High-Speed Data-Acquisition for FlockLab

Master Thesis

Benjamin Dissler

April 10, 2014

Advisors: **Roman Lim, Christoph Walser**
Supervisor: **Prof. Dr. Lothar Thiele**

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

In this Thesis we present an enhancement to the wireless sensor node network testbed FlockLab, the data acquisition unit (DAQ). The DAQ is placed between the monitored target node and the monitoring embedded computer. The DAQ detects events at a higher rate (10 MHz) and applies precise timestamp to an event, with a resolution of $500ns$. This permits to trace function calls of a target node, which was not possible accurately with the former rate of 12.5 kHz.

Further, we synchronize the internal time more tightly than with the NTP protocol by using an GPS PPS signal as reference. A timing error as low as $30ns$ allows to trace time critical network protocols, running on different targets. We implement the DAQ on a FPGA to ensure high detection rate and be able to simultaneously adjust time to the PPS signal.

Acknowledgments

I would like to thank my advisor Roman Lim for his support to realize my Master Thesis and the freedom he gave me during my work.

Further I appreciated all the inputs of my co-advisor Christoph Walser and his helpful instructions for the more practical tasks.

I would like to express my gratitude to Prof. Dr. Lothar Thiele for the opportunity to contribute to a project of the Computer Engineering group.

Then I would like to thank my fellow students Markus Frei, Etienne Geiser and Balz Maag for the great time we had, sharing the same office.

Contents

Chapter 1

Introduction

To evaluate and debug wireless sensor node (WSN) networks, before rolling them out in the field, a testbed is needed. Further, expensive changes in the field, due to bugs, can be omitted by testing a WSN network in a realistic environment before application.

FlockLab [1] is a testbed for wireless sensor nodes. It monitors in a least intrusive way, by tracing GPIO pins, up to 30 distributed sensor nodes. A monitored WSN is called a target, and is located on an observer. An observer consists of a small embedded computer, the Gumstix, and the FlockBoard. The FlockBoard connects the Gumstix to the target nodes. On a FlockBoard up to 4 target nodes can be located, but only one monitored. The Gumstix can influence a target, collects data, attaches a timestamp to the data and sends it to a central server per Ethernet or Wi-Fi.

FlockLab is the complete composition of targets on a observer, connected to a central server, including the software to control the installation and run tests. FlockLab features 4 major services:

- **Tracing:** capture signal changes (GPIO events) of up to 5 output pins of a target. Where the timestamp allows to relate events on different observers.
- **Actuation:** set up to 3 input signals of a target, e.g. to initiate a procedure on multiple WSN simultaneously.
- **Power profiling:** measure the current drawn of a target with an ADC (analog digital converter). The power profiling can be started and stopped at arbitrary times.
- **Callback:** allow a target node to start/stop power profiling trough setting one of its output (tracing) pins.
- **Serial logging:** read or inject data over the target's serial port.

The core is a Gumstix embedded computer. The Gumstix sets pins, collects power samples over SPI and captures and timestamps tracing events. The tracing signals generate an interrupt in the Linux OS on the Gumstix when changed. By handling the interrupt the Gumstix attaches a timestamp to the event, before storing the sample on a SD card.

Current limitations The time needed for the interrupt handling on the Gumstix limits the rate of events that can be detected. Table 3 of the paper [1] shows that consecutive events of a single signal need to be $290\mu s$ apart to be detected correctly. This value decreases to $80\mu s$, if events happen interleaved on different signals. Further the delay between an event triggering an interrupt and the handling of the interrupt is non deterministic. Hence it adds an error to the timestamp precision. The timestamps applied to an event are synchronized with the NTP protocol over all observers. As stated in Table 2 in paper [1], this leads to pairwise timing errors of different observers of up to $1170\mu s$, when Wi-Fi connected observers are involved. The error decreases to $394\mu s$ if just Ethernet connected observers are considered and in average it is below $40\mu s$. This is sufficient for many tests.

Motivation Improvements are needed for example function call tracing using GPIO events. Therefore, tracing pins could be set by entering a function, and the functions can be encoded by using sequences of the 5 tracing pins. A target node running the Opal platform could generate such sequences at a rate of 96MHz. Hence, bursts of high data rates need to be handled.

With a more tightly synchronized time over the observers, events could be correlated more precisely to monitor time critical behavior of network protocols. For example the Glossy [2], a protocol that uses synchronously sent packets. Then to decode the packets at the receiver, they need to be received virtually at the same time. Thus to trace the parallelism of Glossy, the pairwise time error of observers should be in the sub-microsecond range. Further a high resolution and precise timestamp is needed. Preferably with sub-microsecond accuracy, too.

Challenges The main challenges are to provide a high data detection rate and simultaneously ensure high precision in timestamping. Additionally, a more precise approach than the NTP protocol is needed, to synchronize the internal time of the different observers. Due to integration into an existing environment certain requirements are given, and options therefore are limited.

Our contribution We improve the FlockLab by introducing a data acquisition unit (DAQ). The DAQ is placed in between the Gumstix and the target node, and traces events of a target at up to 10MHz over a short time period. Then the DAQ buffers the data locally before forwarding it to the Gumstix. The maximum continuous rate to not exceed the buffers of the DAQ is 285000 events/second, i.e. in average one event every $3.5\mu s$. The power profiling samples are captured too and merged into the same buffer mechanism. Therefore, all data besides the serial logging is transmitted to the Gumstix over one link. Further, a highly precise pulse per second (PPS) signal, e.g. from a GPS receiver, is used as reference to synchronize the time of the DAQs of different observers. Our implementation can apply timestamps to an event with a resolution of $500ns$. At the same resolution actuations can be set. The time drift, in respect to the reference signal, after one second is around $10ns$, with a standard deviation of $10.8ns$.

