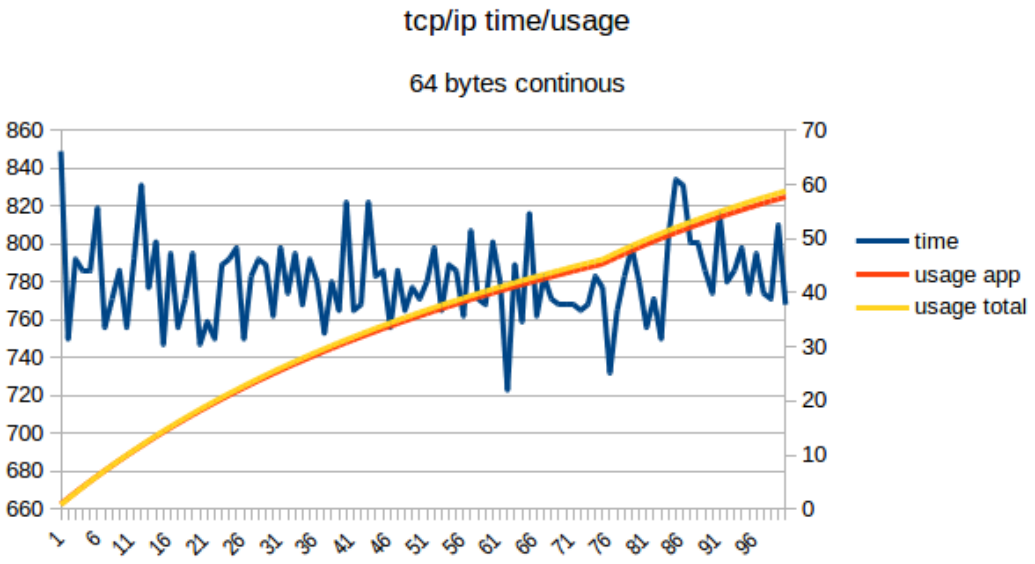

Figure 32: Time of send/receive cycle and cpu usage reported by OS, for 64 bytes packets send continuously.

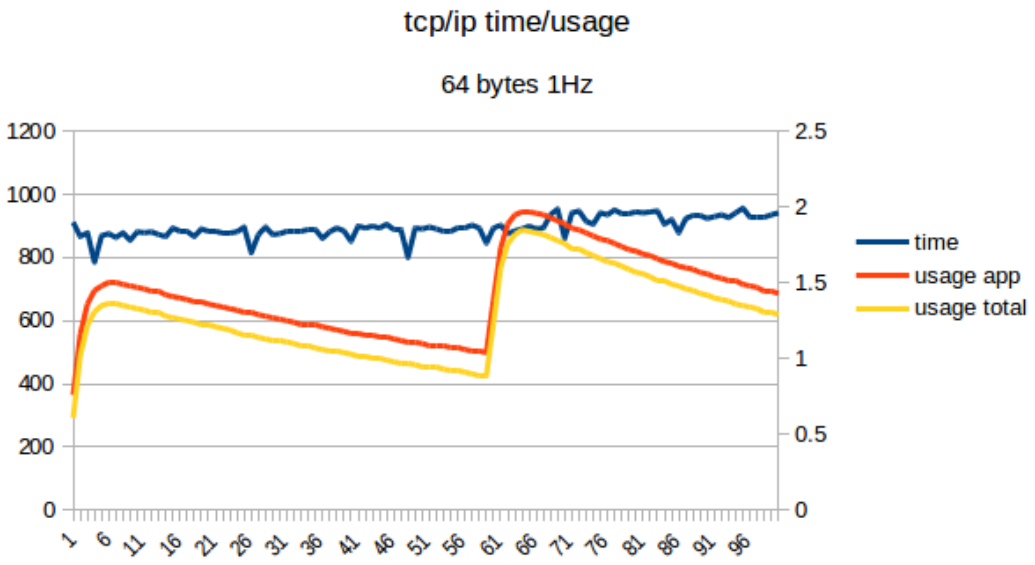

Figure 33: Time of send/receive cycle and cpu usage reported by OS, for 64 bytes packets send with 1Hz frequency.

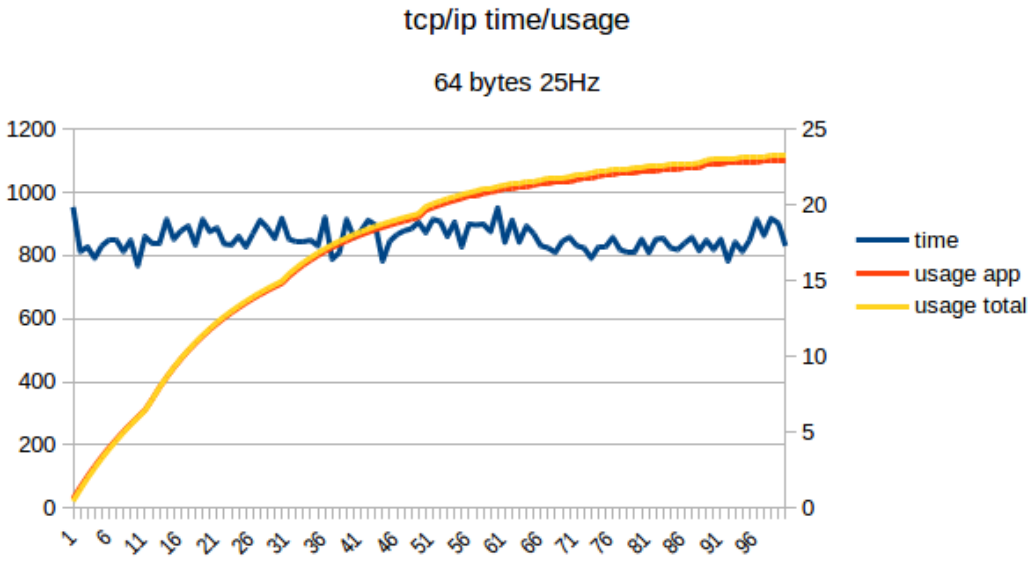

Figure 34: Time of send/receive cycle and cpu usage reported by OS, for 64 bytes packets send with 25Hz frequency.

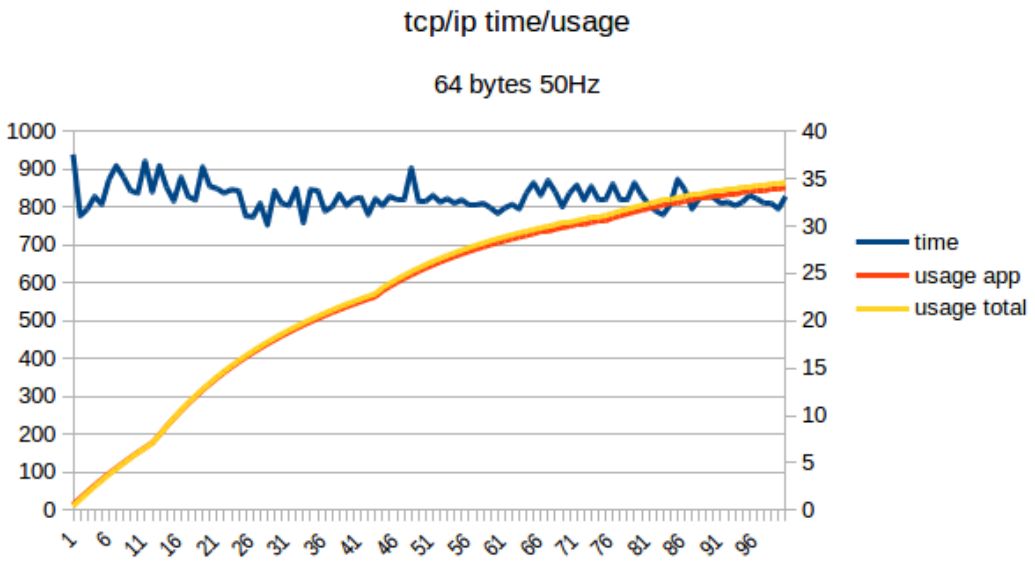

Figure 35: Time of send/receive cycle and cpu usage reported by OS, for 64 bytes packets send with 50Hz frequency.

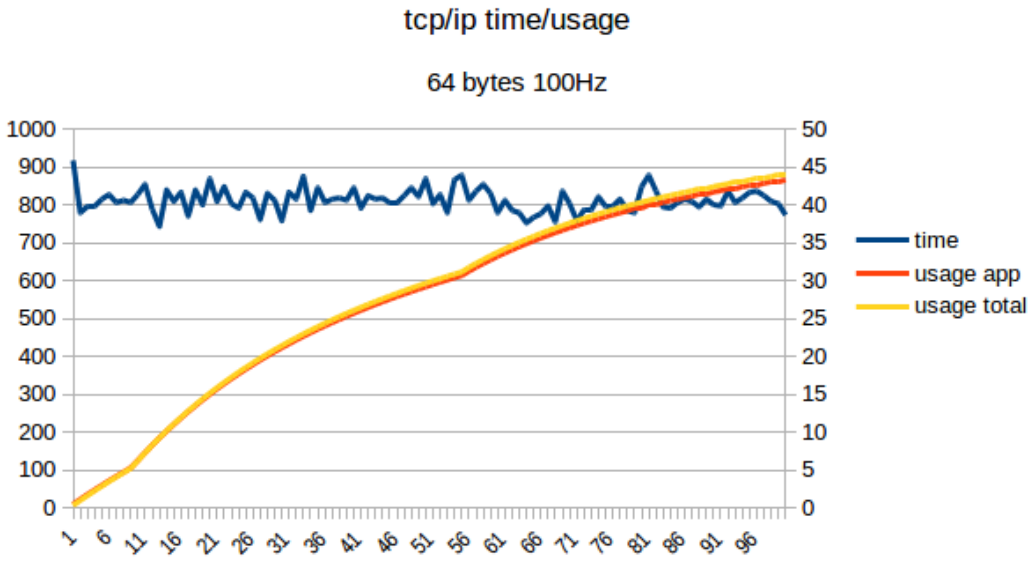

Figure 36: Time of send/receive cycle and cpu usage reported by OS, for 64 bytes packets send with 100Hz frequency.

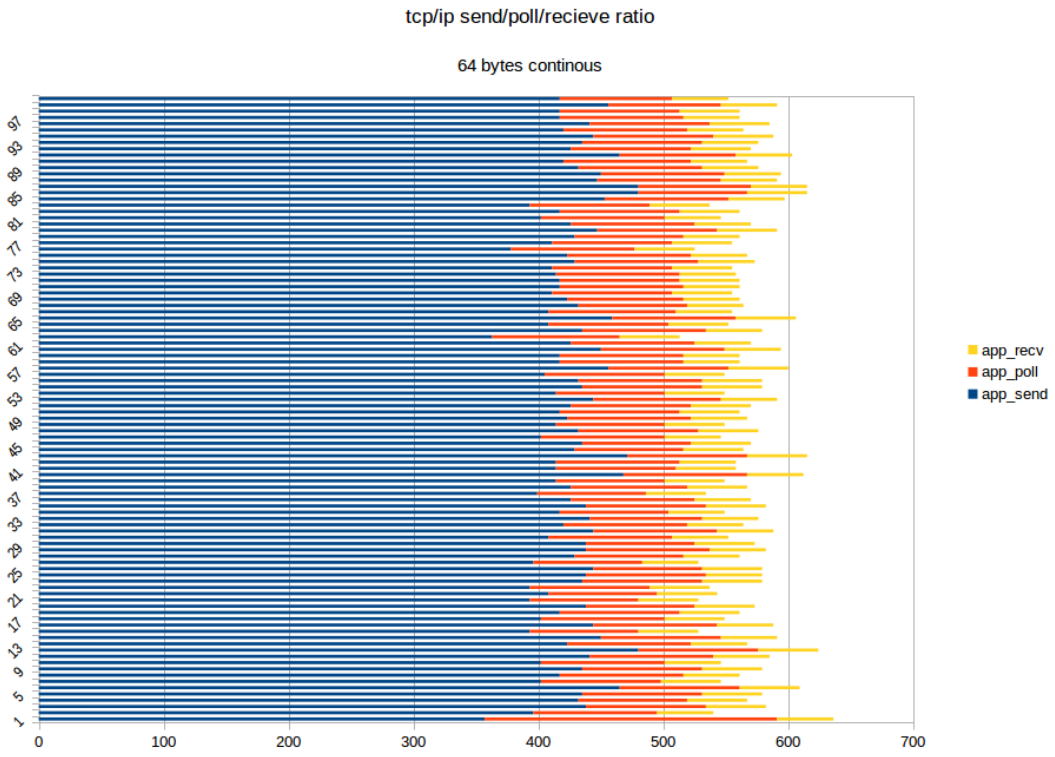

Figure 37: Ratio between time spend sending, receiving and polling.

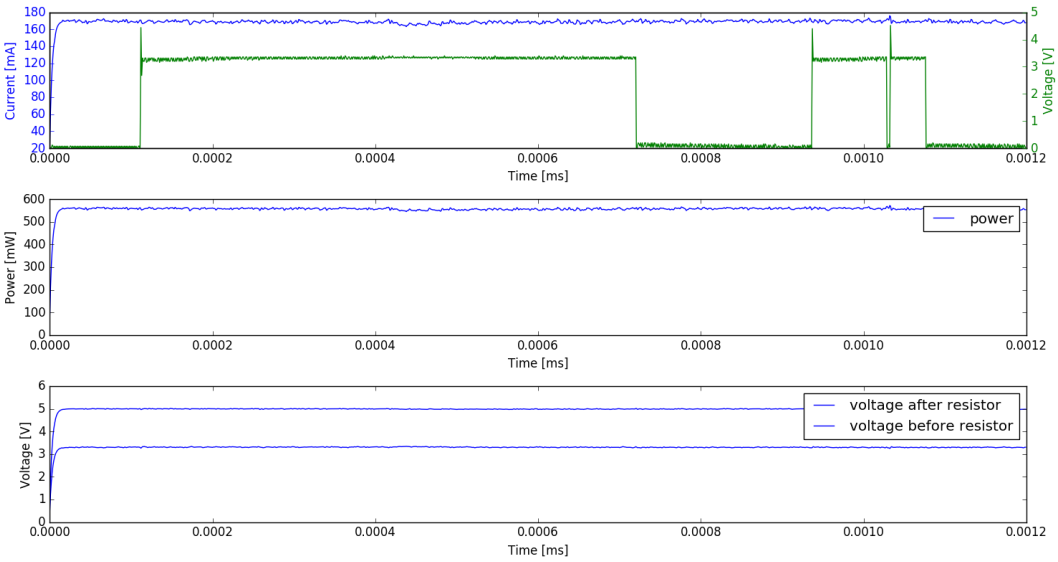

Figure 38: Current and power measurement during single send/receive cycle.

