OULU UNIVERSITY OF APPLIED SCIENCES

Veeti Kraft

PERFORMANCE OPTIMIZATION OF AN EMBEDDED SYSTEM

PERFORMANCE OPTIMIZATION OF AN EMBEDDED SYSTEM

Veeti Kraft
Bachelor’s Thesis
Spring 2019
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Device and Product Design

Author: Veeti Kraft
Title of the bachelor’s thesis: Performance Optimization of an Embedded System
Supervisors: Juha Niemi (ROHM Semiconductor) and Kari Jyrkkä (OUAS)
Term and year of completion: Spring 2019 Number of pages: 34

ROHM Co., Ltd and Kionix, Inc. had a need for a sensor evaluation kit that could process sensor samples at a rate of 25.6 kHz. Sufficiently fast sensors and PC evaluation software already existed but there was no suitable hardware-firmware combination to bridge the gap between computer and sensor. The aim of this thesis was to optimize the firmware of one supported hardware platform to achieve the required sample-processing rate. The work was commissioned by ROHM Semiconductor GmbH.

The work started with research into available tools for measuring the execution time of embedded code. Next, significant bottlenecks were discovered by carefully reviewing the firmware and measuring the execution times of various code sections. Once the bottlenecks were found, possible optimizations were researched and attempts were made at applying them.

The applied optimizations allowed the chosen evaluation kit firmware to process 6-byte wide sensor samples at a rate of 25.6 kHz. The largest performance benefits were gained from general optimizations, such as hoisting invariant checks out of loops, and by optimizing the usage of SPI hardware FIFOs when reading from the bus.

Keywords: Software Performance, Embedded Systems, Sensors, SPI, USB

TABLE OF CONTENTS

VOCABULARY

BLE	Bluetooth Low Energy, a wireless personal area network technology
CDC-ACM	Communication Device Class (Abstract Control Model), a USB subclass definition
CPLD	Complex Programmable Logic Device
FIFO	First-in, First-out
GPIO	General Purpose Input/Output
I^2C	Inter-Integrated Circuit, an inter-IC serial communication protocol
ISR	Interrupt Service Handler, a function that is executed when an interrupt occurs
ODR	Output Data Rate, the frequency at which a sensor refreshes its outputs
PLL	Phase-Locked Loop
RX	Receive
SAR	Successive Approximation Register, a type of analog-to-digital converter
SDK	Software Development Kit
SPI	Serial Peripheral Interface, an inter-IC serial communication protocol
SPS	Samples per Second
TX	Transmit

This document uses decimal and binary prefixes; e.g. 1 kB is equal to 10^3 B and 1 KiB to 2^{10} B.

1 INTRODUCTION

ROHM Co., Ltd. and Kionix, Inc. (a subsidiary of ROHM Group) develop sensors as part of their product portfolio, and they offer various sensor evaluation kits. The most recently published evaluation solution is the RoKiX IoT Development Platform, which is a “complete offering of HW and SW for sensor evaluation purposes” (1, p. 2). The development platform software can be used with a sensor node with embedded sensors or a combination of a development board (e.g. Arduino Uno), adapter board, and sensor boards that are compatible with the adapter board. The development platform software provides a proprietary firmware for the development board and various client programs that run on computers or smartphones. The firmware abstracts the various supported hardware platforms into one interface that the client programs use.

One of the supported hardware platforms, Cypress CY8CKIT-059 with a RoKiX Adapter Board and a Kionix accelerometer, was intended to be suitable for high-speed sensor evaluation but could not achieve the desired sample processing rate of 25.6 kHz. The aim of this thesis was to discover performance issues and find methods to optimize them so that the performance target could be reached.

2 ROKIX IOT DEVELOPMENT PLATFORM

The RoKiX IoT Development Platform is a sensor-evaluation platform that was released in November 2018 and is used in this thesis. The platform can be split into three major components: host adapters, sensor hardware, and client software as illustrated below, in figure 1.

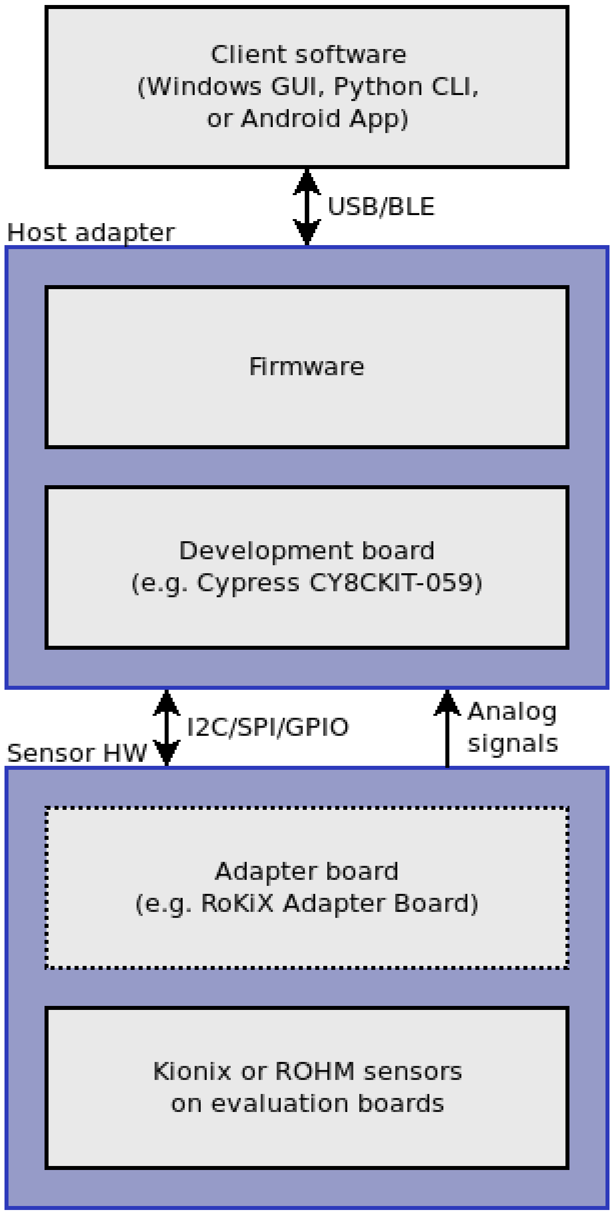


FIGURE 1. RoKiX IoT Development Platform top-level components.

Usually, a computer or smartphone cannot be connected to a sensor directly; hence host adapters and adapter boards are used to bridge sensors to client devices. The adapter boards and sensor evaluation boards work in union to connect the hardware together, and the host adapter provides a uniform way for the clients to control and read the sensors. The host adapter not only serves as a bridge but it also provides some convenience and performance-related features, such as reading a sensor register on GPIO interrupt or writing to a register periodically, both of which are faster and more accurate time-wise than polling interrupt registers or timing intervals on the client side. (1.)

2.1 Hardware

Some RoKiX evaluation kits come with an off-the-shelf embedded development kit (e.g. Arduino Uno or Cypress CY8CKIT-059) combined with an adapter board and some sensor evaluation boards that can be plugged easily into the adapter board. The primary RoKiX evaluation hardware platforms include onboard sensors but as they are not used in this thesis, they will not be mentioned further. (1.)