Outline We start in Chapter 3, by defining the requirements the DAQ has to meet. Further we determine the hardware and interfaces to implement the DAQ with.

Then in Chapter 4, we give a detail description of the implemented prototype. How an accurate internal time is released and how the DAQ handles high event rates. We describe all in- and outputs, and specify the communication packets between the DAQ and the Gumstix.

Further in Chapter 5 we evaluate the prototype in terms of time accuracy, and measure the maximum event rates and data throughput the DAQ can handle. Additionally we provide a testcase, monitoring a target node. The collected data is presented and compared to the system without the DAQ.

We conclude in Chapter 6 recapitulating the major achievements and give an outlook of future work.

Chapter 2

Related Work

For related work, we look into another FPGA based pin tracing environment and its throughput. Then in terms of accurate clock adjustment, we provide two related papers. One adjusts an oscillator to a GPS PPS signal. The other shows a completely degital cirquit, implemented on a FPGA, to reduce the jitter of a GPS PPS signal.

A similar project in order to realize signal tracing is shown in [3], a Logic Analyzer. Their setup on an Altera Cyclone III developer board features a FPGA with 25k logic elements as core unit. The logic analyzer is a stand alone device, which implements the tracing of 8 signals It combines the pin levels with a timestamp into a sample then buffers and stores the samples before finally sending the results over Ethernet to the workstation of a tester. Three stages of hierarchical memory is used: From a fast onboard FIFO over slower SSRAM to large but even slower DDR SRAM memory. We use a similar buffer memory hierarchy, however only with two stages. The Logic Analyzer implements data capturing and buffering in a fast clock domain with 150MHz. The data storing in DDR SRAM and communication with a workstation over Ethernet is handled by a soft core CPU synthesized in a slower clock domain. The Logic Analyzer of [3] shows a sample rate of 150MHz, but with an uncalibrated timestamp. Our implementation of the DAQ works with a 100MHz sampling rate and a timestamp resolution of $500ns$. But in contrary to the Logic analyzer the timestamp is highly accurate over distributed nodes. Additionally we add actuation capabilities and other data streams in our implementation.

A possibility to adjust the clock of multiple distributed devices to a GPS PPS signal is shown in [4], where time and the phase is synchronized. They use a ultrastable local oscillator (USO) to generate a 10MHz GPS-disciplined clock signal. To lock the USO to the PPS signal a direct digital synthesizer (DDS) driven phase locked loop (PLL) synthesizer is implemented. Basically the signal of a very precise oscillator and a PPS signal are modulated and filtered (using digital-analog and analog-digital converters) to generate a stable and phase aligned output signal. Results are theoretically validated without experimental evaluation. Therefore no measurements are available to compare our results with. They assume “somewhat idealized conditions; hence there may be some variations in actual systems”. Our implementation provides a complete digital circuit on a FPGA to adjust a time representation to a PPS signal, with a much simpler approach. Additionally requiring only the FPGAs default oscillator as additional hardware.

A digital circuit, implemented on a Spartan-3 FPGA, to reduce jitter of a GPS PPS signal is shown in [5]. The circuit is clocked by a 100MHz crystal oscillator, and runs a Proportional and Integral (PI) digital controller, similar to a digital PLL. The PI controller takes the GPS PPS as input and generates a PPS signal, with less jitter, as output. The start value for the PI controller is the average GPS PPS period in clock cycles, measured over multiple periods. Then the output PPS is continuously adjusted, by measuring the delay between input and output PPS signal in clock cycles. With this approach the jitter of a Ublox LEA-4T GPS device could be reduced by more than 60%.

Our implementation on the DAQ is very similar. The average PPS period in clock cycles is used as base, and the delay between the estimated and actual PPS signal adjusts our time calibration. In contrary to the circuit in [5] we use the PPS as reference signal instead of stabilizing the signal.

Chapter 3

Design

In Section 3.1 we will explore which requirements the DAQ has to meet in terms of hardware connections, as well as the expected data rates that need to be handled. We define the time accuracy to reach and how the DAQ can be configured from the Gumstix.

With the gathered requirements we will select, in Section 3.2, appropriate interfaces and the main processing unit. The interfaces are used for communication and data transfer between the DAQ and the Gumstix.

3.1 Requirements

The Data Acquisition Unit (DAQ) is a hardware unit placed in between the Gumstix embedded computer and the rest of the FlockLab board, connected through a 60 pin connector, as shown in Figure 3.1. The DAQ needs to serve and timestamp 3 services: tracing, actuation, callback and power profiling. Those events are captured and the state of the pins or the power profiling sample including a timestamp is saved in local memory. Since the power profiling unit sends samples at a constant rate, not every single sample needs a timestamp, this can be interpolated later. Concurrently the already saved data is forwarded to the Gumstix.

Additional to the services, the DAQ has to implement a precise clock, which is tightly synchronized with other boards. Synchronization is achieved through a time pulse, in our case a pulse per second (PPS) from a GPS device. Serial logging is not affected by the DAQ.

The DAQ is an extension for the FlockLab, using the current signals and interfaces available. Hence most in- and outputs are predefined, as seen in Figure 3.1. This section lists the most important preconditions in terms of hardware for the DAQ: The serial peripheral interface (SPI) of the ADC unit (Section 3.1.1); the data rates that have to be handled (Section 3.1.2); the time accuracy needed (Section 3.1.3); and we define a set of commands to configure the DAQ (Section 3.1.4).