7.2.3 16 bytes

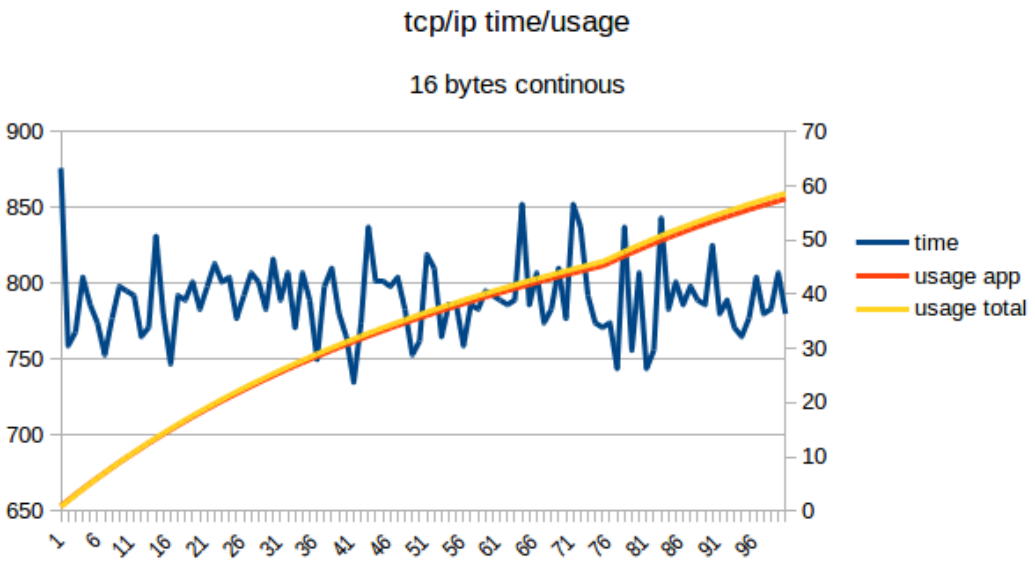

Figure 39: Time of send/receive cycle and cpu usage reported by OS, for 16 bytes packets send continuously.

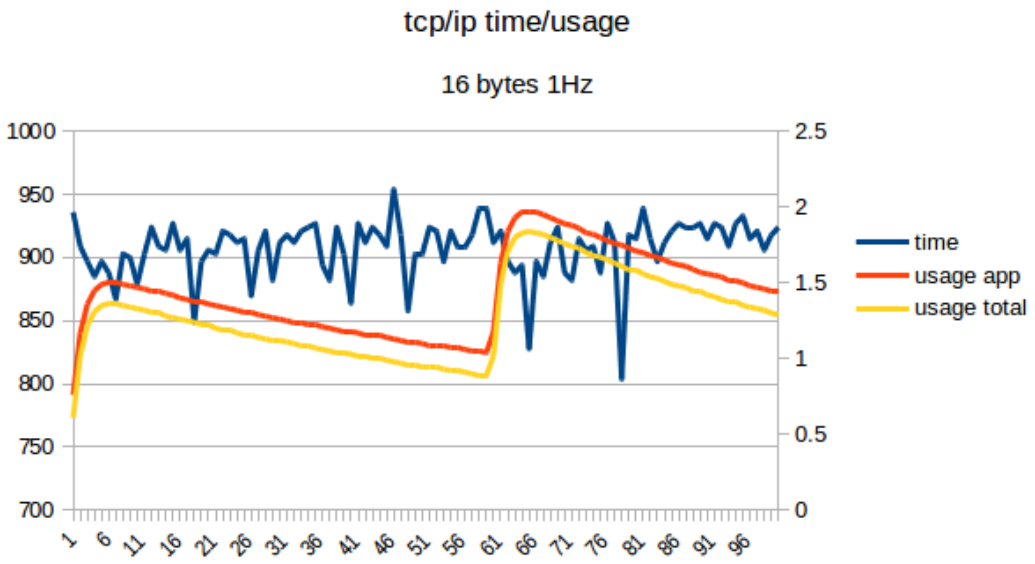

Figure 40: Time of send/receive cycle and cpu usage reported by OS, for 16 bytes packets send with 1Hz frequency.

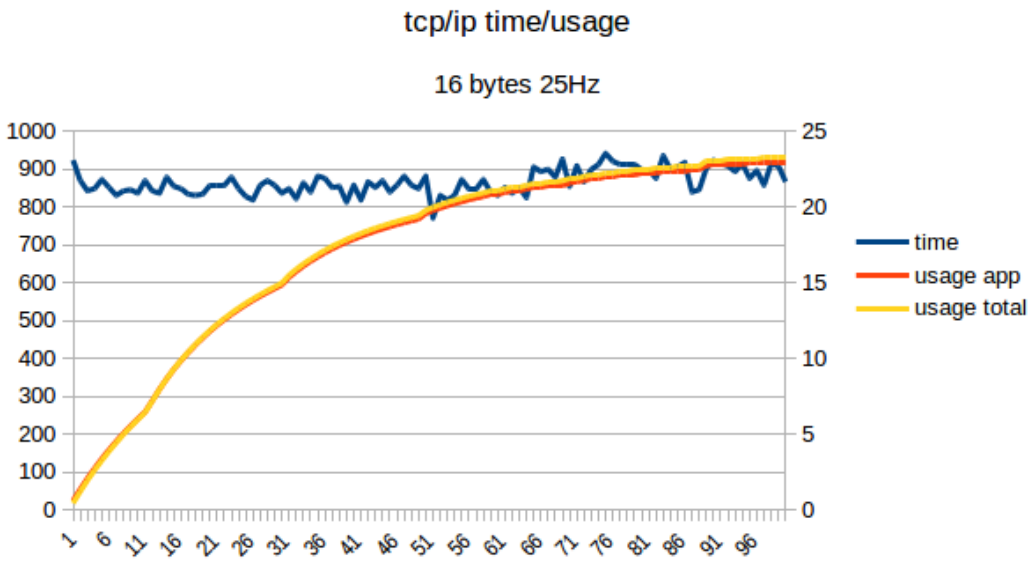

Figure 41: Time of send/receive cycle and cpu usage reported by OS, for 16 bytes packets send with 25Hz frequency.

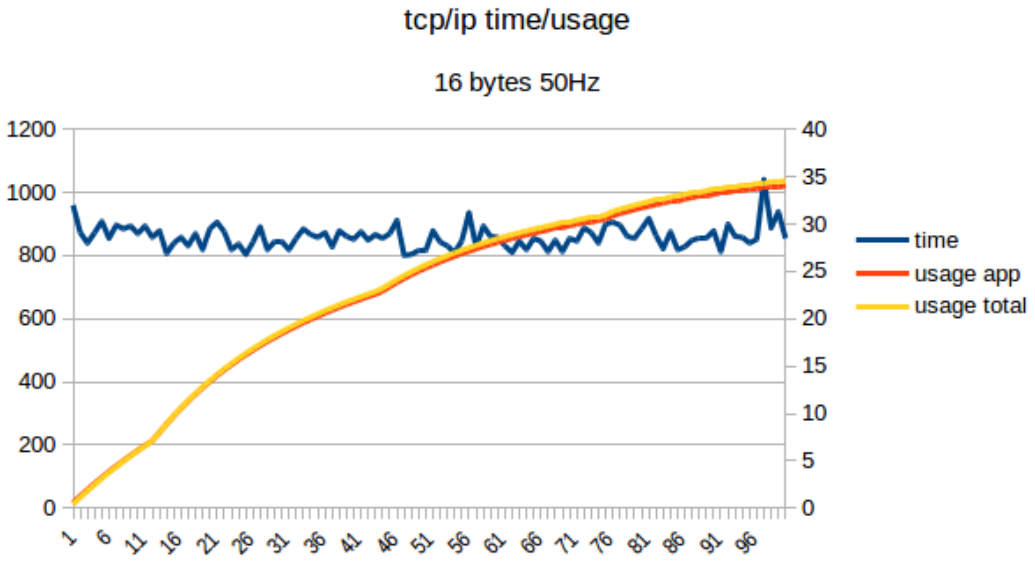

Figure 42: Time of send/receive cycle and cpu usage reported by OS, for 16 bytes packets send with 50Hz frequency.

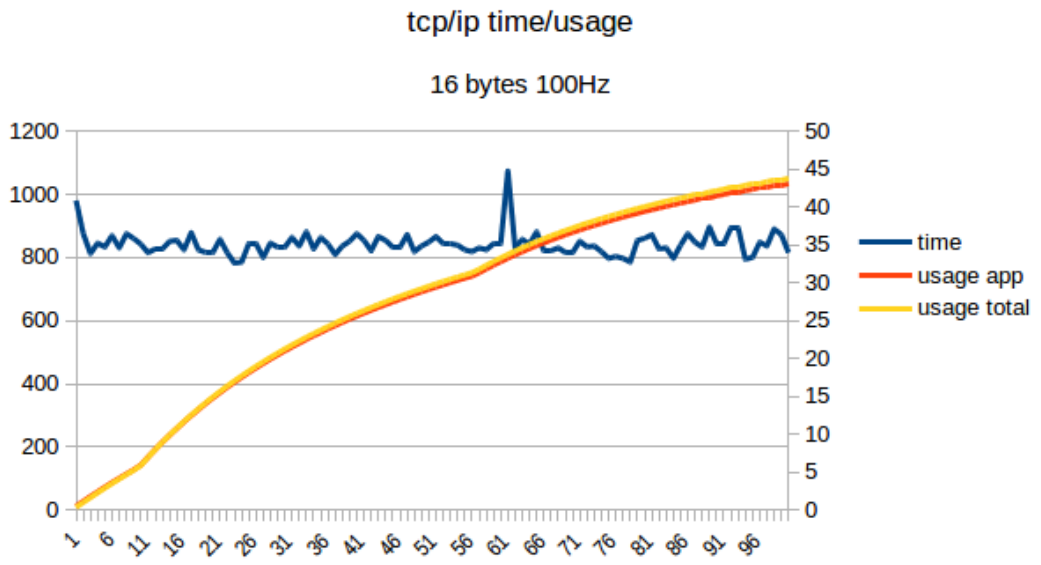

Figure 43: Time of send/receive cycle and cpu usage reported by OS, for 16 bytes packets send with 100Hz frequency.

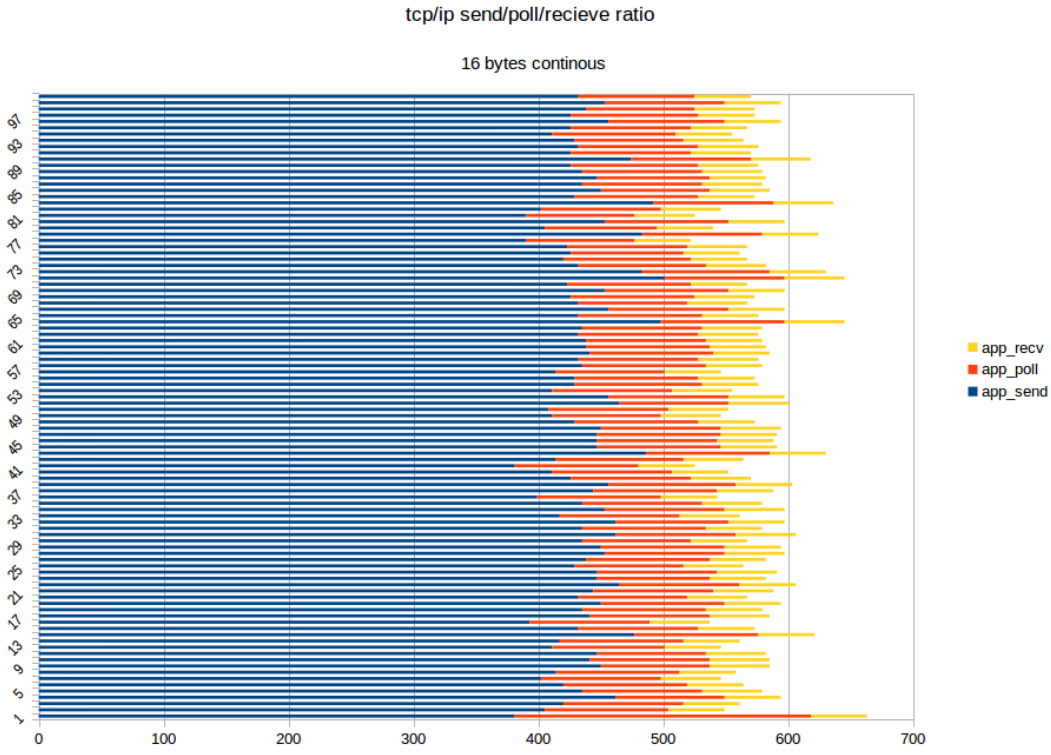

Figure 44: Ratio between time spend sending, receiving and polling.