The hardware used in this project consists of a Cypress CY8CKIT-059 development kit, a RoKiX Adapter Board, and a Kionix KX122 accelerometer. The combination is displayed in figure 2. The remainder of this document will refer to the RoKiX Adapter Board as the adapter board and to Cypress CY8CKIT-059 as the Cypress board for brevity.

The Cypress board features an ARM Cortex-M3-based PSoC 5LP microcontroller, which features a native full-speed USB, SAR and Delta-Sigma ADCs, DMA, and CPLD-based programmable logic. SPI supports frequencies up to 18 MHz, and I^2C can achieve up to 1 Mb/s. The CPU has a maximum operating frequency of 80 MHz, and its clock is generated with a configurable PLL, which allows for considerable freedom in choosing clock frequencies. (2, p. 1.)

The low price, compatibility with an adapter board, and adequate specified performance of the Cypress board were major reasons for choosing the board for the high-performance evaluation kit.

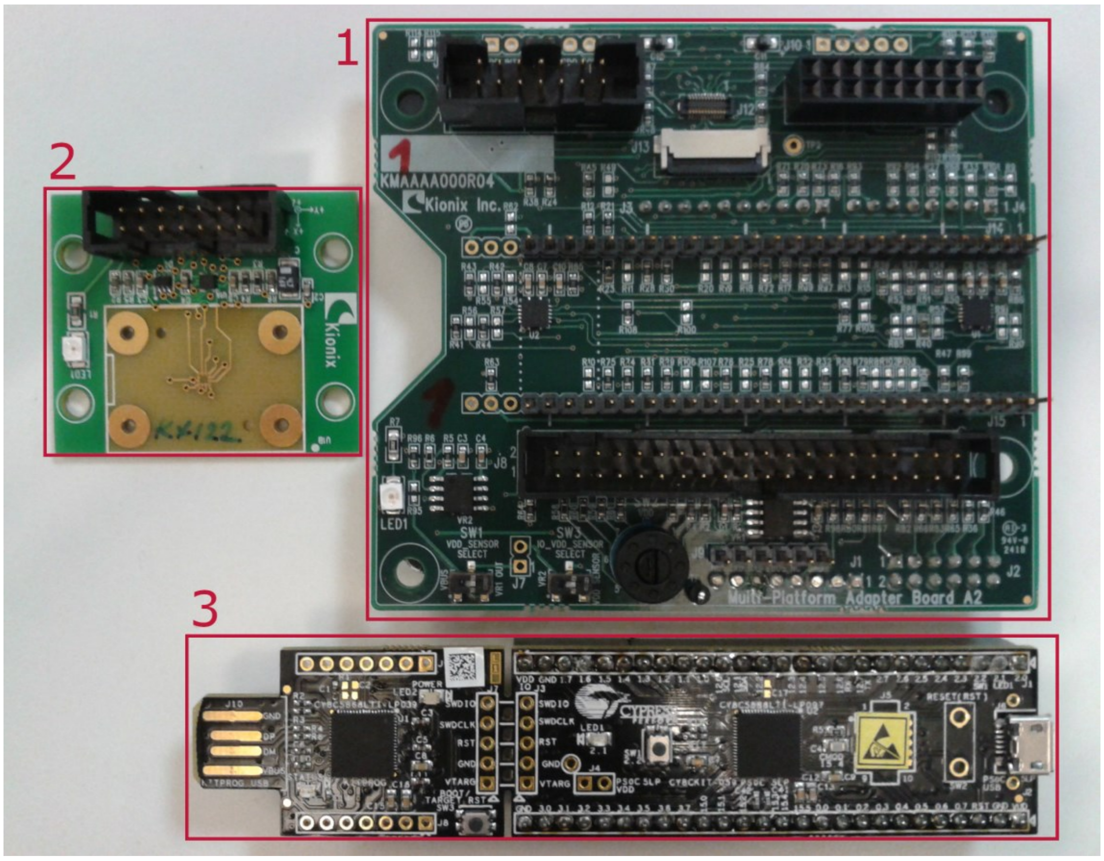


FIGURE 2. RoKiX Adapter Board (1), Kionix KX122 evaluation board (2), and Cypress CY8CKIT-059 (3).

The only requirements for the sensor were that it must support a 25.6 kHz tri-axis accelerometer output data rate (ODR) and at least SPI. The chosen sensor, Kionix KX122, is a tri-axis accelerometer with an integrated buffer, some algorithms (e.g. tap detection), and support for both I^{2}C and SPI. The highest supported ODR is 25.6 kHz, and the highest acceleration range is ±8 g. It comes in a small 2x2x0.9 mm LGA package. Certain other Kionix accelerometers, such as KX112, also met the requirements and could have been used. As long as the sensor meets the aforementioned requirements, the choice is not an important one because once the system has been improved so that it is capable of processing samples at 25.6 kHz, any similar sensor can be used at that rate. (3; 4.)

Surface-mount sensors, such as the chosen KX122, cannot easily be connected to a microcontroller’s GPIO pins, so a sensor evaluation board is used instead of a bare sensor. The Kionix sensor-evaluation board provides a 14-pin connector and supporting components for the sensor and is directly compatible with the adapter board. ROHM sensors have separate evaluation boards with different connectors. As they are not used in this project, they will not be discussed further.

The adapter board support for development kits includes Arduino Uno, several Cypress PSoC kits, and Raspberry Pi. Adjustable level shifters are included on the board for I^2C and SPI communication between hardware that uses different I/O voltages. On the sensor-connector side, there is support for Kionix 14-pin evaluation boards, ROHM 5 or 4-pin sensor shield modules, and RoKiX 24-pin add-on boards. (1.)