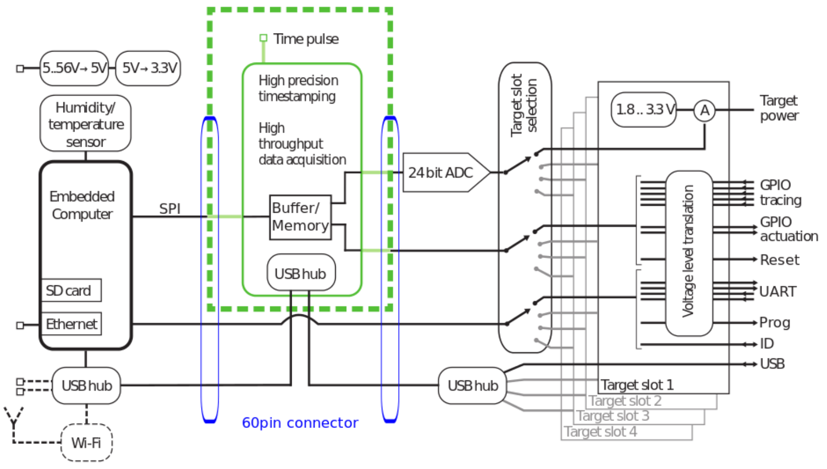


Figure 3.1: The new DAQ component integrated into the existing system

3.1.1 ADC SPI Interface (ADC to DAQ)

An serial peripheral interface (SPI), is used to transmit data between two devices. One device is the master on the interface, it defines the transmission rate, by providing the clock signal. The other device is in slave mode. Further the master device defines when a transmission is initiated or nothing is transmitted.

The analog digital converter (ADC) on the FlockBoard converts an analog power measurement into an digital sample. The clock signal (CLK) of the ADC unit is generated by a 14 MHz quartz. The CLK signal is further divided by 4 by a Flip-Flop to generate the SPI clock rate (SCLK). There are two sampling modes: a 28 kHz high-resolution mode or a 56 kHz high-speed mode, each with 24 bit samples, i.e. 672 kbit/s or 1344 kbit/s of data is generated. The resolution (high/low) is set by a jumper on the FlockBoard, and can not be changed in software. In the current setting, shown in Figure 3.2, the ADC unit sets ADC_nRDY shortly to low, when a new frame starts. This implies that the SPI interface on the DAQ needs to be in slave mode, i.e. the DAQ SPI clock rate, frame indication and data-in signals are all input signals. The ADC_nCS signal is used to turn the ADC on and off.

The waveforms of the SPI signals are shown in Figure 3.2, and the connections of the ADC signals to the Gumstix signals are shown in Table 3.1.

ADC pin name	Board name	remark
CLK	ADC_CLK	14 MHz
$\overline{\text{DRDY}}$	ADC_nRDY	former RTC_nINT
SCLK	SPI_SCLK	ADC_CLK $\cdot\frac{1}{4}$ or $\cdot\frac{1}{2}$
DOUT	ADC_DOUT	= SPI_MISO one clock cycle delayed
DIN	-	not used

Table 3.1: ADC SPI interface signals

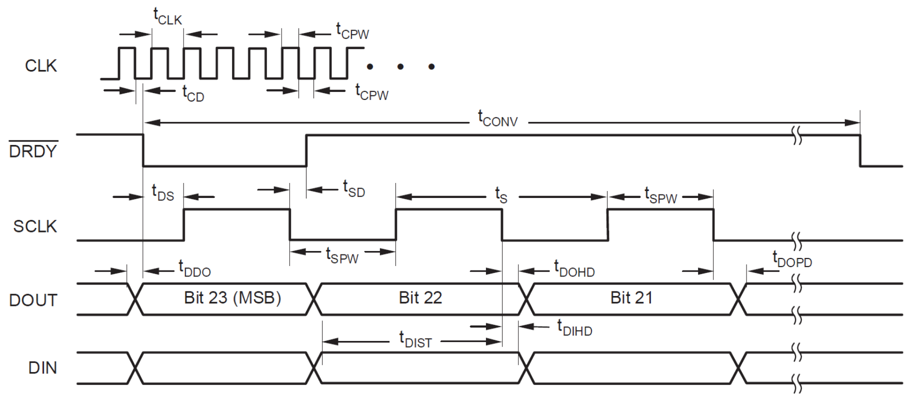


Figure 3.2: SPI timing characteristics of the ADS1271 from [6]

3.1.2 Data Rates

We want to support a high event detection rate with the DAQ. By attaching a timestamp to each event a lot of data traffic is generated. In this section we analyze the event rates triggered by the different target platforms. Hence we can derive the data rate the DAQ will need to handle.

Data is produced by two different services: power profiling and tracing. The ADC sends power profiling samples (24 bit) at a rate of 28 kHz or 56 kHz (e.g. 0.7 or 1.4 Mbit/s). Hence there will be at most a ADC sample every 17.9μs. The power profiling sample rate is independent of the target platform.

In terms of data traffic, generated by the tracing service, two rates are investigated: a long term continuous rate and a short term burst rate. We estimate that in the long run, up to 2 percent of all instructions executed on a target node set traced GPIO pins. Since a target node will execute some productive code and not just constantly toggle a pin. Further typical wireless sensor nodes (WSN) are battery powered and often switch in a power saving mode to extend the battery life. In power saving mode the clock is slowed down or no code is executed at all, hence it also decreases the event rate.

platform	continuous event rate	burst event rate
Tmote Sky	16kHz	0.8MHz
Tinynode	48kHz	2.4MHz
IRIS	80kHz	1.6MHz
Opal	384kHz	19.2MHz

Table 3.2: estimated GPIO event rates per platform, continuous and bursts

Table 3.2 shows the continuous and burst event rates expected at different target platforms. Time differences of two consecutive events range from 2.6μs on a Opal, up to 62.5μs on a Tmote Sky. By assume a event packet size of 32 bit, continuous data rates can reach from 512 kbit/s to 12 Mbit/s.