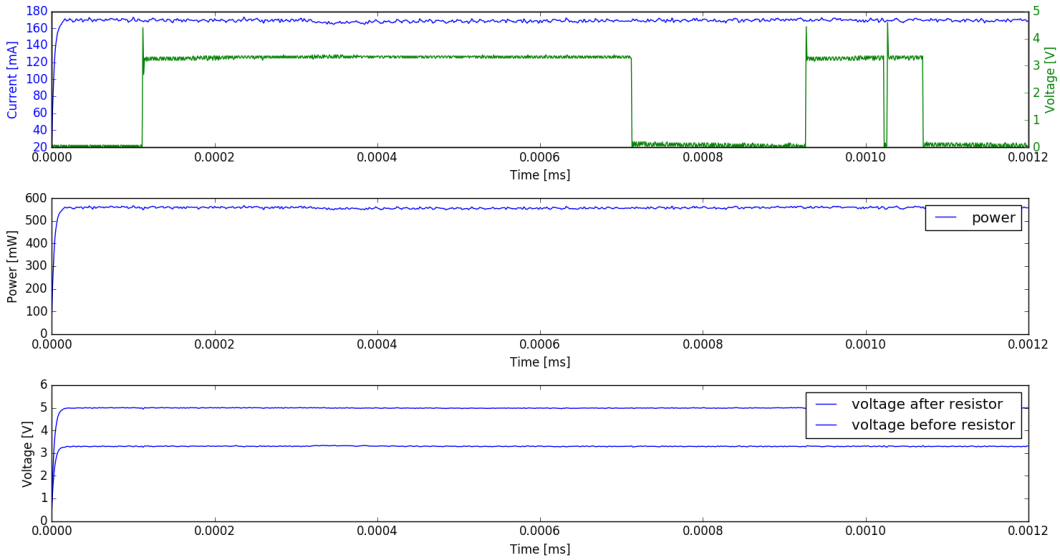

Figure 45: Current and power measurement during single send/receive cycle.

7.2.4 Comparison

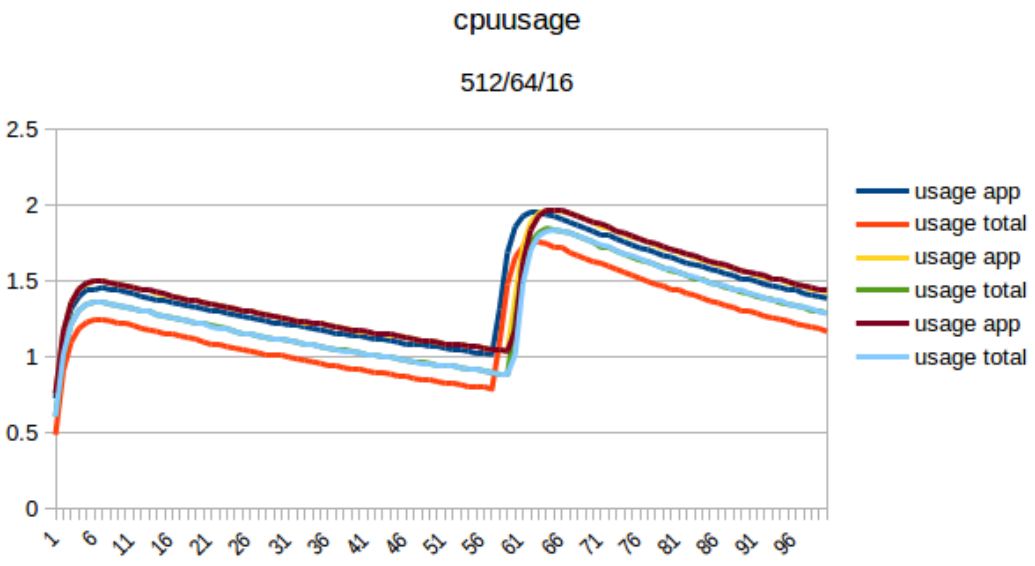

Figure 46: Comparison of send/receive cycle time for 512/64/16 bytes packets send over network with 1Hz frequency.

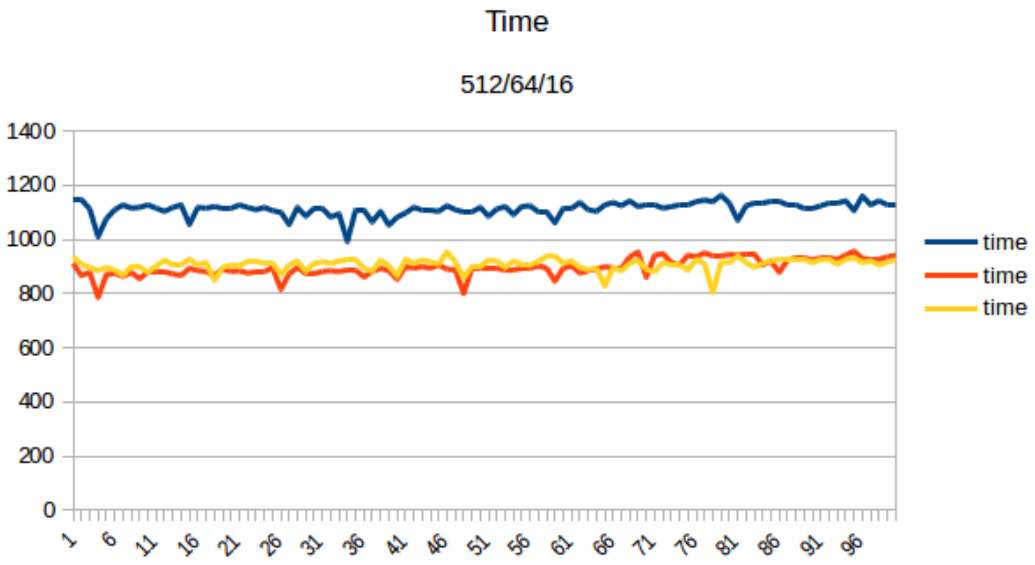

Figure 47: Comparison of cpu usage reported by OS for 512/64/16 bytes packets send over network with 1Hz frequency.

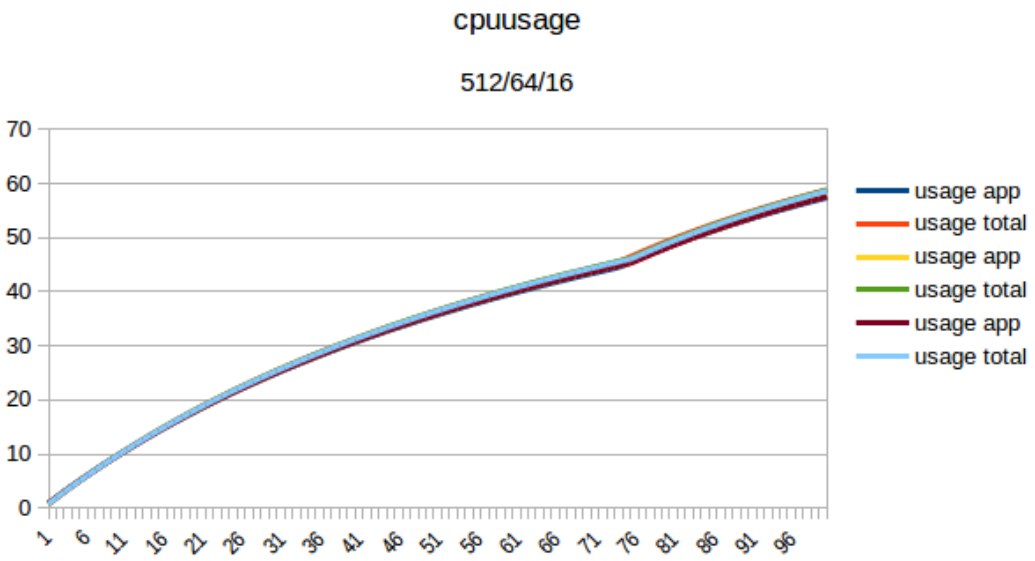

Figure 48: Comparison of send/receive cycle time for 512/64/16 bytes packets send over network continuously.

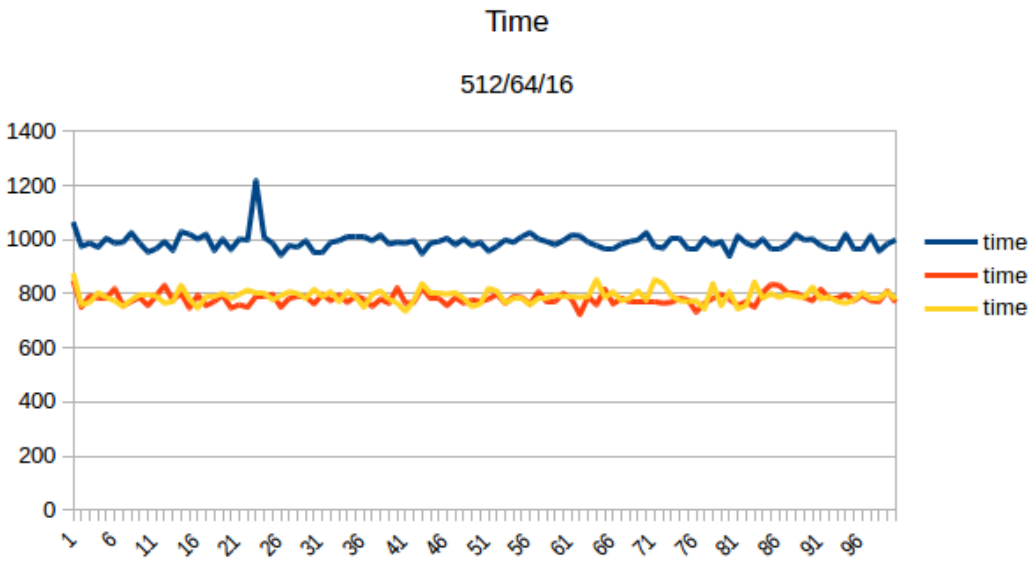

Figure 49: Comparison of cpu usage reported by OS for 512/64/16 bytes packets send over network continuously.

7.3 6LoWPAN radio

7.3.1 512 bytes

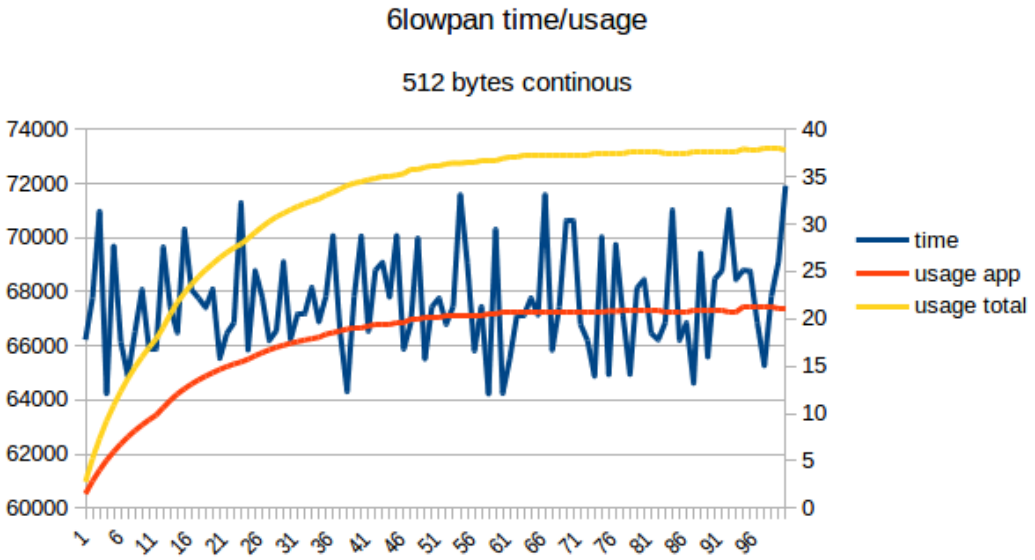

Figure 50: Time of send/receive cycle and cpu usage reported by OS.

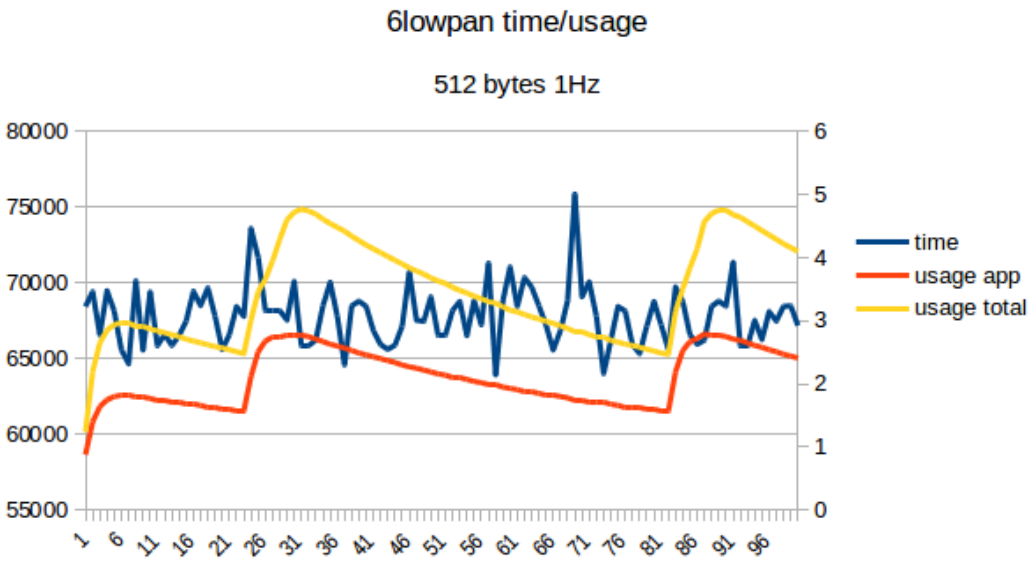

Figure 51: Time of send/receive cycle and cpu usage reported by OS.

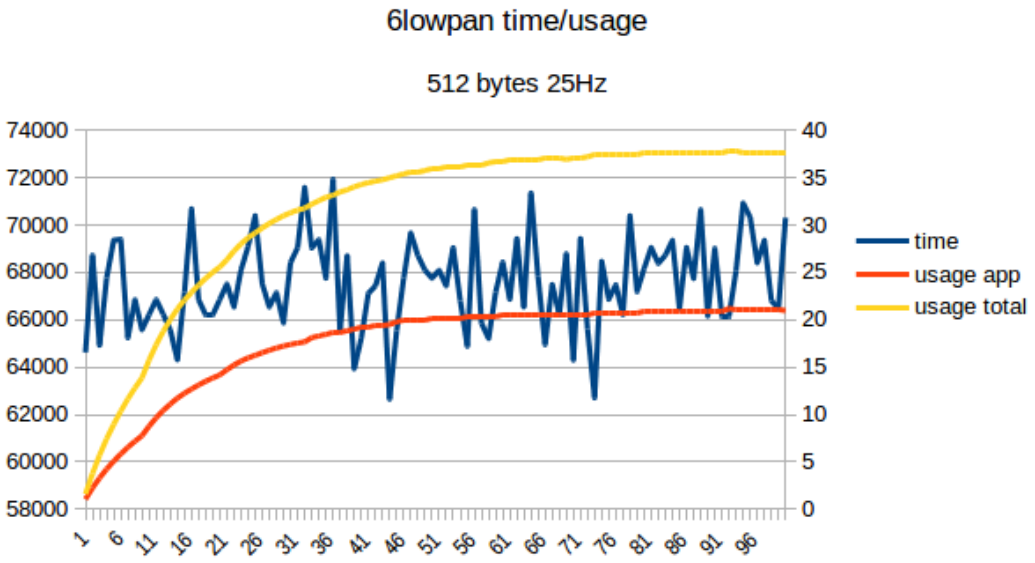

Figure 52: Time of send/receive cycle and cpu usage reported by OS.