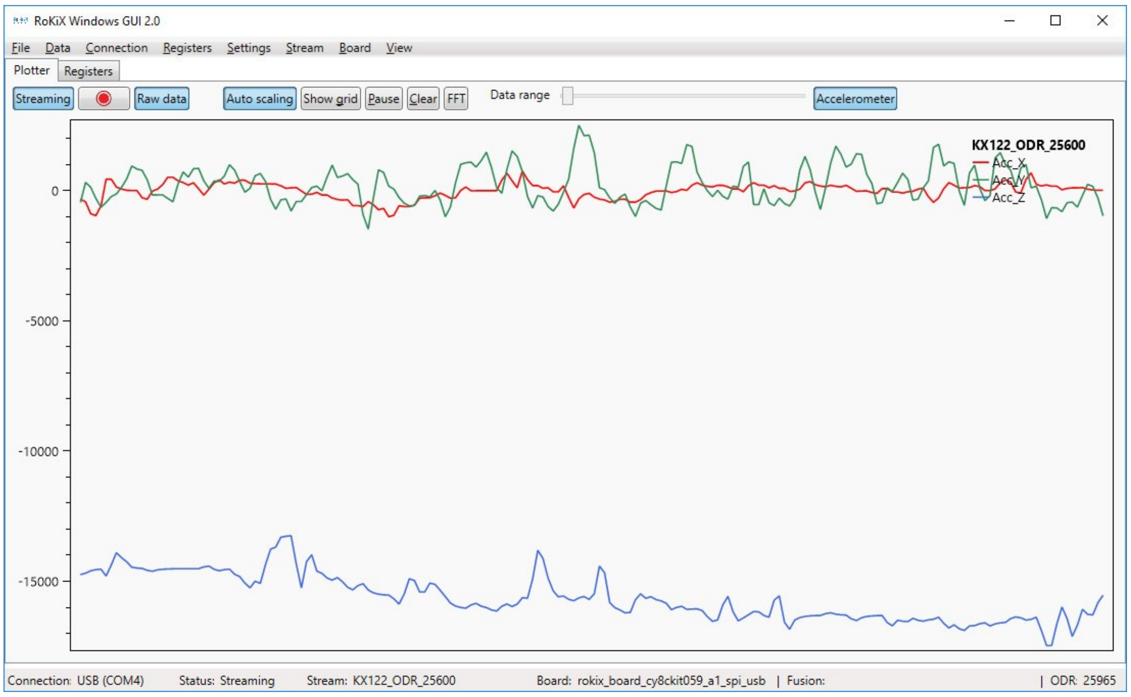
2.2 Firmware

The purpose of the firmware is to offer a uniform interface for manipulating sensors and streaming sensor data from a microcontroller to a computer or smartphone. The interface has been implemented as a binary-message based system. The interface uses the request-response pattern and it can be used to control various features offered by the firmware. The interface is mostly platform-independent; the same messages can be used regardless of the underlying hardware platform. However, not all platforms support all features, and GPIO pin numbers are not portable.

The firmware code has been written in ISO C99 (5) and is split into a portable common core and platform-specific implementations of the platform API that the core uses to control hardware. The core handles message serialization and deserialization and delegation of hardware operations to the hardware via the platform API. The core itself does not directly use any platform-specific features (e.g. vendor APIs or hardware registers). It restricts itself to only use parts of the C standard library and the platform API, which is a platform-independent abstraction over the necessary hardware. The abstraction was created to permit the re-use of the portable parts of code by separating the platform-specific parts behind an interface that can be implemented separately for each platform.

2.3 Client software

The three major clients available are RoKiX Windows GUI (figure 3), RoKiX Python CLI, and RoKiX Android App. They offer interfaces for editing sensor registers, streaming sensor data, and collecting or displaying data. The Windows GUI and the Android App have graphical user interfaces, whereas the Python CLI has a command-line oriented interface. The Android App only works with Bluetooth-capable RoKiX hardware, and thus cannot be used with the Cypress board. The GUI and CLI work with both USB and Bluetooth. The CLI cannot achieve the high data rates used in this project and thus the only client that works with the Cypress board at high data rates is the Windows GUI.



Connection: USB (COM4) Status: Streaming Stream: KX122_ODR_25600 Board: rokix_board_cy8ckit059_a1_spi_usb | Fusion: | ODR: 25965

FIGURE 3. 3-axis acceleration data at 25.6 kHz output data rate (ODR) in RoKiX Windows GUI's plotter-view.

3 SOFTWARE PERFORMANCE ANALYSIS TOOLS

The firmware has to be able to process samples at the frequency of the output data rate of the sensor and it is thus a real-time system. The time window for execution is fixed per sensor ODR and thus it is very useful to be able to measure the execution times of various parts of the system. The measured execution times can tell which parts of the system are taking the largest portions of the available time-window; the parts that consume the most are often good candidates for optimization. The ability to measure execution times is also vital for measuring the effects of attempted optimizations.

All performance profilers, which are not assisted by hardware, add some additional strain to the system under test, which can impact the measurements themselves. The tools also have other qualities that affect measurement resolution, convenience, and accuracy. With a 25.6 kHz sensor ODR, the entire execution time-window in this project is only a mere 39 µs, and thus both resolution and accuracy are important.

3.1 GPIO pin toggling

An oscilloscope or a digital analyzer can provide good but somewhat inconvenient measurement capabilities in conjunction with toggling GPIO pins in firmware. This requires that there are available pins for this purpose. The Cypress board had many unused pins, so this was not an issue. The code below shows an example of measuring the execution time of a function.

```
pin_set();
function_of_interest();
pin_clear();
```

The above pin toggling will cause a high pulse that is as long as the execution time of the measured function. The pin-toggling functions were defined as inline functions in a header file to avoid the function call overhead. The toggling will cause a minor performance difference but it is typically a very cheap operation (one register write for setting or clearing) and was measured to take approximately 200 ns on Cypress. Such a small time was considered to be insignificant in this case.

It is important to add compiler memory barriers to the pin-toggling functions because otherwise the compiler may, in some cases, reorder the statements as part of its optimization passes. An example of a potential reordering is shown below.

```
pin_set();
pin_clear();
function_of_interest();
```

The above example will obviously produce incorrect timings when the pulse is measured with an oscilloscope since the measured function is outside the pin toggles. Compiler barriers are not provided in a portable way by C99, so a compiler specific way of implementing the barrier had to be used. The Cypress toolchain uses GNU GCC as the compiler, which uses the below inline assembler statement as a barrier (6, p. 460).

```
__asm__ volatile ("":::"memory")
```

CPUs that do out-of-order execution (e.g. modern Intel, AMD, and high-end ARM processors) would also require a CPU-specific memory barrier to prevent reordering the GPIO register writes but the Cortex-M3 used in the PSoC 5LP MCU does memory accesses in-order and thus the compiler barrier alone is sufficient (7).

Figure 4 shows an example of measuring main loop, SPI-read, and USB write execution times.

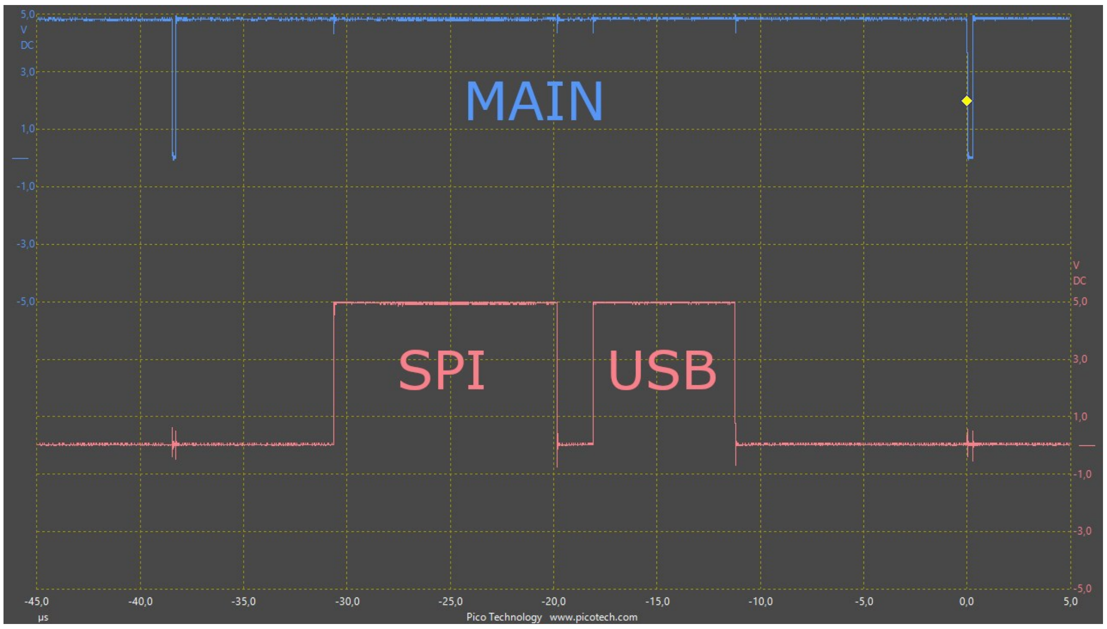


FIGURE 4. Execution time profile for an SPI sensor read and USB write. The MAIN signal measures the main loop time and serves as a reference point for other measurements.