The DAQ should be able to trace bursts of events at much higher rates, generated for example when setting a sequence of pins. The maximal event rate a target can generate is dependent on the clock speed and cycles needed to execute a command to set a GPIO pin. The rates shown in Table 3.2 correspond to time differences of two events of $52ns$ up to $1250ns$, again the Opal is the fastest and the Tmote Sky the slowest target platform.

To properly detect a sequence of events triggered by the Opal platform, the DAQ at needs to oversample the Opals burst event rate of 19.2MHz.

3.1.3 Time Accuracy

To relate time critical events of multiple FlockBoards, the DAQ needs an highly accurate time. Wi-Fi connected observers have an average time error of $166\mu s$, with the currently used NTP protocol to synchronize the time. As mentioned before to monitor time critical behavior of network protocols, we need an accuracy in the sub-microseconds.

We propose to use a pulse per second (PPS) signal from a GPS receiver, as a a reference signal. A PPS signal can have as little jitter as $6.7ns$ (sampling deviation from [7]). And we measured pairwise time differences of different GPS receivers of below $40ns$. The DAQ will therefore adjust its internal time continuously to a PPS signal.

The internal time can be a relative time, for example relative to the start second of a test run. And the relative timestamp can later be converted into an absolute time by the Gumstix or server.

Further to utilize the accuracy of the internal time, the timestamps applied to events need sub-microsecond resolution, too. However, to get the order of a sequence of events is enough for tracing function calls. The timestamp therefore does not need to be as precise as the maximum event rate, as long as the events are detected and their order is determined.

3.1.4 Input Commands

We define a set of commands to control the DAQ, to set up a test run and configure the needed services. The commands include:

- reset the DAQ to an initial state [on,off]
- route signals directly to the FlockBoard [on,off]
- start and stop a test run [on,off]
- tracing service: specify pins to trace
- actuation service: specify pins and the time to set the pins
- power profiling service: start and stop
- power profiling service: specify the sample divider
- callback service: specify a tracing pin as callback pin

To be able to reset the whole DAQ including the interface which receives and handles the other commands, as reset signal a GPIO pin of the Gumstix is selected. The rest of the commands is received over a serial interface, when the

DAQ is not in reset state. Routing all signals directly through implies that no test is running, or at least without the DAQ.

Then before starting a test run some of the commands have to be configured: The pins to trace and the sampling divider of the power profiling service will not change during the test; The callback service, when turned on, does not change either.

The actuation service and the starting/stopping of the power profiling service can be combined, since the ADC unit is controlled by one single signal (*ADC_nCS*). Therefore the *ADC_nCS* signal can be handled as the other actuation signals. The actuation information (pin and time) will be sent to the DAQ shortly before the actuation event. The time needs to be specified to transmit and process actuation commands, depending on the final control interface of the DAQ. Hence the DAQ continuously receives actuation information and requires a little buffer to store them before their execution time.

The standard setting for tracing pins and setting pins are all inactive and all low respectively. Further the ADC needs to start/stop depending on a target output pin (ADC callback).

3.2 Consequences of the Requirements

Given the requirements from the previous section, we define further details of the DAQ. First in Section 3.2.1, we select a communication interface between the DAQ and the Gumstix, based on the expected data rates. In Section 3.2.2 we give a overview of data packets sent from the DAQ to the Gumstix. Then in Section 3.2.3 we compare micro processors and FPGAs as possible core unit for the DAQ.

3.2.1 Interfaces DAQ - Gumstix

Multiple Interfaces could be used as communication link between the DAQ and the Gumstix. As seen in the previous sections, the DAQ will have high data throughput of 12 Mbit/s continuously or even higher rate burst. Therefore interfaces like I2S, I2C, MSL and Multimedia Card are ruled out, because of slow transfer rates (I2S: 3 Mbit/s, I2C: 300 kbit/s) or because of missing pin outputs (MSL, SD/MultiMediaCard) on the 60 pin connector. Which left us with two feasible Interfaces: SPI and USB.

SPI

- Baud rate: 6.3 kbit/s - 13 Mbit/s
- sample size (frame): 4-32 bit
- Interrupt on specific FIFO buffer level
- FIFO: 16x 32bit (space for 16 samples)
- DMA possible

USB

- Bit rate: 12 Mbit/s
- shared with other services (serial input, WLAN)
- DMA possible and has 4 kbytes of endpoint memory

The SPI can transmit data at the highest rate and the Gumstix can process the data through DMA. The USB interface on the other hand can not guarantee the full bandwidth for transmitting data,. Since USB is a shared bus and other services, as serial logging, use it to. Therefore to connect USB to the DAQ a hub is needed to provide the existing link from Gumstix to the FlockBoard and a new link from Gumstix to the DAQ.

We will use the two interfaces in a combined manner. SPI is used to transfer continuously data from the DAQ to the Gumstix, and USB for commands from Gumstix to DAQ. Hence the SPI handles the high data rates and provides the full bandwidth, and the USB transmits the sporadic commands, therefore a shared bus is not an issue. Further USB can be used as debug connection for further components on the DAQ, without influencing the detection rate.

When we compare the throughput rate of the SPI (12 Mbit/s), with the event rates from Section 3.1.2, we see the SPI is too slow to handle bursts of events. Therefore we need to buffer the burst of events. We solve this with a fast memory block on the DAQ, e.g. SRAM, where the events and their timestamps are stored before transmitted over SPI to the Gumstix.