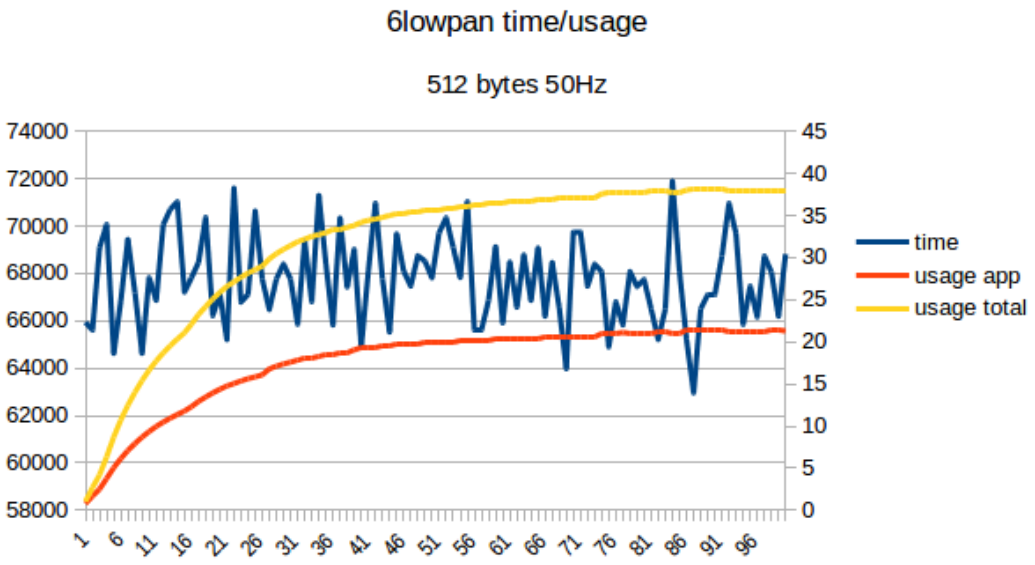

Figure 53: Time of send/receive cycle and cpu usage reported by OS.

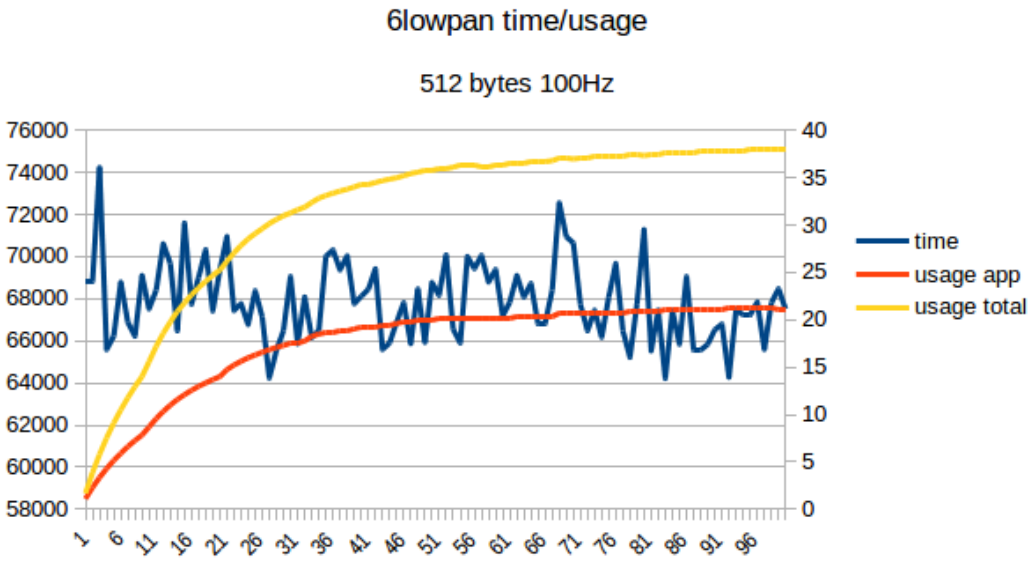

Figure 54: Time of send/receive cycle and cpu usage reported by OS.

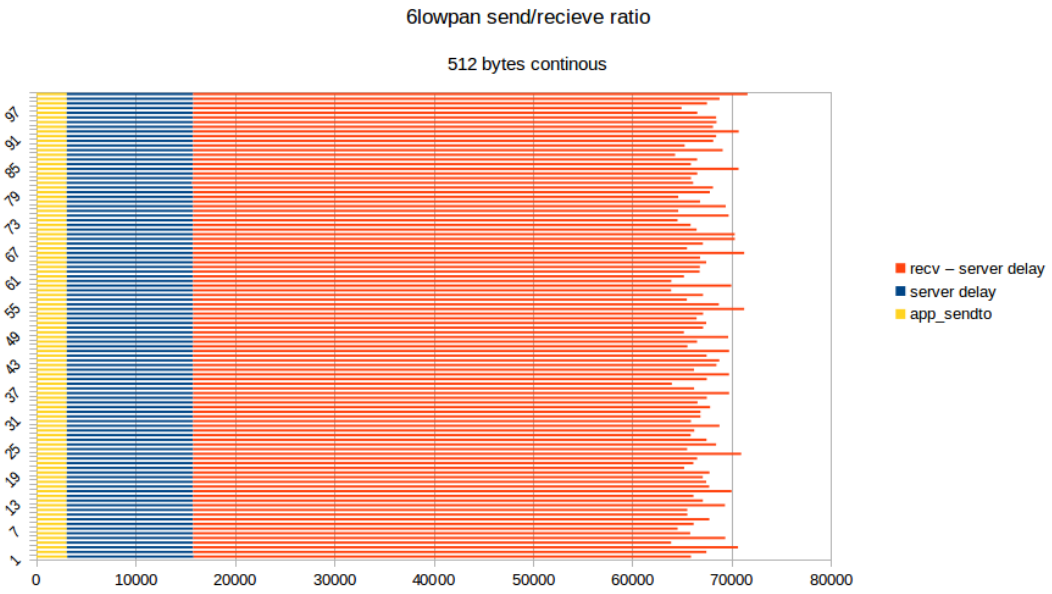

Figure 55: Ratio between time spend sending and receiving. Note that receive time is divided into server delay and actual receive (recv-server delay).

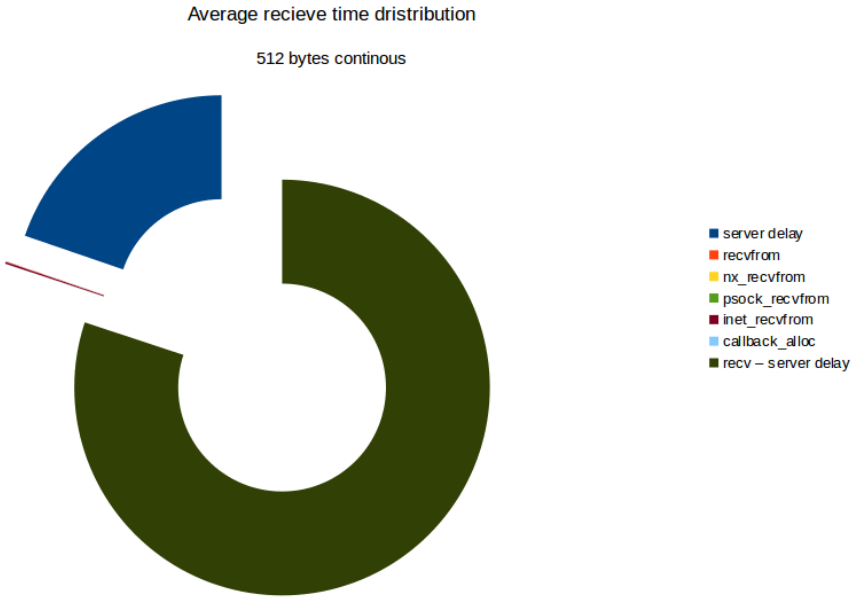

Figure 56: Comparison of time spend in various functions during send part of benchmark (note that this doesn't add to 100%).

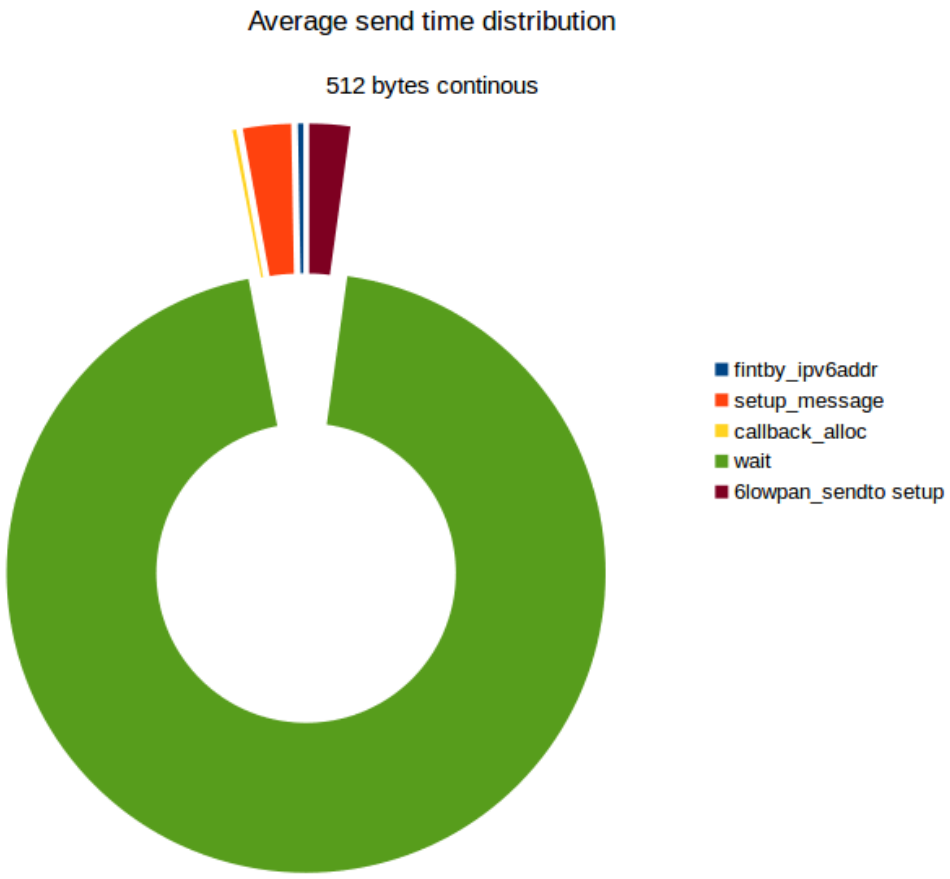

Figure 57: Comparison of time spend in various functions during receive part of benchmark (note that this doesn't add to 100%).

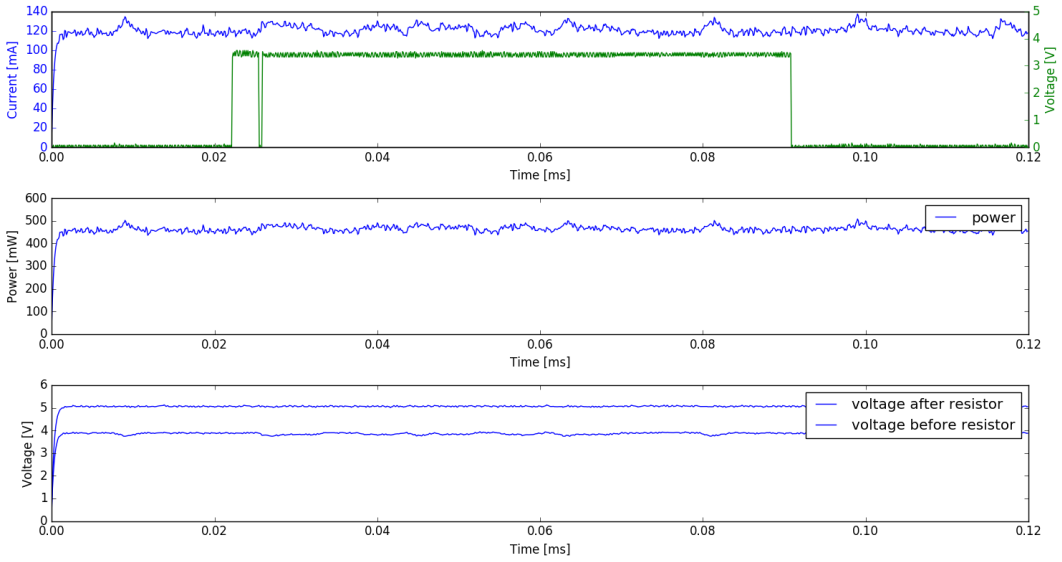

Figure 58: Current and power measurement during single send/receive cycle.