The execution times can differ and thus it is important to know the maximum time for the measured code. This can be achieved by using a pulse-width trigger on the oscilloscope and incrementing the pulse width until the oscilloscope cannot trigger anymore. This trigger feature is not supported by all oscilloscopes but the PicoScope 5243A used in this project had support for it, allowing the measurements to be done. If measuring only one thing and the execution time differs only by small amounts, the persistence display mode (see figure 5) can be a more convenient way to measure minimum and maximum times. The mode superimposes multiple collected waveforms on the same view, allowing the maximum time to be measured from the longest captured pulse. The mode is unsuitable for multiple measurements or large differences because in those cases, the pulses have a good chance of overlapping with each other, rendering pulse edges indistinguishable from each other.

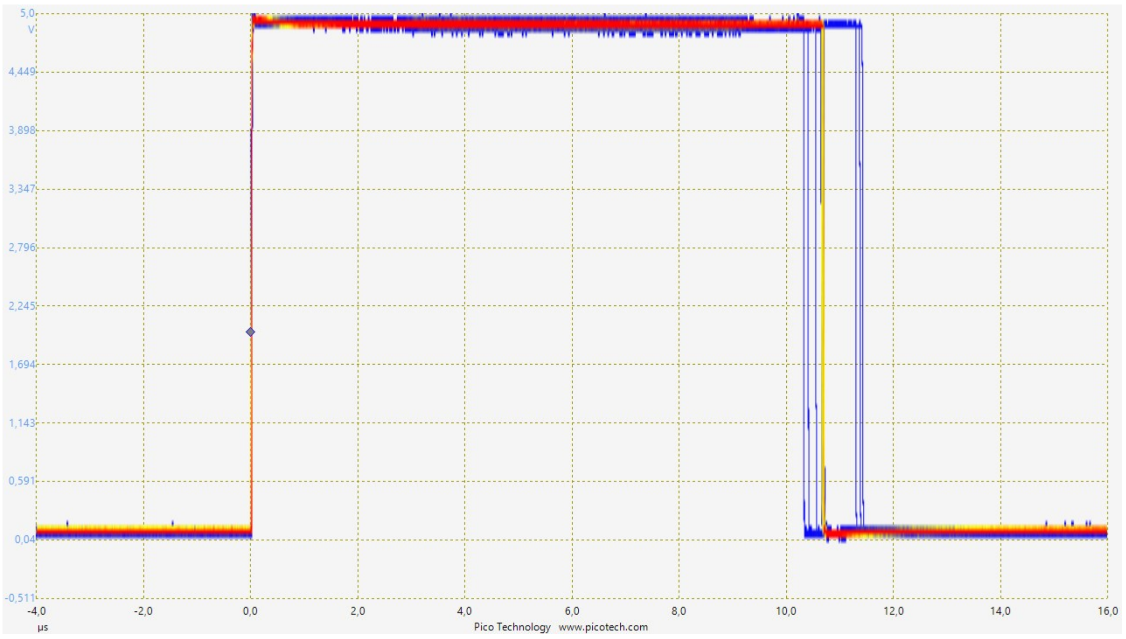


FIGURE 5. Oscilloscope capture in the persistence display mode. The colors go from blue through yellow to red as more samples are collected of the same waveform.

If measuring more than one action per channel, it is necessary to dedicate one channel to some stable reference point, such as the main loop, so that the pulses can be ordered properly. Without a reference point, it is impossible to tell which pulse matches the first pin toggle in the code.

The main drawbacks to this method are that it involves a lot of manual work and measuring several points at once is difficult due to it requiring multiple pins and channels. It is however cheap to set up, it is compatible with every microcontroller that has available GPIO pins, it incurs very little overhead and provides a visual execution time graph as a side effect.

3.2 Execution time profiling in software

After noticing that the pin-and-oscilloscope method was somewhat cumbersome when dealing with several simultaneous, possibly nested, measurements, an attempt was made to write a simple and fast execution time profiler library in C. The library API was developed to provide functions for initialization, starting a delta-time measurement, stopping a delta-time measurement, and printing statistics about the collected samples. The library itself does not automatically instrument code sections; the user must manually add calls to the measurement functions of the library around the code sections of interest. Overlapping measurements were supported by storing the data of each measurement in separate locations. The delta-times were stored as 32-bit unsigned integers with the time unit being platform-specific.

The library was written to be portable and thus it does not rely on platform-specific behavior. Due to this, users must define a specific function for retrieving a timestamp from a monotonic clock. (For Cypress, the system tick module, a 24-bit timer clocked by the CPU clock, was used as a time source.) For convenience, users can also override the end-of-line marker and formatted print function. (Embedded platforms do not often support using printf directly by default.)

An example of a typical usage for this profiler is shown below.

```
perf_init();
/* The arguments are free-form names for the configs. */
struct perf_config *loop_cfg = perf_create("main");
struct perf_config *spi_cfg = perf_create("spi");

for (;;) {
    /* Measure main loop execution time. */
    perf_sample_begin(loop_cfg);
    /* do other work ... */

    /* Measure SPI write execution time. */
    perf_sample_begin(spi_cfg);
    spi_write(/* ... */);
    perf_sample_end(spi_cfg);
    perf_sample_end(loop_cfg);

    /* Print statistics if anything is sent via serial. */
    if (uart_get_char()) {
        perf_print_stat(loop_cfg);
        perf_print_stat(spi_cfg);
    }
}
```

The API of the profiler proved cumbersome in use, as the measurements were often done in separate files and having to keep track of the `perf_config` structures across files was inconvenient. This usability issue could have been improved by keeping the structures within the profiler module and accessing them by name or identifier in user code. This was not done because it would have incurred even more overhead due to the lookups.

The overall performance impact of the profiler was larger than with the oscilloscope-and-GPIO method because of the delta-time calculation and polling the serial line for a trigger to print the statistics. The delta-time sample buffers resided on the heap and the amount of samples was thus restricted by heap size (8 KiB on Cypress). Overall, the library was less convenient and incurred a larger execution speed and memory overhead than when toggling GPIO pins and measuring pulse lengths with an oscilloscope. Due to the aforementioned factors, the library was not used for measurements in this thesis.

3.3 Common profilers

C developers in hosted environments often use tools, such as GNU gprof, Linux perf, or profilers built into IDEs. Some of these tools are operating system specific and by their nature will not work in a free-standing environment. GNU gprof on the other hand would have to be ported to bare-metal targets as it relies on a special system call and file system support, which are often not present on embedded targets (8). The Cypress IDE for PSoC 5LP development, PSoC Creator, does not have any support for performance profiling of any kind. Some of these profilers (e.g. GNU gprof) do not accurately measure function execution times but instead approximate it with function call counting and program counter sampling (8). For the aforementioned reasons, many profilers used in hosted environments are not directly suitable for use in free-standing environments.