3.2.2 Output Packets

The DAQ sends continuously the acquired data in packets to the Gumstix. To reduce data traffic, the packet size should be as small as possible. To simplify packet decoding, the following information have to fit into the same packet size each:

- tracing service: level of the 5 signals, plus a timestamp
- actuation service: level of the 3 signals, plus a timestamp
- power profiling service: timestamp for the first, last and every 100th ADC sample
- power profiling service: ADC sample (24 bit)

The actuation of signals are traced and sent back to the Gumstix, like the other tracing signals. Hence the 8 signal levels can be handled and represented in the same packet. Every time one of the 8 signals changes all 8 levels of the signals are captured and timestamped as one tracing packet. If multiple signals change at the same time, only one tracing packet is generated.

The power profiling service on the DAQ produces two kinds of packets: One containing the ADC samples and the other containing a timestamp after every 100th ADC sample. The ADC unit has a constant sampling rate, timestamps of a single ADC sample can therefore be interpolated later on the server. This saves roughly half the bandwidth compared to a scenario sending a timestamp packet along with every ADC sample packet.

Most of the packets contain a timestamp. Therefore the larger a timestamp is the more data throughput is generated. For example if the timestamp resolution is $500ns$ and a test runs no longer than a day, then 38 bits are needed to represent all the $500ns$ timeslots of the day. Hence every packet would contain a 28 bit timestamp. To reduce the payload of the packets we introduce a smaller time frame combined with an overflow counter. A test runs consists of multiple time frames, distinguished by the overflow counter. At the beginning of every time frame a packet with the overflow counter is generated. Then each timestamp of an event would represent a part of a time frame, and can be related to the last generated overflow packet.

We will set the time frame to one second, because it simplifies computation and corresponds to the reference PPS signal. Using 21 bit for the timestamp, allows us to implement a time resolution of $500ns$. Without changing the size a timestamp could be further improved, by selecting a smaller time frame.

The timestamp size of $21bit$ is a tradeoff between packet size and time precision. The packet size with a $21bit$ timeslot, 8 bit for pin levels and 3 more bits for a header, is therefore 32 bit.

3.2.3 Core Unit

In this section we discuss the selection of a suitable processing unit. The processing unit has to handle different task concurrently, as tracing high event rates and adjusting the internal time. Hence high parallelism is asked. For accurate timestamps deterministic delays between event detection and event timestamping is needed. The DAQ will be deployed in a later step on 30 FlockBoards, therefore the cost of a single processing unit matters.

We compare two different device families: Micro controllers (MCU), and field programmable gate arrays (FPGA).

Micro controller MCUs process their task sequentially, therefore interrupts are used to detect events. The interrupts are then processed later, depending on the load of the MCU and the occurrence of other interrupts, for example of another tracing signal. Interrupts therefore add delays between the event and the timestamping of the event. This is similar to the current implementation with a CPU on the Gumstix, which has non deterministic delays and limits the maximal detection rate, as mentioned before.

Most MCUs come with a set of standard interfaces, which can reduce the workload of the core. For example SPI or UART, and also USB is simpler to connect than to a FPGA. Those interfaces are configurable on a MCU with little effort. The price range of a MCU is around 5$ to 20$ [1], which can be much cheaper than a FPGA, where prices start at 10$ but can rise up to 150$ [2]. And if little space is available on the final implementation board, a MCU is the better choice.

FPGA The main advantage of a FPGA, with the most effect on the DAQ, is the capability of executing different tasks in parallel. For example time calibra-

[1]For example the STM32F2xx or LPC182x

[2]For example the Spartan-6 LX4 to LX150

tion can be implemented without influencing the detection rate of events. At the same time ADC samples can be received over SPI and pins can be actuated.

An FPGA can guarantee the detection of an event in every clock cycle, by implementing a small interface just to detect an event and buffering the event before further processing. This ensures the feasibility of an exact clock cycle count between two PPS signals, needed to calibrate the internal time. Additionally a delay between detection and timestamping of an event would be deterministic.

While interfaces as SPI or UART have to be implemented from scratch (or imported from an other source), they are therefore highly customizable and not as inflexible as MCU interfaces. And USB can be converted on an external chip into an UART device, accessible from the FPGA, depending on the FPGA the same connection could also be used to program the FPGA. But an external chip means additional hardware and costs.

FPGA: later multiple devices: price relevant. Space requirements difficult to estimate, without experience. Therefor a large FPGA is needed to ensure space. But can later be transferred to smallest fitting FPGA of family.

Space requirements of a FPGA difficult to estimate, whiteout experience. Therefore a large enough FPGA has to be selected, to ensure the number of logic elements meets the requirements of the final code. A large FPGA implies high cost ($>$ 50$)[3], and the cost difference to a MCU is multiplies by 30, when later deployed on all FlockBoards.

However most of the device families have a broad range of chip sizes. For example an average Spartan-6 chip from Xilinx with 43600 logic cells (LX45) costs 60$, but the smallest Spartan-6 FPGA chip with 3800 logic cells (LX4) only costs 10$. Therefore a system like the DAQ can be developed on a more expensive chip, but the final implementation then transferred to the smallest (and cheapest) FPGA chip of the same family, which meets the requirements of the implementation in I/O ports and Logic elements needed.

And finally a FPGA can implement a small MCU if need be.

Conclusion To meet the requirements to detect high event rates of up to 19.2 MHz and simultaneously applying accurate timestaps, we decided to implement the DAQ with a FPGA. Although the implementation might also work on a MCU, it adds a complexity, due to uncertainties in delays and interrupt handling, as well as processing virtually simultaneously events sequentially. Therefore an implementation on a FPGA is simpler, and certain detection rates can be guaranteed.