7.3.2 64 bytes

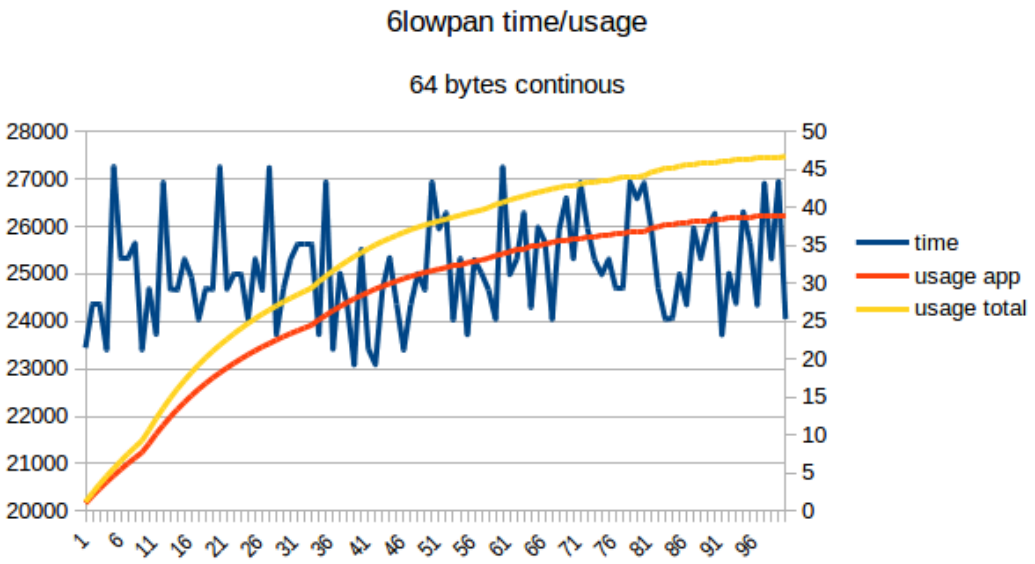

Figure 59: Time of send/receive cycle and cpu usage reported by OS.

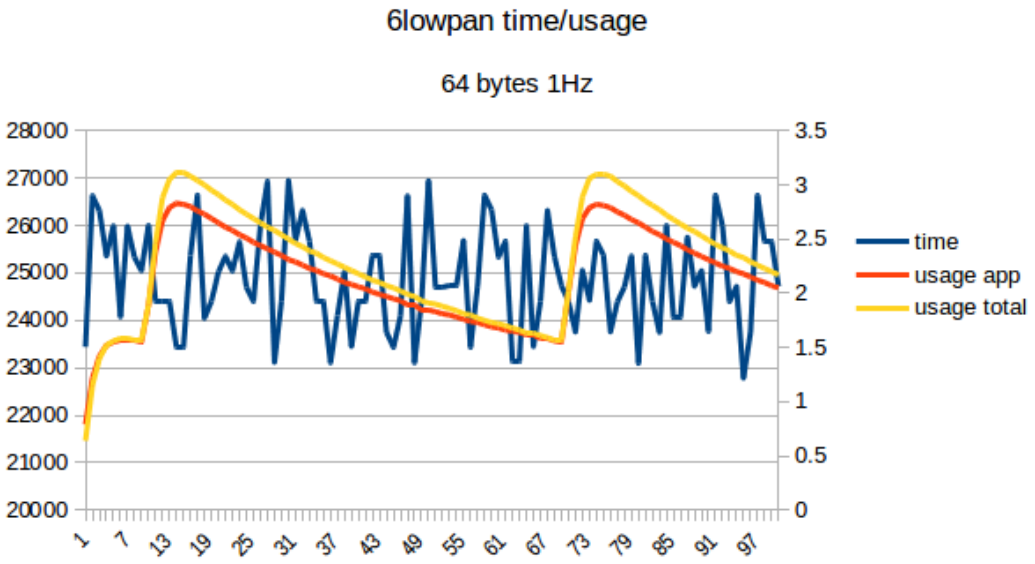

Figure 60: Time of send/receive cycle and cpu usage reported by OS.

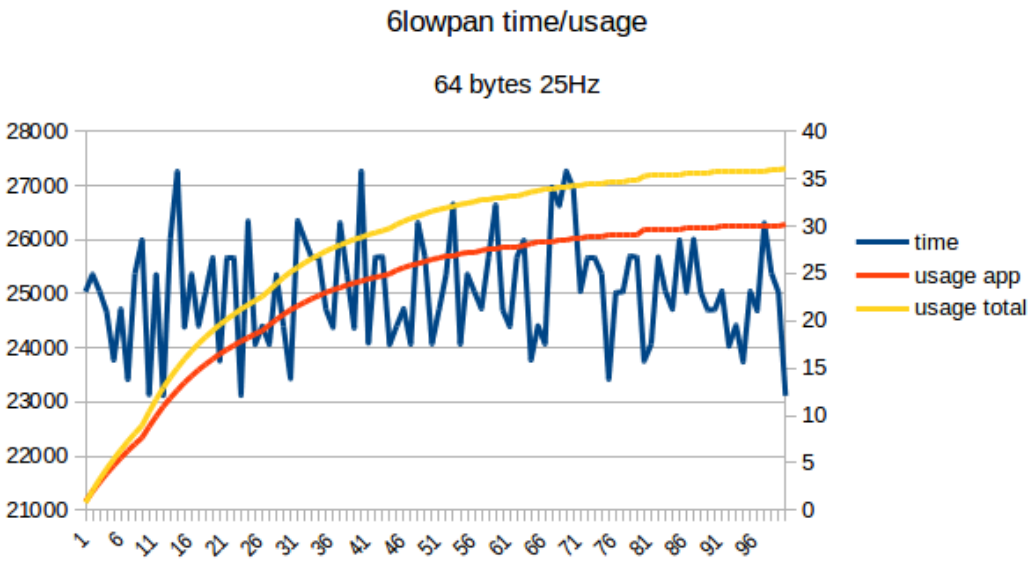

Figure 61: Time of send/receive cycle and cpu usage reported by OS.

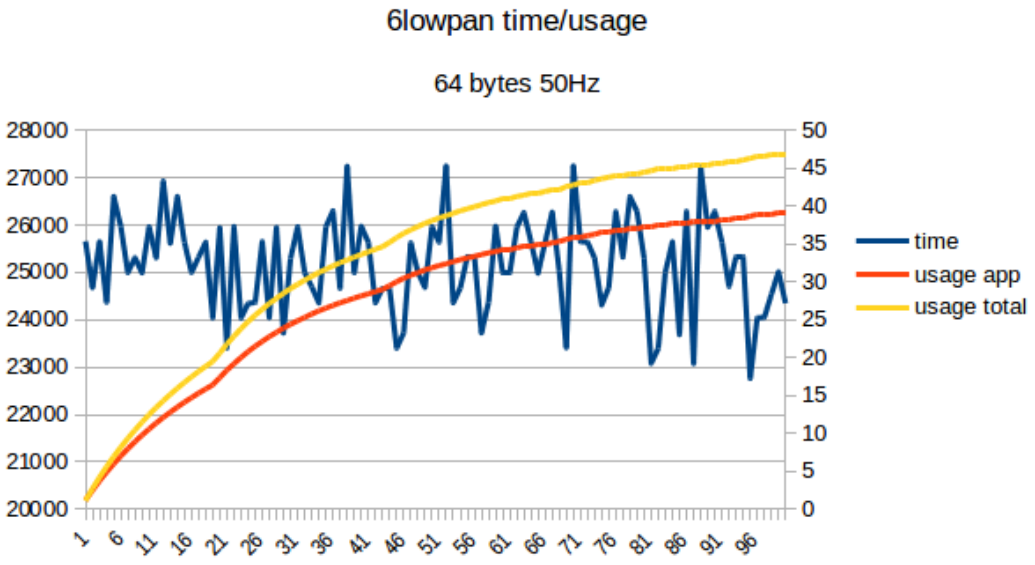

Figure 62: Time of send/receive cycle and cpu usage reported by OS.

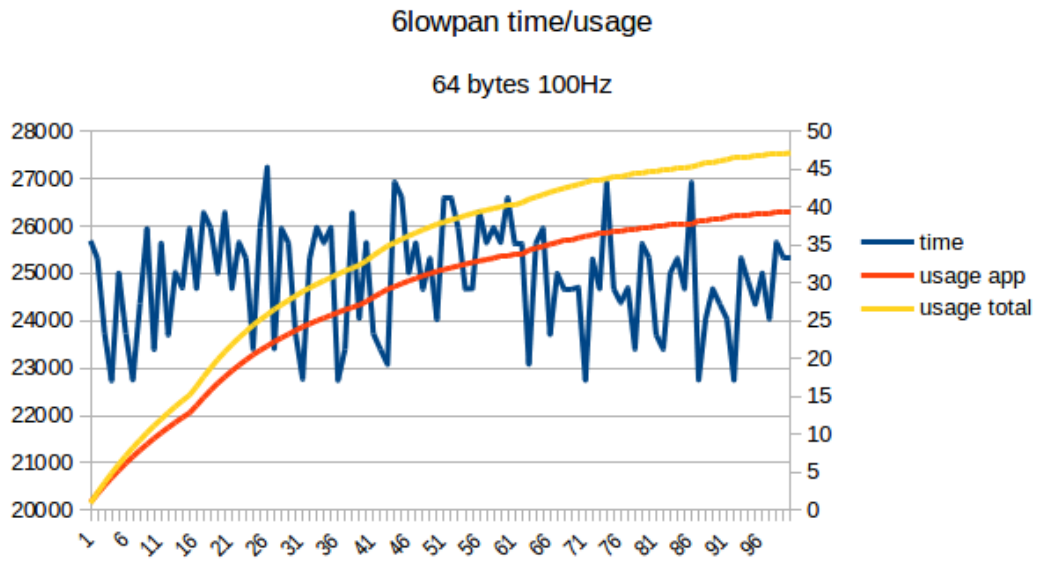

Figure 63: Time of send/receive cycle and cpu usage reported by OS.

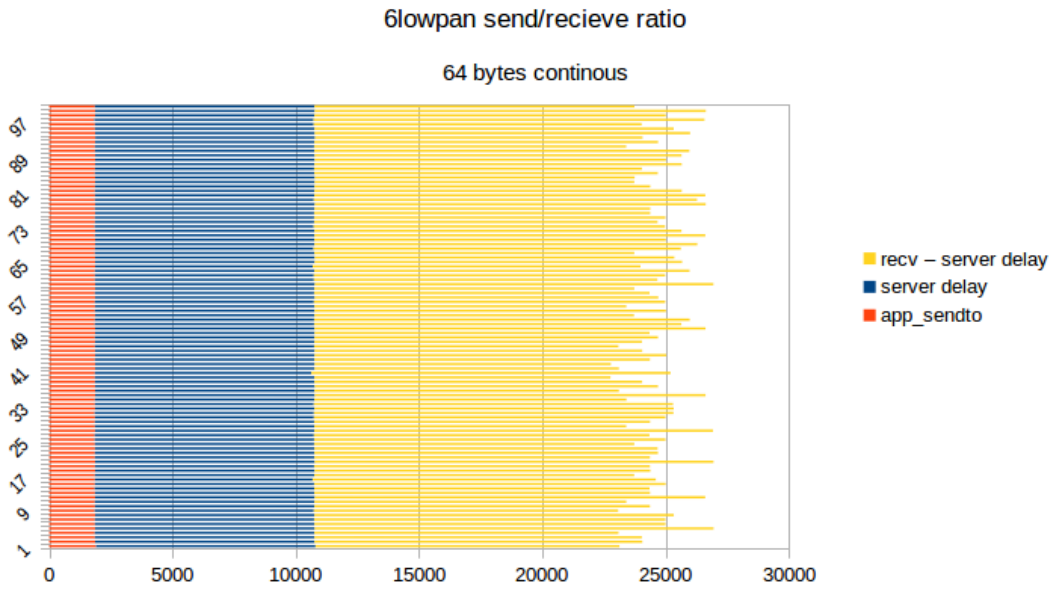

Figure 64: Ratio between time spend sending and receiving. Note that receive time is divided into server delay and actual receive (recv-server delay).

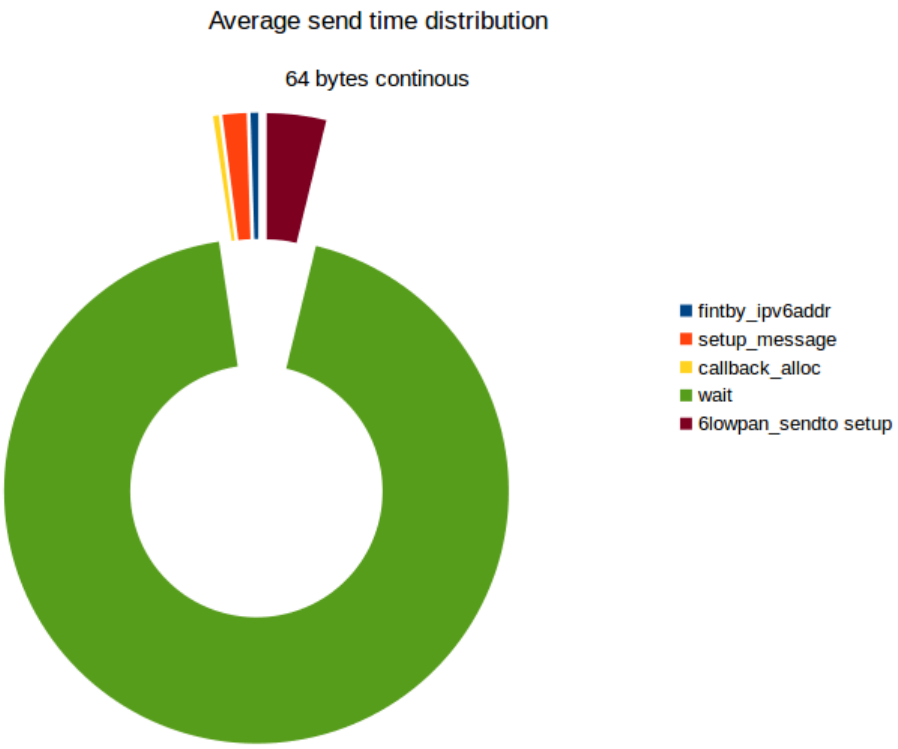

Figure 65: Comparison of time spend in various functions during send part of benchmark (note that this doesn't add to 100%).