3.4 Hardware-assisted profiling

The ARM Cortex-M3 used in the Cypress PSoC 5LP MCU has been configured with the optional Embedded Trace Macrocell (ETM), which is a “real-time trace module providing instruction and data tracing of a processor” (2, p. 61; 9, p. 16). Its use requires a trace-enabled debug probe. Such probes are expensive (e.g. SEGGER J-Trace PRO for Cortex-M costs 1,400 EUR) and were not available for this thesis (10; 11). ETM was not used in this project as it was considered too expensive and excessive for the needs of this project.

However, the Cortex-M3 has another module, the Data Watchpoint and Trace (DWT) unit, which can be used in conjunction with the Instrumentation Trace Macrocell (ITM) to sample the program counter and send it to another device via the Serial Wire Viewer (SWV) (9; 12). The DWT can also be used to generate data about clock cycles, folded instructions, load store unit operations, sleep cycles, cycles per instruction, and interrupt overhead (12). These features would have allowed some form of hardware-assisted performance analysis. Unfortunately, Cypress's own IDE did not support any form of profiling and thus it would have been necessary to find and configure third-party tools. No suitable, easy to use tool was found after a quick search. The hardware profiling features seemed overly complex and hard to use when compared to the GPIO pin toggling method. Also, there was insufficient time to investigate the hardware-assisted approaches further. For these reasons, the hardware-assisted methods were not used in this thesis.

3.5 In-program statistics

While investigating the USB performance, it was noticed that the USB write FIFO sometimes fills up, which forces the firmware to wait until the FIFO has more space. This causes delays that mean that some sensor samples are going to be lost at high sampling rates. Ideally, the FIFO would never block, but a suboptimal implementation or a busy USB host may cause stalls in the USB traffic, which causes the FIFO to fill up. A software counter was added for keeping track of how many times the firmware has to wait for a free FIFO slot. The counter is only accessible by reading the variable with a debugger.

In a typical sensor data streaming case, the sensor is configured to assert a data-ready interrupt signal every time the sensor finishes sampling. The firmware is then configured to use the sensor interrupt pin as an interrupt source for performing actions. It is important to know whether the firmware is able to handle all interrupts in time in order to avoid losing sensor samples. For this reason, two more debugger-readable counters were added for the amount of GPIO interrupts received and the amount of GPIO interrupts handled.

4 OPTIMIZATION

The aim was to be able to read 3 two-byte acceleration values from a sensor and to send them to a client application at 25.6 kSPS. The amount of time the firmware has to do all of this can be calculated by taking the reciprocal of the 25.6 kSPS processing rate, which is approximately 39 µs. The firmware is allowed to occasionally exceed the aforementioned time frame as long as no sensor samples are lost.

As the sensor itself was already sufficiently fast, and the RoKiX Windows GUI client had already been proven to be fast enough, so they were excluded from optimization. That left three major components to investigate: USB, firmware, and the sensor-bus (I^2C or SPI).

Execution times were measured with the GPIO pin toggling method described in section 3.1 because it was sufficiently accurate and the easiest to set up and use with the relatively small codebase of the RoKiX firmware. The performance overhead of the method was also low enough not to disturb the operation of the firmware.

4.1 General

Optimizing the higher-level firmware code, which did not deal with the hardware, involved code-reviewing and applying old, but effective, optimizations.

One of these old and well-known optimizations was to hoist invariant checks out of loops, so that they are done only once. In this case, event parameter validation was moved from an event loop into the event configuration function, so that the checks are done only once during the configuration instead of doing them on every iteration of the event loop.

An attempt was made to enable Link-Time Optimization (LTO) and to switch from the -Os optimization level to the more aggressive -O3 level in the compiler to generate more efficient code (6, p. 111–113; 13, p. 683). The attempt failed as the changes caused several errors deep in vendor code. The errors might have been caused by the vendor code being incorrect or the compiler being too old (ARM GCC 5.4-2016-q2-update). Upgrading the compiler was not an option because only one version was supported by the vendor, and fixing the vendor code would have been infeasible due to the potential size of the effort. Hence, adjusting compiler optimizations was not a good avenue for increasing performance.

The firmware code used manual byte-by-byte copying loops in certain places and some of them were replaced with calls to the C standard library function memcpy, which has often been hand-optimized by the library vendor to perform copies efficiently. Unfortunately, the performance benefit from these changes was not measurable. This may have been due to most copies in the firmware being very small (less than 10 bytes).

In C, pointers that follow the aliasing rules can alias each other (5, p. 68). This possibility of aliasing can cause the compiler to reject many good optimization opportunities because it cannot always determine whether separate pointers actually alias each other or not. One of the most troublesome aliasing cases are character pointers (often used for buffers), which are allowed to alias any object, causing the compiler to be very pessimistic in determining whether something is aliased or not (5, p. 68). C99 defines a *restrict* qualifier for object pointers, which can be used to tell the compiler that the object pointed to by the restricted pointer is not modified by any other pointer in the block in which the restricted pointer was declared (5, p. 109). In other words, the qualifier can be used to tell the compiler that the programmer guarantees that the restricted pointer is not aliased by anything in a block. The sole purpose of the qualifier is to allow the compiler to optimize more aggressively (5, p. 109). An example of using restrict is shown below.

```
void pmemset(char *buf, int * restrict val, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        // Without restrict, the compiler would be forced
        // to reload val on every iteration, because the
        // store to buf might modify val.
        buf[i] = *val;
    }
}
```

In the example function, the restrict qualification on `val` allows the compiler to see that it cannot be modified via `buf`. This allows the compiler to load `val` into a register once before the loop instead of loading it on every iteration of the loop.

Restrict-qualifiers were added to pointer parameters in certain key firmware functions but the effects were not measurable.

4.2 USB

The firmware exposes its services over USB using the CDC-ACM subclass, which is an official subclass defined by USB Implementers Forum, Inc. (14, p. 7). It is intended for PSTN modems but it can be used in embedded products as an easy way to transfer bulk data bi-directionally. Common operating systems, such as GNU/Linux and Windows, have built-in drivers for the subclass and thus there is no need to write a custom driver. With the exception of actual modems, devices that use CDC-ACM for simple data transfer do not implement the mandatory support for the modem command set defined in ITU V.250 and thus they are not strictly conformant.

The CDC-ACM driver is implemented by the USB stack of the operating system, and optimizations on it are outside the scope of this project. All of the optimizations focus on writing because reading is only done for control messages, which are usually not used during an active stream for anything but stopping streams. The USB hardware on the Cypress board communicates at full speed (12 Mb/s nominal signaling rate) (2, p. 1).

The firmware uses bulk data transfers for transferring data over USB. According to Cypress (15, p. 12), isochronous transfers are “intended for streaming data to a host through a constant and real time stream of information,” which indicated that they may be well suited for the real-time needs of the firmware. The primary performance-related differences between the two transfer types are that isochronous offers a higher maximum packet size (1023 B vs 64 B), a guaranteed bandwidth, and a bounded latency (16, p. 21). However, unlike bulk transfers with CDC-ACM, isochronous transfers require a custom driver on the host side and do not guarantee delivery. Due to these factors, switching to isochronous transfers was considered non-viable and was not attempted.

After reviewing the USB transmission code written on top of the vendor USB driver, it was discovered that there was one unnecessary 64-byte copy in the write path. The copy was in the write coalescing algorithm that was used in the USB code module, which combined multiple writes into a packet and then copied the packet into the transmission FIFO, from which the packet was eventually sent. The copying of the combined packet into the FIFO was unnecessary as a packet could be reserved from the FIFO and the writes could be coalesced directly into the FIFO-allocated packet. The USB module was modified to do the aforementioned direct combining into a FIFO packet.