[3]For example the Spartan-6 LX45

Chapter 4

Implementation

To develop a prototype of the Data Acquisition unit (DAQ) we choose an FPGA developer board from Digilent. The Anvyl Developer Board features as main component a Spartan-6 LX45 FPGA. The LX45 has 218 user I/O pins and 43661 logic cells, which is most likely over sized, but therefor prevents us from running out of space/options. In a later step (Section 5.5) we explore which is the smallest FPGA of the Spartan-6 family capable of implementing our code/design.

Other components of the Anvyl board used to implement the DAQ are a 100 MHz oscillator, which is used as clock source; 2 MByte of SRAM, where 1 MByte is used to buffer the data flow; and a USB interface, which is needed to program the FPGA and connect the DAQ through UART to a PC.

As reset signal (DAQ_RESET), one of the '0-1'-switches on the Anvyl board is used. During the implementation a lot of the available switches, LEDs and user I/O pins were used to debug and evaluate the code on the FPGA.

In Table A.4 in the Appendix a range of boards is compared. We choose the Anvyl Developer Board, because of the existing SRAM and DDR RAM, as two possible buffers. As well as the availability and reasonable cost of the smaller Spartan-6 chips.

And as reference time, to calibrate and synchronize time over multiple FlockBoards, a pulse per second (PPS) signal from a u-blox LEA-6T GPS device is connected. The 'T'-line of the u-blox GPS devices supports precision GPS timing.

In the development setup, as seen in the overview in Figure 4.1, the developer board is placed between the Gumstix PC and the FlockBoard and captures 12 of the 60 GPIO pins connecting the two devices. The rest of the pins are connected regularly and are not affected by the DAQ.

On the FlockBoard side 4 of those pins are used to turn the ADC unit on and off (*adc_off*) and to receive the ADC samples per SPI protocol (*clk*, *nfrm* and *rx*). The next 5 pins carry the tracing events generated by a target node. And the remaining 3 pins are actuation signals to generate input signals for the target node.

On the Gumstix side mainly the 4 SPI pins are used. This interface transmits all the data collected and timestamped by the DAQ, from the DAQ to the Gumstix, which then forwards the data to a PC over Ethernet. Three of these signals implement the SPI interface (*clk*, *nfrm* and *rx*) and one is set by the

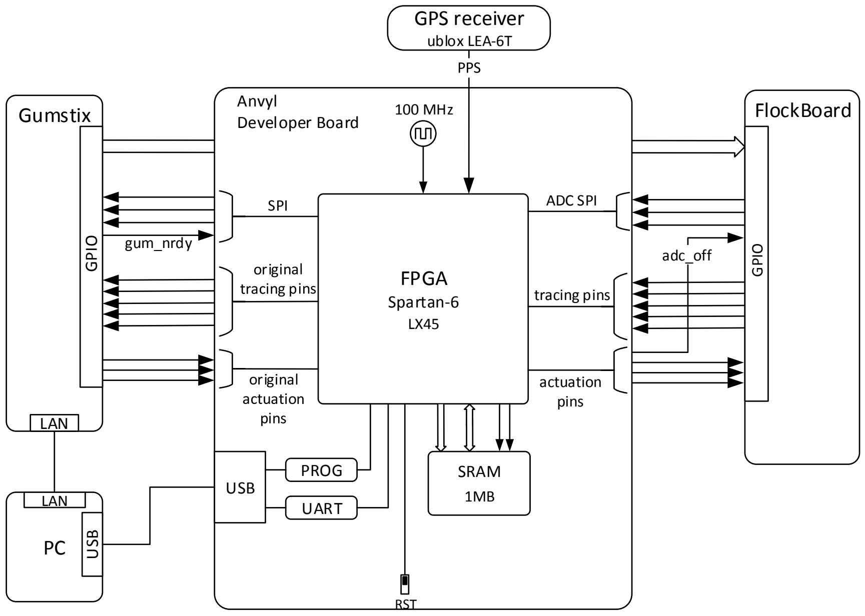


Figure 4.1: DAQ develop setup Overview

Gumstix when it is ready to receive data (*gum_ nrdy*).

The other 8 captured pins between the Gumstix and the DAQ are the original tracing and actuation pins. The DAQ is implemented in such a way, that (trough UART command) all captured signals can be routed trough directly (direct trough mode). In that mode the Gumstix and the FlockBoard can communicate with each other as if there is no DAQ in their middle.

The DAQ and its services can be configured by the PC through the USB-UART connection. The configuration options include for example, which pins are to trace, when which pins to actuate and as mentioned before, putting the DAQ in direct trough mode.

The idea of the final and integrated DAQ is to control the DAQ from the Gumstix, without the need of a PC or manual switches. Which means that the DAQ_RST signal would be another GPIO pin and the USB-PROG and USB-UART Interfaces are connected direct to the Gumstix. This can be solved by using the already existing USB signals between the Gumstix and the Flock-Board, intercept those and put a USB hub on the DAQ.

4.1 FPGA/VHDL modules

In the previous section the outer signals to/from the FPGA are described. This section shows more detailed software modules and their functions inside the FPGA. All the code for the FPGA is written in VHDL and some common blocks, as for example the FIFOs, are generated code by the software suite. For Coding, Syntax checking, simulating, compiling and programming the FPGA the Xilinx ISE 14.7 suite was used. It combines the complete tool set needed to develop on a FPGA in one software suite.

The basic FPGA Layout is shown in Figure 4.2. An internal time representation is calibrated, with an external pulse per second (PPS) signal. A tracing module detects a signal change as event, and generates a packet with the signal levels and a timestap. A power profiling module receives ADC samples, generates packets with the samples and periodically generates timestamp packets. The tracing and power profiling module each put the generated packets in a FIFO, which acts as a fast buffer. The two packet streams are merged in a SRAM memory module, this is a larger and still fast buffer, but slower than a FPGA FIFO. Through another FIFO the SRAM module than forwards the packets to a SPI interface, which sends the data packets to the Gumstix. The main purpose of the FIFOs and SRAM module is to capture burst of events, i.e. packets, which the SPI interface can not transmit fast enough. The mentioned modules are explained in more detail through the Sections 4.1.1 to 4.1.5.