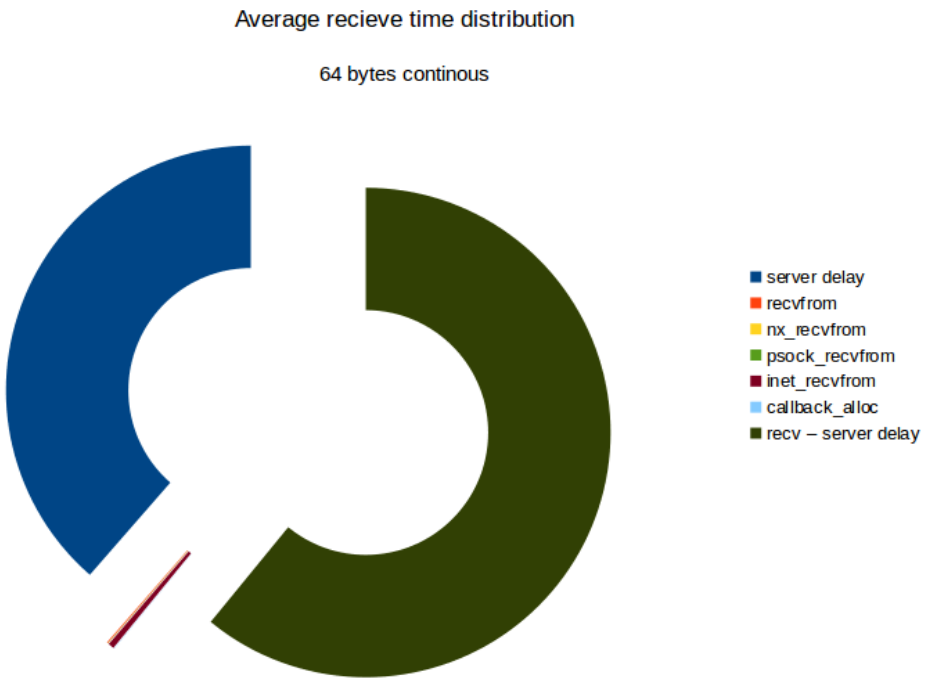

Figure 66: Comparison of time spend in various functions during receive part of benchmark (note that this doesn't add to 100%).

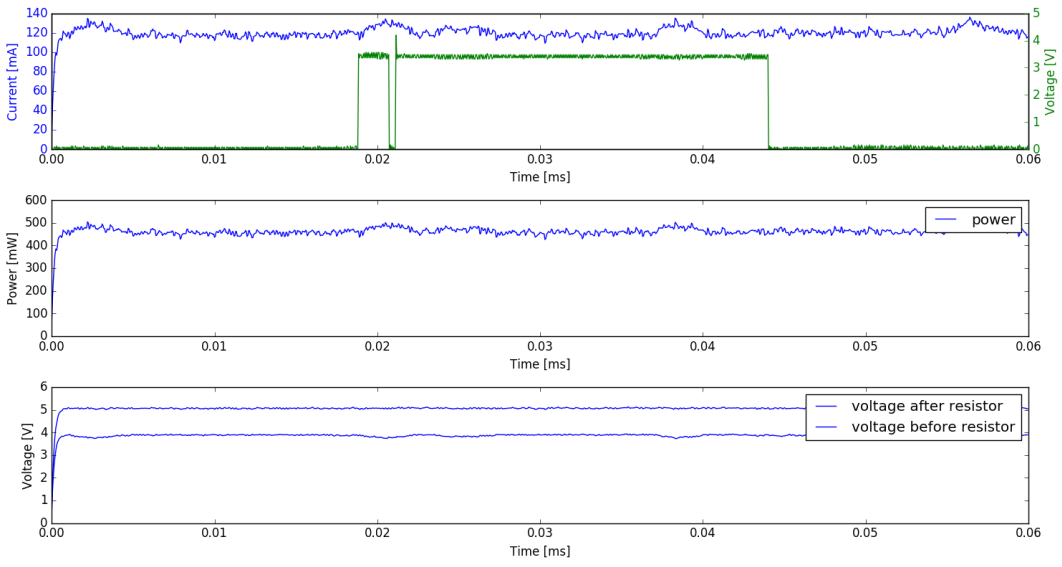

Figure 67: Current and power measurement during single send/receive cycle.

7.3.3 4 bytes

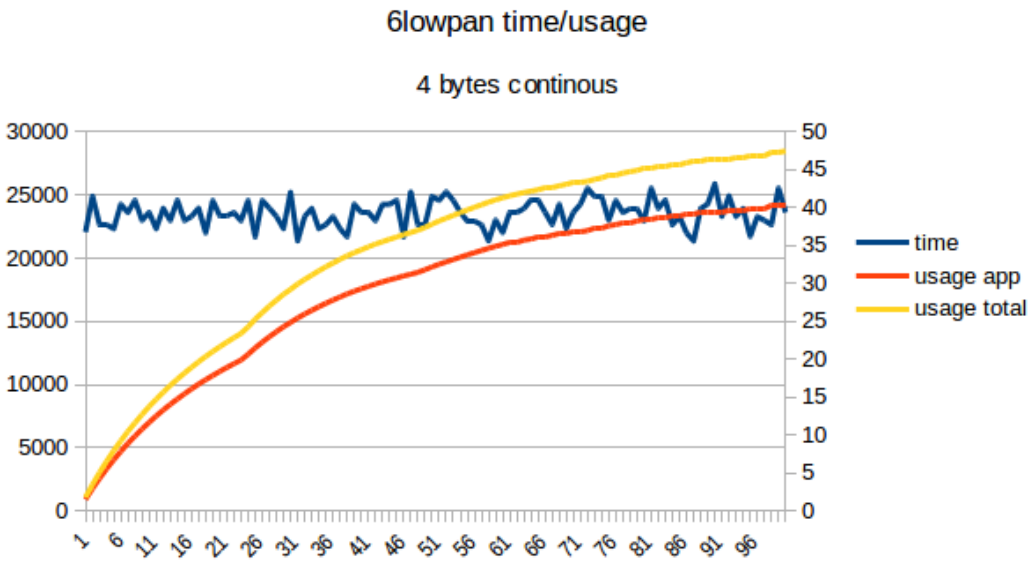

Figure 68: Time of send/receive cycle and cpu usage reported by OS.

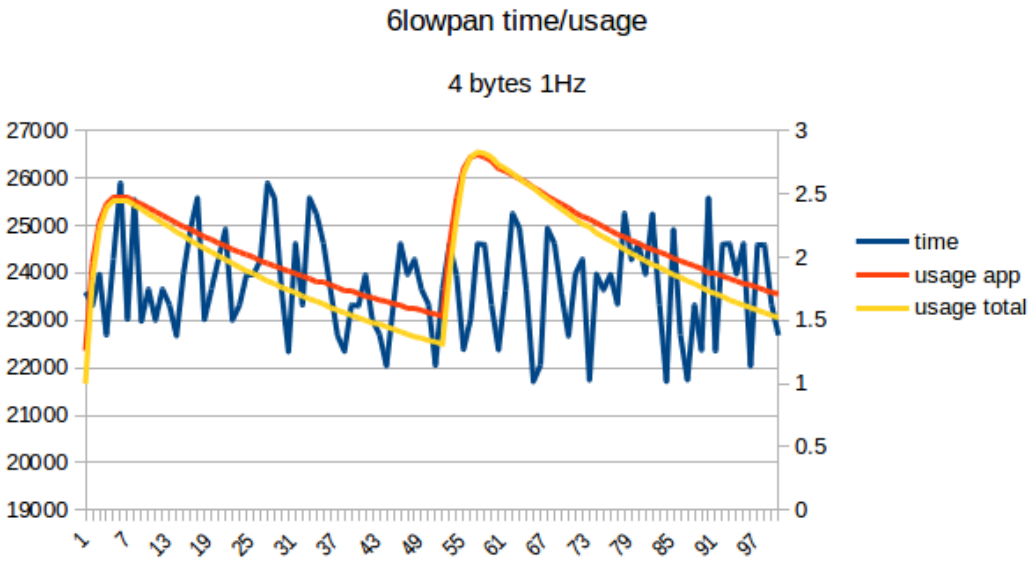

Figure 69: Time of send/receive cycle and cpu usage reported by OS.

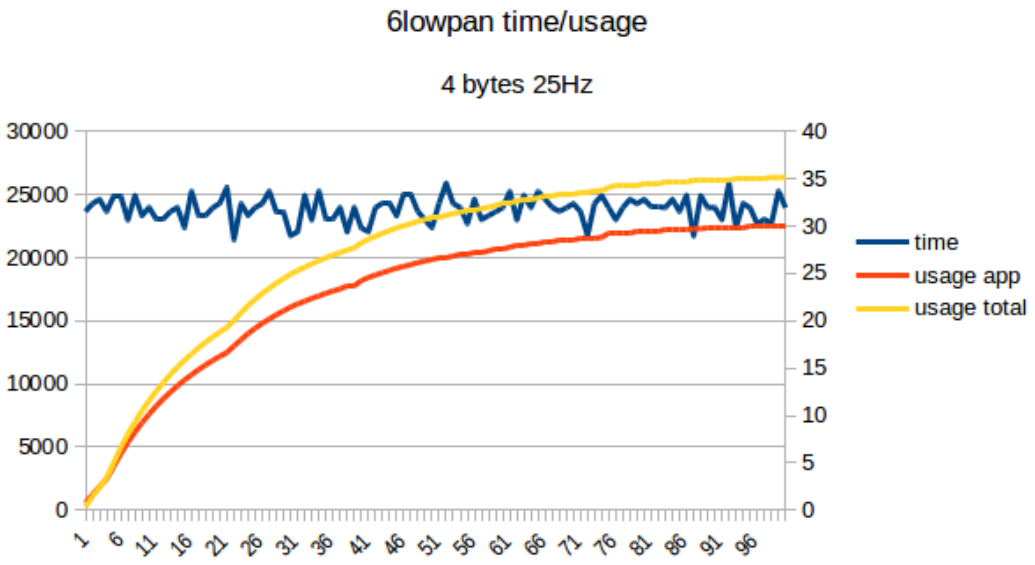

Figure 70: Time of send/receive cycle and cpu usage reported by OS.

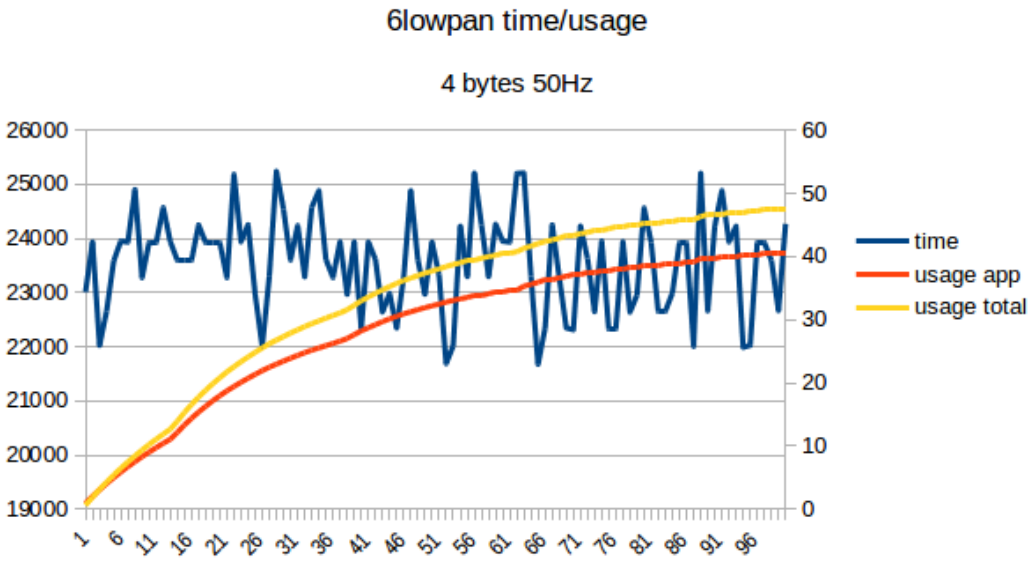

Figure 71: Time of send/receive cycle and cpu usage reported by OS.

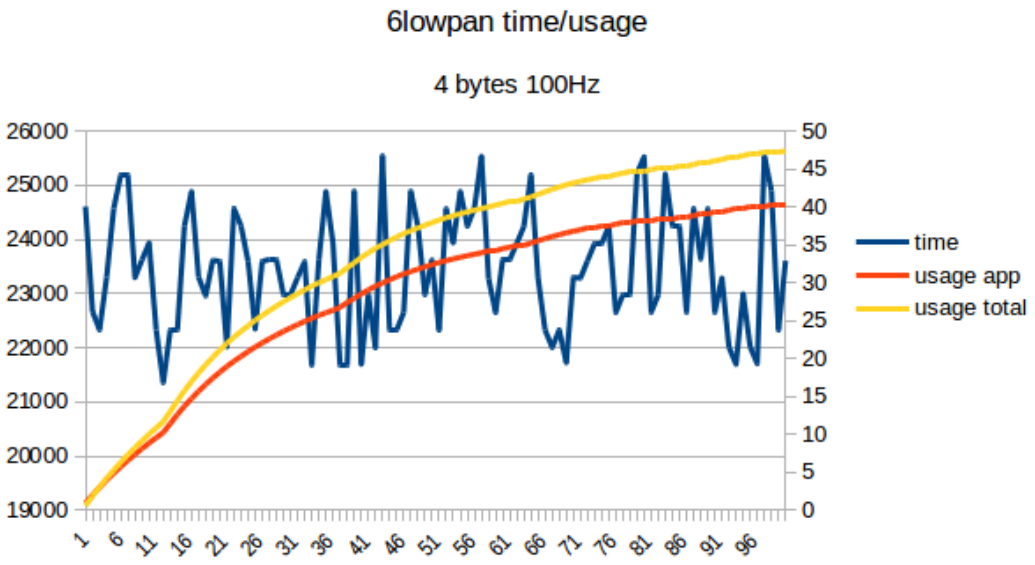

Figure 72: Time of send/receive cycle and cpu usage reported by OS.