In search of better parallelization, DMA was tried out with USB. The attempt begun with the development of a very simple firmware that would only write constant data to USB as fast as it could and a simple C++ client program that would ingest the data and calculate average throughput. These tools were written for benchmarking the USB API function call execution times and approximate the USB throughput with and without DMA. (The throughput was approximate as it was calculated by dividing the amount of bytes received with the amount of time the transmission took, which does not take buffering or variance into account.)

The preliminary results from benchmarking, shown in table 1, seemed promising and there were no issues with DMA in the simple test firmware. In particular, the execution time reduction with DMA seemed promising.

TABLE 1. Measured USB performance with and without DMA using full 64-byte packets.

	Write(…) execution time (μs)	Average throughput (kB/s)
No DMA	10.5	740
DMA	3.5	940

Enabling DMA for USB was an easy process that involved changing the memory management scheme of the USB component and ensuring that USB write buffers have lifetimes that match the duration of the DMA transactions. While there were no issues with DMA in the test firmware, the actual production firmware wrote only zeros to USB when DMA was enabled. After a significant amount of time spent debugging and reading through Cypress forums, the cause was discovered to be a bug in Cypress's USB driver. The bug was worked around by temporarily removing the bug-triggering non-critical function call from the firmware.

Unfortunately, the switch to DMA caused the approximate maximum sample processing rate to drop from 25 kHz to 23 kHz, which directly contradicted the results gathered from the simple benchmark. No obvious cause was found for the discrepancy. The USB DMA configuration was entirely within vendor code, so optimizing it was not an option. Writing a custom firmware driver for using USB with DMA was briefly considered but would have taken too much time. As a result, the choice was made not to use DMA with USB.

4.3 Choosing the sensor buses

The hardware used in this project supports both I^2C and SPI for sensor communication. There was a need to choose which of these buses would receive optimization and be supported for high-speed sampling.

The selection started by specifying minimal requirements for the buses. The first requirement was that the bus should at least theoretically support the bandwidth required for a 25.6 kHz sample processing rate. This requirement assumes that the sensor-bus can occupy the entire available time window, which is not the case. The requirement is thus not strict enough for a final decision but sufficient to prune out candidates that are far too slow for this project. The absolute minimum sensor-bus bandwidth requirement for the use case of this project can be calculated as

$$b = xf_{\mathrm{s}}$$

where
f_{s} is the sampling frequency
x is the size of a transaction in bytes.

Kionix and ROHM digital sensors use a very simple protocol for manipulating registers via serial buses. In order to read one or more consecutive registers, the address of the first register to read must be written to the slave by the master. Then, the master can read as many consecutive registers as necessary; the sensor will internally increment the register address by one every time a byte is read. This internal, automatic address manipulation is known as auto-increment. Registers that are not consecutive must be read in separate transactions. In I^{2}C, the transfer direction (i.e. write or read) is indicated with a special RW-bit that is part of the slave addressing procedure. SPI, on the other hand, is full-duplex and does not have a built-in mechanism for indicating whether a register is being written to or being read from. Kionix sensors have thus reserved the most significant bit of the register addresses for indicating the direction.

The acceleration data is stored in consecutive registers in the KX122 sensor and thus the data can be read in one transaction. Each sample is two bytes wide and the register address is one byte. Reading the acceleration samples for three axes hence requires a 7-byte transaction. From these values, it can be calculated that the absolute minimum sensor-bus bandwidth requirement for this project is 1434 kb/s.

The I^{2}C bus on the Cypress is operated at full speed (400 kb/s) but due to clock restrictions, the actual clock line frequency is 375 kHz, which translates to a bandwidth of approximately 375 kb/s. I^{2}C also has some innate overhead (slave addressing, per-byte ACK-bit, start, restart, and stop), which further reduces the effective bandwidth. Based on these numbers, even if the overhead is not taken into consideration, it is clear that I^{2}C at 375 kb/s cannot meet the minimum requirements. The Cypress board and sensor both support fast-mode plus (1 Mbit/s), which would have been faster but the adapter board forces the use of I^{2}C pins that do not support such speeds (3, p. 17; 17, p. 1). However, even the faster mode would not have met the required minimum.

SPI, on the other hand, is used at a 9.25 MHz signaling frequency, and has nearly no overhead. This translates to a bandwidth of approximately 9.25 Mb/s, which greatly exceeds the absolute minimum requirement and even leaves some time for other parts of the system to run. SPI was thus the only feasible choice for the sensor bus.

4.4 SPI

The simplest optimization would have been to raise the signaling frequency to something greater than 9.25 MHz. A 750 kHz increase to 10 MHz would have been possible since the sensor supports a maximum SPI frequency of 10 MHz. Unfortunately, this would have forced a 14 MHz drop in the Cypress CPU clock from 74 MHz to 60 MHz as the SPI component must run at twice its signaling frequency and there is no integral divider that could divide 74 MHz into 20 MHz. (Increasing the CPU frequency to the nearest, greater frequency, 80 MHz, would not have worked, because the frequency violates the tolerances of the microcontroller.)

The time difference from a signaling frequency change can be calculated for a transaction as

$$\Delta t = \frac{x}{f_2} - \frac{x}{f_1},$$

where
Δt is the change in time
x is the amount of bits transferred
f_1 and f_2 are the old and new frequencies, respectively.

Reading a Kionix sensor via SPI requires that a 7-bit register address combined with a read-write bit is written to the sensor at the start of a transaction. Hence, reading 6 bytes of sensor data causes a 7-byte transfer. In such a case, the gain from a 750 kHz change in SCLK frequency to 10 MHz would only be a mere 450 ns. Considering that the CPU frequency would have to be dropped by 14 MHz to achieve that, the change would likely only hurt overall performance. Thus, it was decided that the SPI signaling frequency would not be changed.

A look at the bus signals with an oscilloscope (shown in figure 6) revealed that there were significant transmission gaps between each transmitted byte. Each inter-byte gap consumed approximately 1 μs, which is as much time as is taken by sending one byte. In other words, these gaps were reducing the bandwidth by half. A thorough read of the datasheet of the SPI component and the source code of the SPI driver indicated that these gaps are likely caused by the SPI hardware not being given data at a fast enough rate.