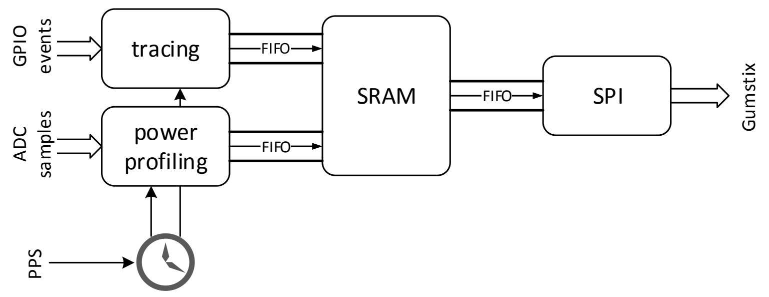


Figure 4.2: Dataflow

Then in Section 4.1.6 the module setting actuations and managing the callback is described. Finally in Section 4.1.7 follows the Control module. Essential to configure and run tests with the DAQ integrated into the FlockBoard.

4.1.1 Time Calibration

An important input for the tracing and ADC packaging modules is an accurate internal time, needed for proper timestamping. The internal time as timestamp in form of a seconds counter and a timeslot counter is provided by the Time Calibration module (Figure 4.3).

The seconds counter is 17 bit wide and counts from 0 to 86400 (one day in seconds). It starts to count the seconds passed in a test run when the test starts, therefore the internal time represents a relative time in respect to the absolute start time of the test run. The timeslot counter is 21 bit wide and allows a sub-second resolution of $500ns$. The internal time representation is computed on the basis of the 100MHz oscillator and a pulse per second (PPS) signal.

In the following paragraph we discern two units, a clock cycle of the oscillator of $10ns$, with which most of the FPGA is clocked, and a timeslot of $500ns$ to represent the sub-seconds of the internal time.

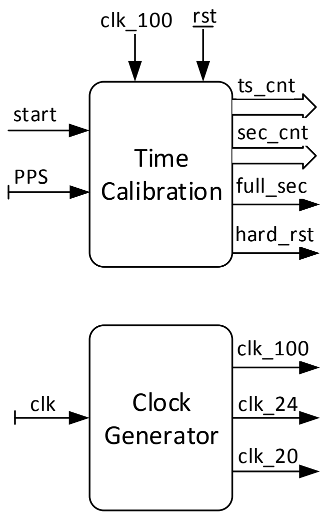


Figure 4.3: Time Calibration module

The Time Calibration module has multiple counters inside. The timeslot counter is increased every 50 cycles (i.e. $500ns$), therefore a cycle counter with overflow at 50 is used. When the timeslot counter reaches $2M$ (i.e. $1s$), a second is full and $full\_sec$ is emitted. Further the seconds counter is increased and the timeslot counter starts with 0 again.

Since the oscillator is not ideal one second has not exactly 100M cycles. Hence the module has to add or omit single cycles in the cycle counter. To measure the cycle difference of the oscillator to an ideal 100MHz signal, the highly accurate pulse per second (PPS) signal from a u-blox LEA-6T GPS receiver with a standard deviation of $6.7ns$ [7] is used. The module takes the average cycle difference over the last 8 seconds to estimate the cycle difference of the new period.

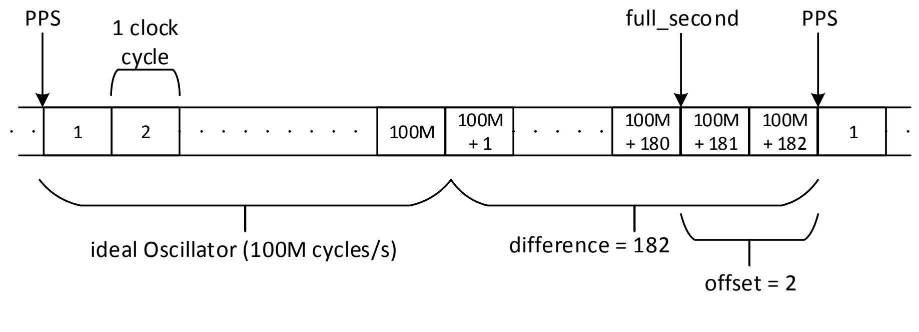


Figure 4.4: Cycle offset and cycle difference

Additionally to the cycle difference the module has to compensate an offset too. At every PPS signal the cycle offset between the PPS signal and the estimated $full\_sec$ is measured. The offset error is a combination of the oscillator clock drift, e.g. due to temperature change, and a sampling error. In Figure 4.4 the cycle difference and cycle offset is illustrated. Finally the module has to compensate during every second the cycle offset and the average cycle difference. Therefore the module omits or adds up to one cycle to the cycle counter per timeslot. Those compensations are evenly spread over all timeslots of a second. By spreading the compensations the error of single timeslots are minimized, and the timestamp does not jump.

For example every $100'000'182$ cycles a PPS signal is measured. The average cycle difference is accordingly 182 cycles. One second has $2M$ timeslots, therefore every $\frac{2M}{182} = 11k$ timeslots the cycle counter overflows at 51 instead of 50. Hence within a second the difference is compensated.

To make the module more robust, a new cycle difference is calculated, even when a PPS signal should be lost, due to bad GPS reception for example. The module counts the cycles between two PPS signals and detects if up to 20 PPS signals are missing and computes an average cycle difference accordingly. To count for more than 19 seconds clock cycles, would exceed the 32 bit cycle counter.