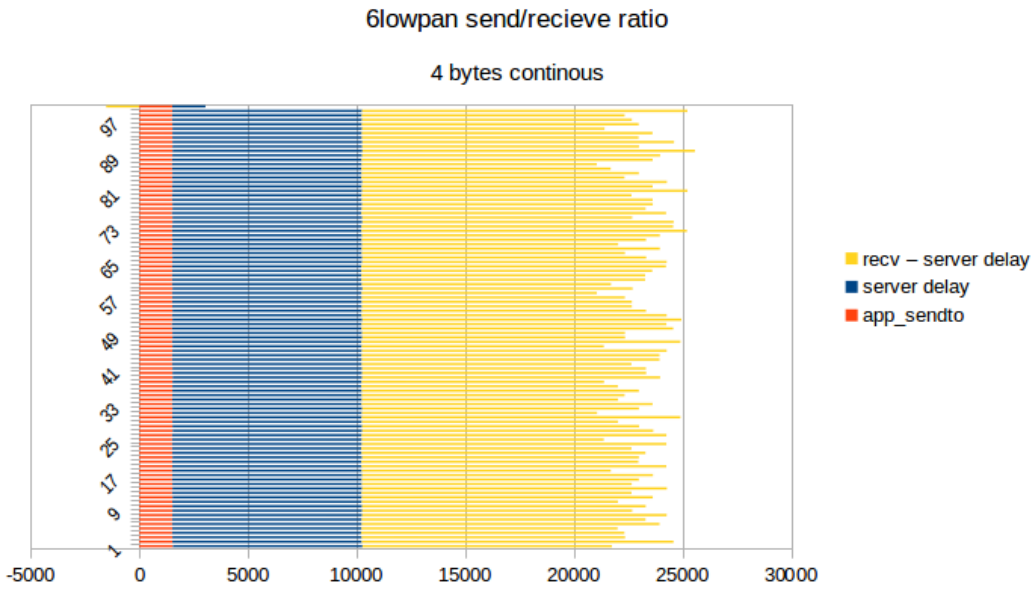

Figure 73: Ratio between time spend sending and receiving. Note that receive time is divided into server delay and actual receive (recv-server delay).

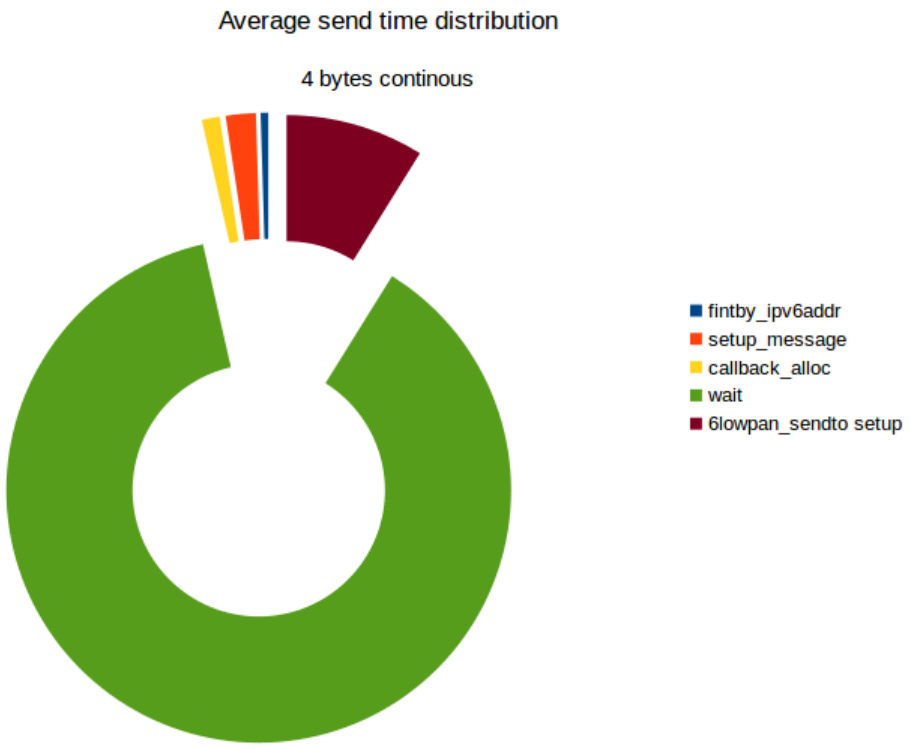

Figure 74: Comparison of time spend in various functions during send part of benchmark (note that this doesn't add to 100%).

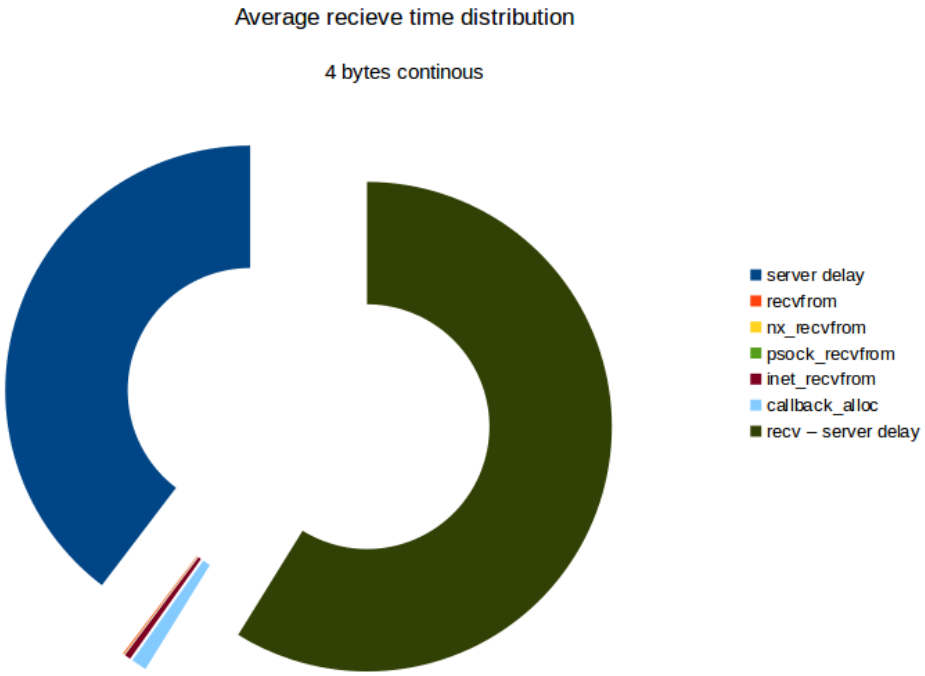

Figure 75: Comparison of time spend in various functions during receive part of benchmark (note that this doesn't add to 100%).

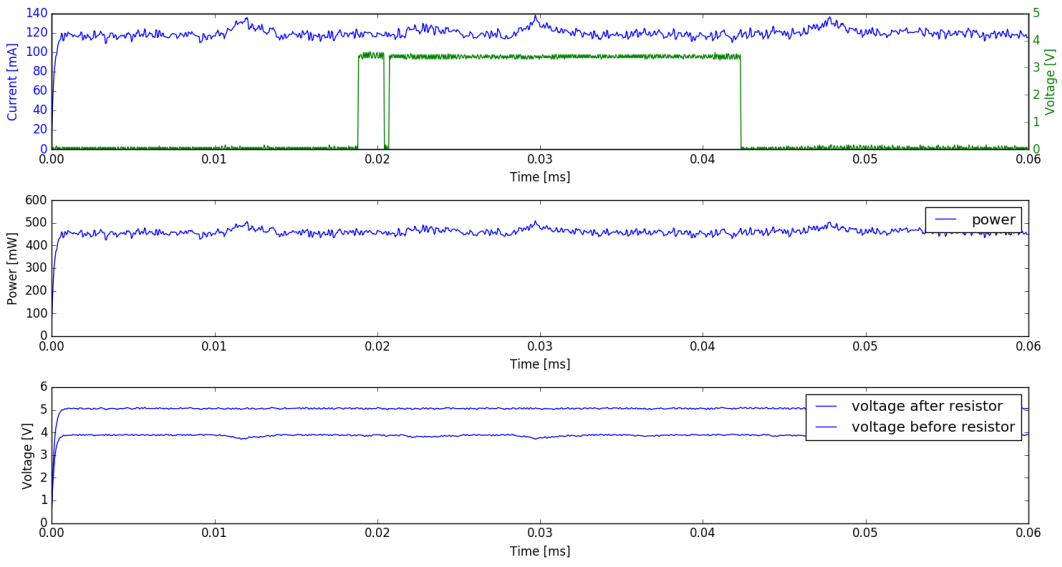

Figure 76: Current and power measurement during single send/receive cycle.

7.3.4 Comparison

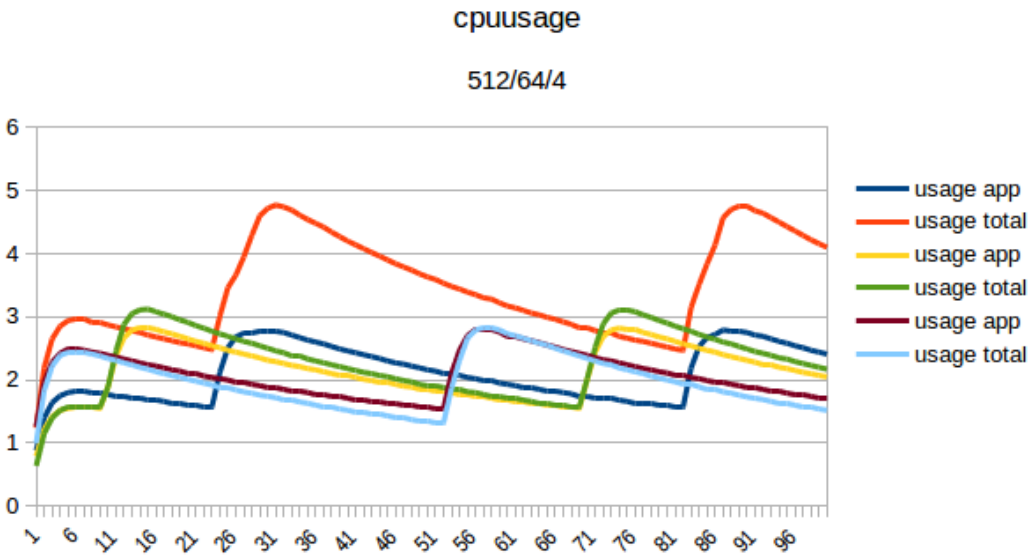

Figure 77: Comparison of send/receive cycle time for 512/64/4 bytes packets send over radio with 1Hz frequency.

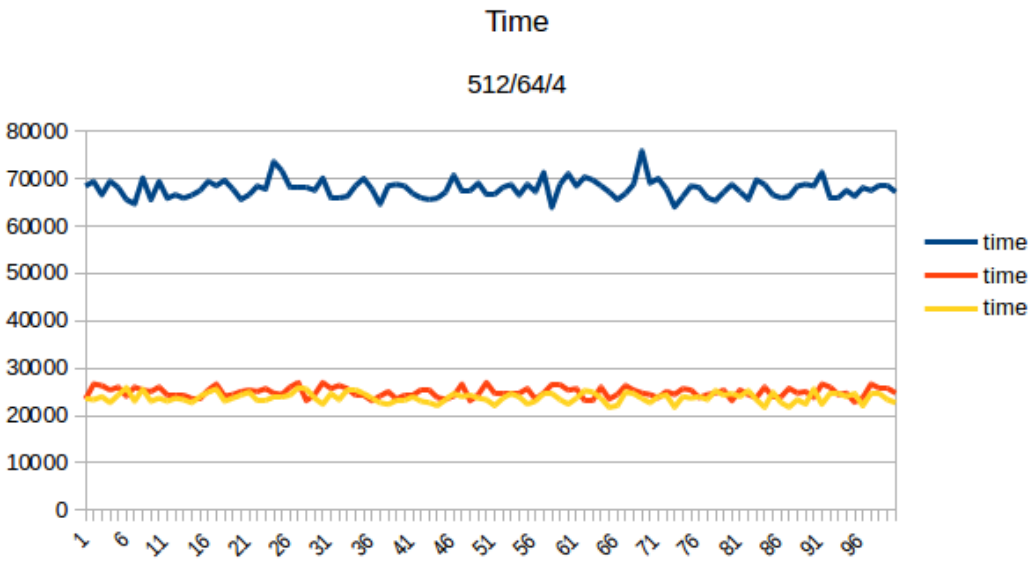

Figure 78: Comparison of cpu usage reported by OS for 512/64/4 bytes packets send over radio with 1Hz frequency.

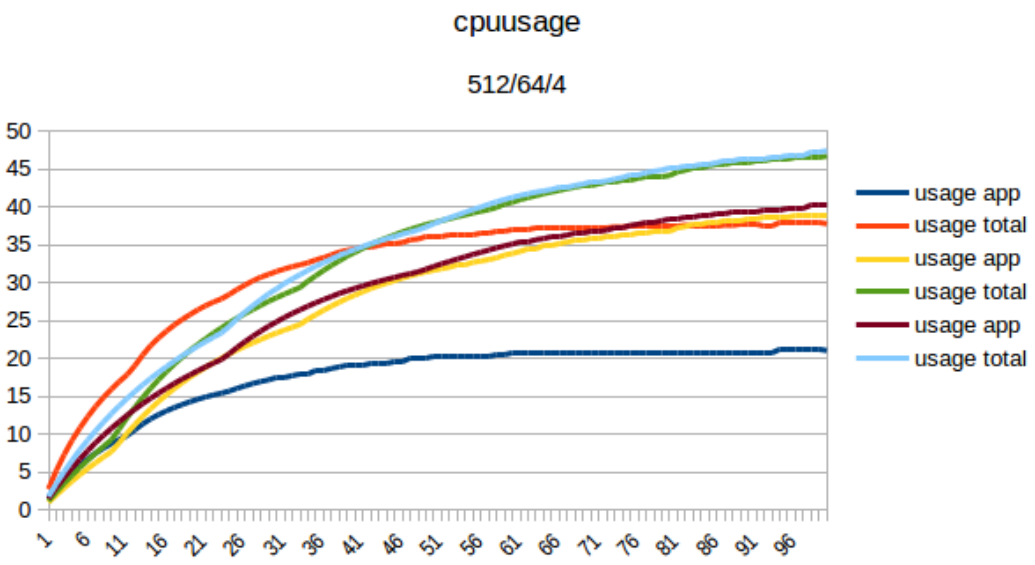

Figure 79: Comparison of send/receive cycle time for 512/64/4 bytes packets send over radio continuously.