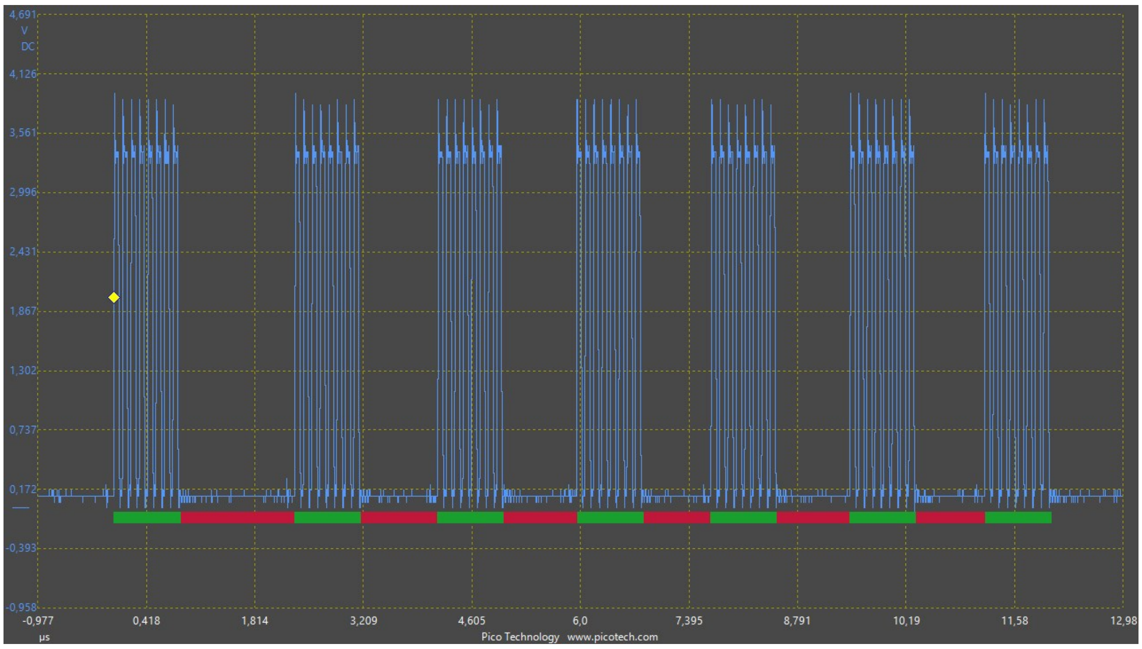


FIGURE 6. Oscilloscope capture of the SCLK signal during a 6-byte sensor read. (The first byte is the register address and a R/W bit).

The SPI module has two operating modes for buffering. The first mode is to only use the 4-byte hardware to receive and transmit FIFOs. The other is to use the software FIFO mode of the driver that keeps the data in RAM. The hardware FIFO is fast but requires careful management so that neither FIFO overflows. The RX FIFO in particular is easy to overflow since for every byte that is shifted out of the TX FIFO, a byte is shifted into the RX FIFO. This means that writes must ensure that there is space in the TX FIFO and also check that the RX FIFO can accommodate for both the new write and everything currently in the TX FIFO. The software FIFO mode has an interrupt service handler (ISR) that moves data one byte at a time between the RAM TX and RX buffers and the SPI hardware FIFOs. More precisely, the ISR writes a byte into the TX FIFO when it is not full and reads a byte from the RX FIFO when it is not empty.

The firmware used the software FIFO but after seeing that the hardware FIFOs were not being utilized well, a new implementation was written. The implementation disabled the SPI interrupt mechanism and software FIFO implementation, managing both hardware FIFOs manually in user-mode by writing into the TX FIFO when there was no risk of overflowing the TX and RX FIFOs. The RX FIFO was read whenever it was not empty to allow more space for writing into the TX FIFO. The aim of keeping the TX FIFO as near to full as possible was to ensure that the hardware would always have something to send so that it would not have to stall the bus and wait for new data. Also, since the FIFO management is done in user-mode, the negative effect of interrupt latency is avoided. The optimized implementation still suffers from occasional gaps but they are far fewer. The gaps are also significantly shorter at 250 μs. The reduction in gaps and their duration is shown in figure 7.

The optimized implementation was tested by reading varying (1–18 bytes) amounts of known, constant data from the configuration registers of the KX122 sensor and checking whether it matched expectations.

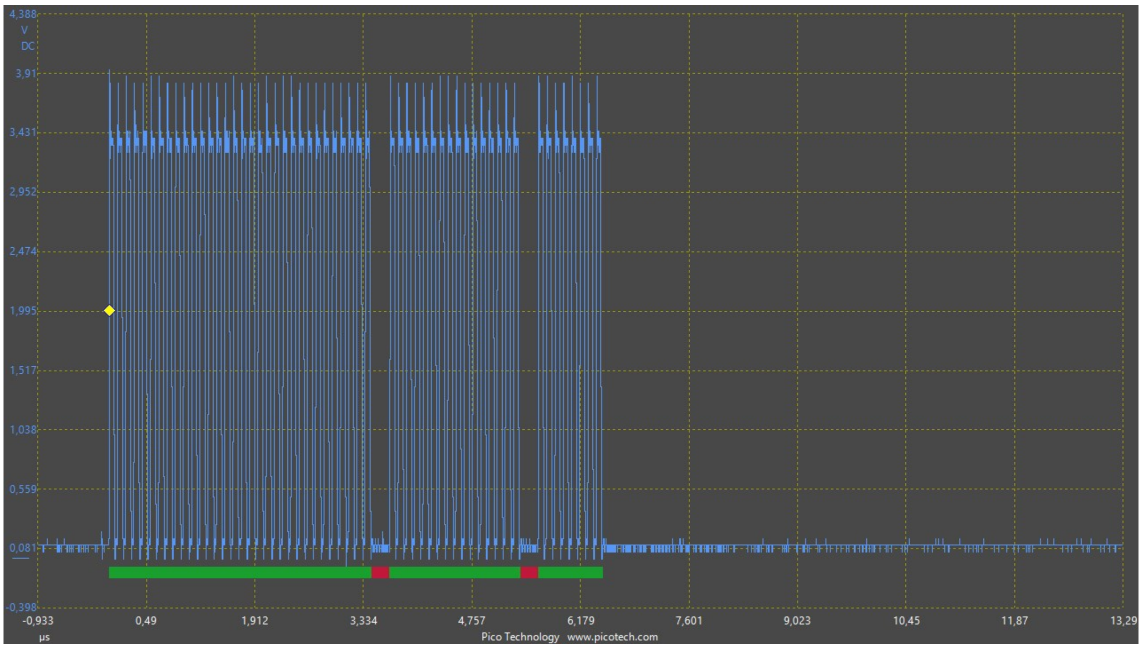


FIGURE 7. Oscilloscope capture of the SCLK signal during a 6-byte sensor read with the FIFO-optimized implementation.

The SPI read function requires that the SPI FIFOs are empty in order to work properly. This was ensured by clearing the FIFOs in the read function before performing the actual read. The FIFO-flushing was measured to take several microseconds and it was decided that instead of ensuring that the FIFOs are empty in the read function, all the other SPI functions would ensure that they leave the FIFOs clean on exit. This moved the flushing cost from read to write, which was beneficial since reading is a far more common operation than writing in this system.

During the implementation of the FIFO-usage optimizing implementation, there was a FIFO overflow bug the cause of which remained hidden for several weeks, and it was suspected that the cause might have been a hardware bug.

During this time, an alternative approach using DMA was attempted. It was written a test firmware that only did some simple SPI operations with DMA. A look at the SPI SCLK signal with an oscilloscope showed that there were no gaps between consecutive bytes during transmission, which seemed promising. However, the execution time of the SPI-read function was worse than that of the unoptimized version with identical transfer sizes. An attempt was made to reduce the amount of DMA configuration in the special case where the only SPI operations are reads and the buffers and read sizes remain constant. The execution time for the special case was lowered to 12.5 μs, of which 6.1 μs was consumed by bus signaling and the rest went into DMA configuration. The overall read execution time remained larger when compared to the FIFO-optimized version. While gapless transfer offers very good performance in general, the fixed cost of DMA configuration dominates the time consumed when doing small reads (e.g. 6 bytes in this project). The complexity of the implementation, bad best-case performance and even worse general performance rendered the DMA option non-viable.