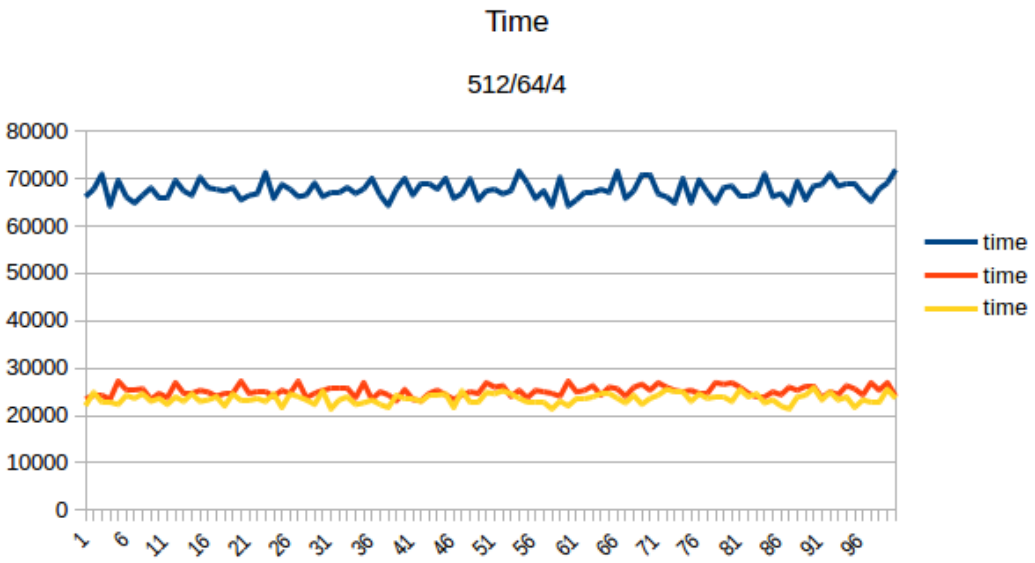

Figure 80: Comparison of cpu usage reported by OS for 512/64/4 bytes packets send over radio continuously.

8 Benchmark results for Discovery board

Benchmarks on discovery board were done differently than the ones on Olimex board. This was due to much lower clock frequency for microcontroller used on this board. In Olimex benchmarks code was augmented with snippets recording timer value (and by this measuring time), in case of

Discovery, this proven not to be precise enough, so I/O was used (and time was measured externally, using logic analyser).

This procedure needs further automation (that will be done, and described in D5.3 MicroROS benchmarking and validation tools). Without those tools, there is no easy way to validate such complex OS like NuttX even in simplest scenario on low end microcontroller.

8.1 Serial

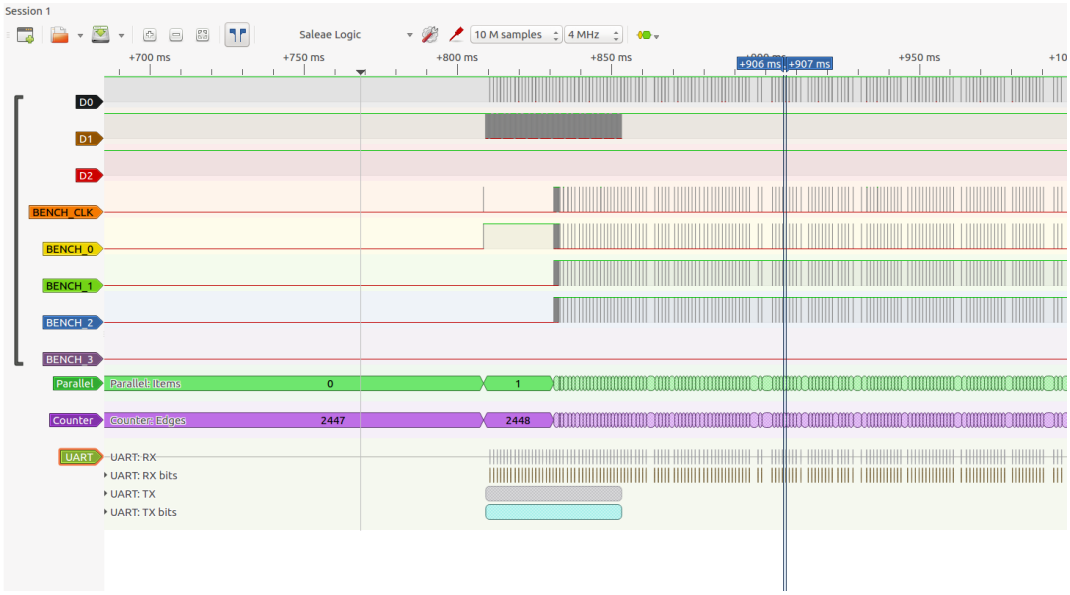

Figure 81: Benchmarking of olimex board

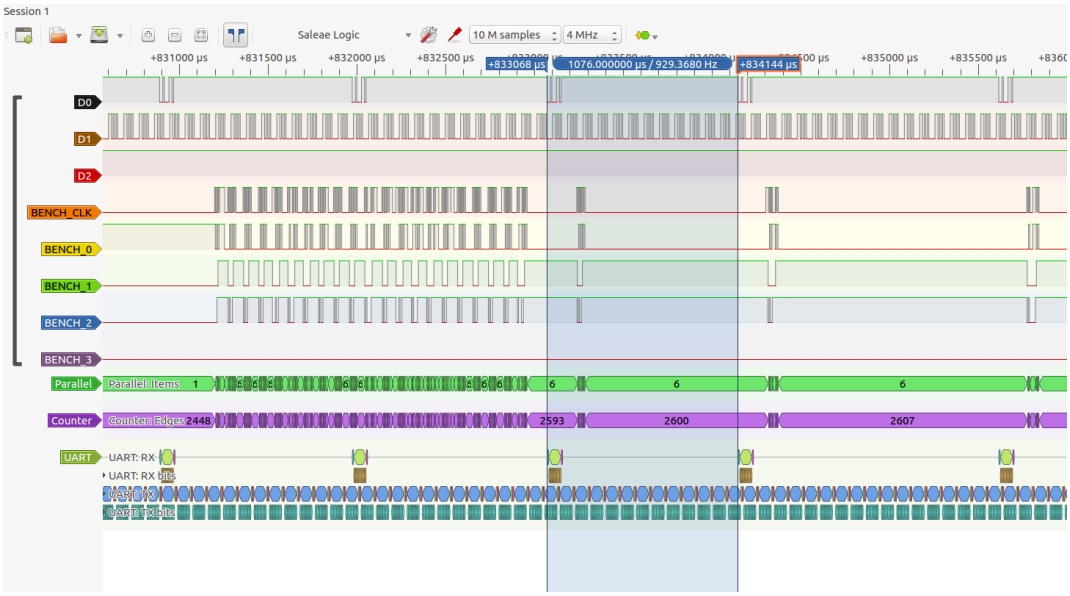

Figure 82: Benchmarking of olimex board

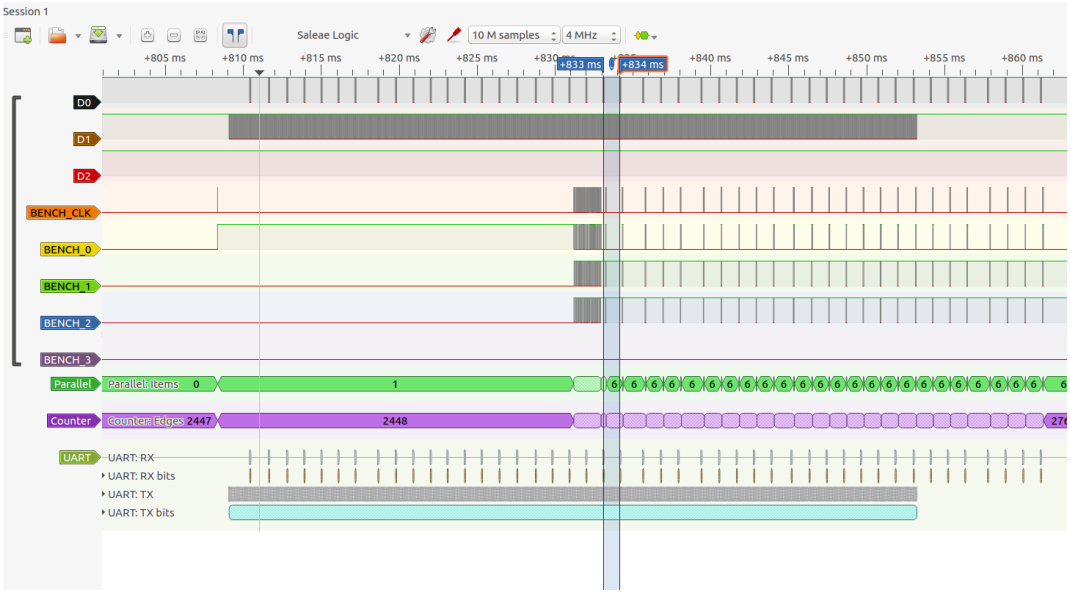

Figure 83: Benchmarking of olimex board

Figures above show attempt of benchmarking serial on Discovery board. This is same test for serial as benchmarked for Olimex.

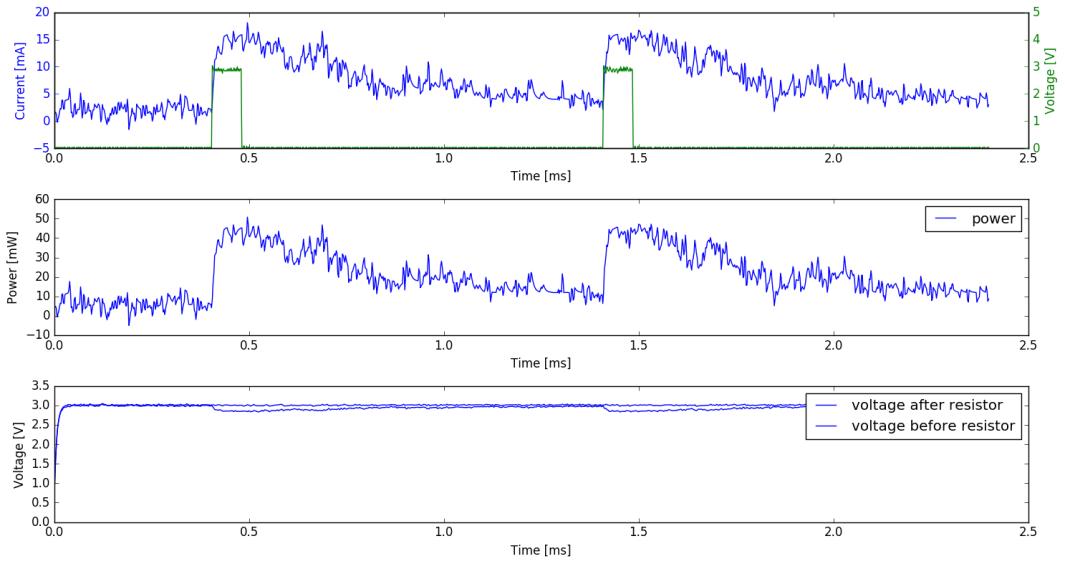

Figure 84: Benchmarking of olimex board

On Olimex board, power consumption was not influenced by running application. This is not the case for discovery board. This is probably due to it's general lower power consumption, ans specially designed power current capability of discovery board. Because of that we can measure just power for microcontroller.

9 Conclusions

Current/power measurement doesn't make much sense here, as basic application doesn't do any power management, therefore power consumption is almost constant. Also for Olimex board a way for more precise measurement of power consumption by microcontroller has to be developed. Current/power values are similar to those measured by ALR and described in D2.1.

Powerful STM32F4 microcontroller on Olimex board gives us plenty of room (both in therms of memory and clock speed) to augment application with benchmarking code. This is not the case for STM32L1. This is reflected in our requirements for tooling described below.

Benchmarking methods shown here are focused on use cases. This allows to ignore certain complexities, and provide results that are important for developers that are in line with prepared use case.

9.1 Requirements for tool set

This chapter summarizes requirements for benchmarking tool set to be developed within project.

1. Toolset needs to offer precise way to measure power consumption, just for microcontroller, synchronized with executed operations.
2. There is a need to measure function execution time in constrained environment. For that ETM trace was identified as possible solution. Alternatively ITM buffer could be used with some code augmentations.
3. Benchmarking toolkit has to be fully automated, to facilitate repeated benchmarks and allow for long time data gathering.
4. Use of proprietary and/or expensive equipment and software should be limited to allow for community use.
5. Platform resource usage should be limited as much as possible. This includes: CPU cycles, memory (both Flash and RAM), I/O.
6. Tools should work both on Olimex and Discovery platforms and be possibly extended to support others in the future.

References

[1] Y. Maruyama, S. Kato, and T. Azumi, 'Exploring the performance of ROS2', in *Proceedings of the 13th international conference on embedded software*, 2016, p. 5.

[2] NuttX, 'Testing TCP/IP Network Stacks'. [Online]. Available: http://nuttx.org/doku.php?id=wiki:networking:testing

[3] NuttX, 'TCP Network Performance'. [Online]. Available: http://nuttx.org/doku.php?id=wiki:networking:network-performance

[4] M. Unemyr, '8 debugging techniques every ARM developer should use'. [Online]. Available: http://blog.atollic.com/8-debugging-techniques-every-arm-developer-should-use

[5] J. Orensanz, 'Advanced Debugging for Cortex-M Microcontrollers'. [Online]. Available: https://www.arm.com/files/pdf/AT_-_Advanced_Debug_of_Cortex-M_Systems.pdf

[6] 'Python package for the Rigol DS1054Z Oscilloscope'. [Online]. Available: https://github.com/pklaus/ds1054z

[7] 'Sigrok webpage'. [Online]. Available: https://sigrok.org/

[8] EEMBC, 'ULPMark, And EEMBC Benchmark'. [Online]. Available: https://www.eembc.org/ulpmark/

[9] ST, 'STM32 Power shield, Nucleo expansion board for power consumption measurement (UM2243)'. [Online]. Available: https://www.st.com/en/evaluation-tools/x-nucleo-lpm01a.html