The cause of the FIFO overflow bug was found to be a FIFO upper limit that was too large by one. The bug evaded discovery because there was a slightly misleading comment above the upper limit comparison, which described why a different, unrelated upper limit was too large by one. After the bug had been fixed, the hardware FIFO-optimized implementation was taken into use.

4.5 Verification

There was a need to check that all interrupts were being handled on time even at the target sampling rate. For this purpose, a simple firmware and client were developed. The firmware was written in C for a Nordic nRF52-DK development board and it generates a known amount of interrupt signals at a predefined frequency. The client was a small command-line tool written in C++ that configures the RoKiX firmware to listen for interrupts and send data, to instruct the Nordic firmware to start generating interrupt signals, and then to read the data forwarded by the firmware. The RoKiX firmware was configured the same way as it would have been if it were used with a real sensor. The SPI MISO pin of the Cypress board was grounded as the Nordic board did not support the required SPI frequency and the read values were not used for anything. The purpose of the system was simply to generate N interrupt signals and then check if N indicator messages with the correct amount of data were received. If the generated interrupt and received indicator counts match, the firmware has handled all interrupts.

Also, the RoKiX Windows GUI client was used to run an actual KX122 at 25.6 kHz, and the output data rate display of the GUI was confirmed to constantly be above 25.6 kHz as shown in figure 3. The internal GPIO interrupt and handled GPIO interrupt counters of the firmware also matched, indicating that no interrupts had been missed.

5 CONCLUSION

The aim of being able to achieve 25.6 kHz when streaming 3-axis accelerometer data with Kionix KX122 and Cypress CY8CKIT-059 was achieved. Admittedly, the reliability is not 100% as the host may stall USB traffic for an indeterminate amount of time and the firmware does not have unlimited buffering capability.

Of the investigated software performance analysis tools, the simplest method, GPIO pin toggling, proved to be the most usable option for this project due to its low impact on performance and ease of use with the small codebase. A sampling profiler may have been worth the setup effort in a larger codebase, where the execution paths would not have necessarily been as easy to follow as they were in the RoKiX firmware.

The optimization process involved thinking of alternate ways to do things and trying them out. The Cypress SDK sometimes made this harder than it had to be with somewhat confusing API design choices and suboptimal documentation. Many of the attempted optimizations unfortunately did not result in measurable benefits. The DMA-utilizing optimizations in particular were disappointing. The largest gains were made from the SPI hardware FIFO-usage optimizations. Small improvements, such as moving invariant operations out of loops and removing an unnecessary copy, collectively closed the remaining gap for reaching the target sampling rate. Table 2 shows a compiled view of the attempted optimizations and their effects.

Future optimizations for the hardware-firmware combination used in this thesis may include finding a way to use USB efficiently with DMA or restructuring the firmware internals. Achieving the next, higher sensor output data rate of 51.2 kHz would very likely require more performant hardware and firmware optimization for that hardware.

TABLE 2. Attempted and considered optimizations and their effects. (NA indicates that the optimization did not work at all or was not attempted.)

Optimization or change	Effect on performance
Better use of SPI HW FIFOs	Major benefit
Removal of one USB copy	Minor benefit
Hoisting invariants out of loops	Minor benefit
Use of DMA with SPI[1]	Minor benefit
Copy loops to memcpy calls	None
Selective use of the restrict qualifier	None
Use of DMA with USB	Negative
More aggressive compiler optimizations	NA
Use of isochronous USB transfers	NA

[1] The optimization is mutually exclusive with the SPI HW FIFO optimizations.

REFERENCES

1. ROHM Co., Ltd. 2018. RoKiX Platform User's Guide. Revision 001. Available: https://github.com/RohmSemiconductor/RoKiX-IoT-Platform

2. Cypress Semiconductor Corporation. 2018. PSoC 5LP: CY8C58LP Family Datasheet. Revision *M. Available: http://www.cypress.com/file/45906/download

3. Kionix, Inc. 2018. ± 2g / 4g / 8g Tri-axis Digital Accelerometer Specifications. Part number: KX122-1037. Revision 6.0. Available: https://www.kionix.com/product/KX122-1037

4. Kionix, Inc. 2018. ± 2g / 4g / 8g Tri-axis Digital Accelerometer Specifications. Part number: KX112-1042. Revision 6.0. Available: https://www.kionix.com/product/KX112-1042

5. ISO/IEC 9899:1999. 1999. Programming languages — C.

6. Stallman, Richard – GCC Developer Community. 2015. Using the GNU Compiler Collection. For GCC version 5.5.0. Date of retrieval: 10.12.2018. Available: https://gcc.gnu.org/onlinedocs/gcc-5.5.0/gcc.pdf

7. ARM Limited. 2011. Does Cortex-M3 need Memory Barrier instructions? ARM Technical Support Knowledge Articles. Date of retrieval: 21.12.2018. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13593.html

8. GNU prof. Date of retrieval: 21.12.2018. Available: https://sourceware.org/binutils/docs/gprof/index.html

9. ARM Limited. 2011. Embedded Trace Macrocell. ETMv1.0 to ETMv3.5. Architecture Specification. Version Q. Available: https://static.docs.arm.com/ihi0014/q/IHI0014.pdf

10. Magnus Unemyr. 2016. The Ultimate Guide to ETM Instruction Tracing [Cortex-M debugging]. Atollic TruePERSPECTIVES Blog. Date of retrieval: 14.1.2019. Available: http://blog.atollic.com/the-ultimate-guide-to-etm-instruction-tracing-cortex-m-debugging

11. SEGGER Microcontroller GmbH. J-Trace PRO. Date of retrieval: 21.12.2018. Available: https://www.segger.com/products/debug-probes/j-trace/

12. ARM Limited. 2010. Cortex-M3. Revision r2p0. Technical Reference Manual. Available: https://static.docs.arm.com/ddi0337/h/DDI0337H_cortex_m3_r2p0_trm.pdf

13. Stallman, Richard – GCC Developer Community. 2019. GNU Compiler Collection Internals. For GCC version 9.0.0 (pre-release). Date of retrieval: 23.1.2019. Available: https://gcc.gnu.org/onlinedocs/gccint.pdf

14. USB Implementers Forum, Inc. 2007. Universal Serial Bus Communications Class Subclass Specification for PSTN Devices. Available: https://www.usb.org/document-library/class-definitions-communication-devices-12 (file `CDC1.2_WMC1.1/usbcdc12/PSTN120.pdf` in the archive).

15. Cypress Semiconductor Corporation. 2017. PSoC 3 and PSoC 5LP – Introduction to Implementing USB Data Transfers. Revision *M. Available: http://www.cypress.com/documentation/application-notes/an56377-psoc-3-and-psoc-5lp-introduction-implementing-usb-data

16. Compaq – Hewlett-Packard – Intel – Lucent – Microsoft – NEC – Philips. 2000. Universal Serial Bus Specification. Revision 2.0. Available: https://www.usb.org/document-library/usb-20-specification (file `usb_20.pdf` in the archive).

17. Cypress Semiconductor Corporation. 2015. PSoC Creator Component Datasheet. I2C Master/Multi-Master/Slave. 3.50. Available: http://www.cypress.com/documentation/component-datasheets/i2c-mastermulti-masterslave