The Time Calibration module, shown in Figure 4.3, has 4 outputs, the 3 regular ones are: the timeslot counter to represent the current sub-second, the seconds counter, which counts the seconds since the start of the test run; and a $full\_sec$ signal is emitted every time the timeslot counter overflows (starts at 0 again), it represents the estimated (full) second.

The $4th$ signal is an error signal: $hard\_rst$. It is emitted when the cycle offset between the PPS signal and the estimated $full\_sec$ is to large to be compensated in one single second. The threshold is $2M$, due to the modules maximum compensation of one cycle per timeslot. When the threshold is exceeded the sub-second is at least 2% off. In case of a $hard\_rst$ the timestamp counter is set back to 0, to align the counter with the PPS signal. By outputting the $hard\_rst$ signal other modules, as for example the Tracing module, are informed that the time representation made a significant jump.

The input signal *start* tells the Time Calibration module a test run is started and the second counter starts to count too. Therefore the time representation is always relative to the test runs absolute starting time. The rest of the module runs already before the *start* signal is applied, to calibrate the internal clock to the PPS signal.

The DAQ_RESET signal totally stops the whole module and puts the outputs to a default initial value. This reset behavior is the same for all modules.

4.1.2 Tracing

The Tracing module (see Figure 4.5) basically checks every cycle if at least one of the traced pins has changed, i.e. an event occurred. If so it puts all 8 pin levels and the current timeslot counter in a packet and writes it to the tracing FIFO. To relate tracing packets to the correct second, at every $full\_sec$ event the current seconds counter is packed instead of the timeslot counter (which would be 0 by definition). Because the tracing packets will be transmitted to the Gumstix strictly chronologically, every event in a timeslot packet can be related to the previous sent full-second packet.

The module is able to detect an event, compute a packet and write it to the tracing FIFO every cycle. This means there could be more than one event during the same timeslot. Maximal, if an event occurs every $10ns$, up to 50 events can be detected and packaged. Therefore when multiple packets have the same timestamp, it is not possible to tell when during the $500ns$-timeslot they happened, but in which order they did.

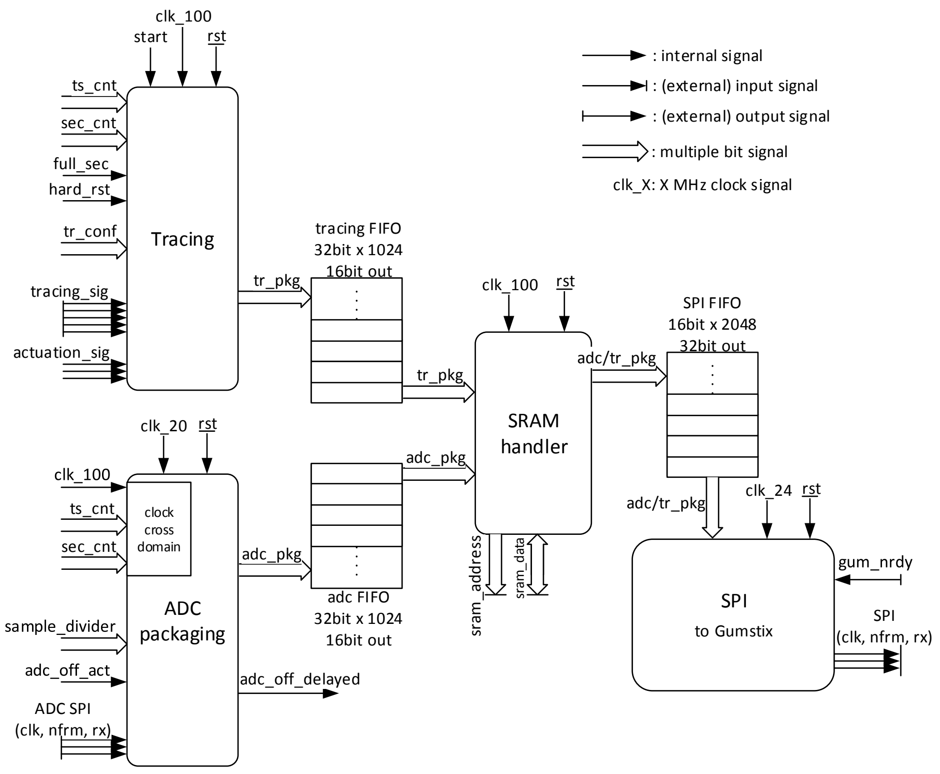


Figure 4.5: Data Flow from the tracing signals and adc samples to the SPI-Gumstix interface in detail.

In the header of a tracing packet the payload is specified: it can be distinguished between a regular tracing packet containing pin levels and timeslot and a full second packet containing pin levels and a full seconds count. Further the header specifies following events: if a *hard_rst* occurred and if the FIFO was full.

When the tracing FIFO is full and the module tries to write, the packet is lost. The header of the next successfully written packet will then specify $fifo\_full$. This means when the Gumstix receives a $fifo\_full$ header, there could be lost any number of packets between the $fifo\_full$ packet and the last regular tracing packet. However packets with a regular header, following a $fifo\_full$ packet, can then be properly interpreted again. If the last $full\_sec$

packet was lost too, the timestamp can be related to the next $full\_sec$ packet. A problem remains if there are multiple $fifo\_full$ periods and multiple $full\_sec$ packet were lost. Then it is not possible to recover, in which $fifo\_full$ periods the $full\_sec$ packets were lost. And just the regular packets before or after all $fifo\_full$ periods can be related correctly to a second.

More details to the packet definitions can be found in Section 4.2.

4.1.3 ADC packaging

The ADC packaging module (see Figure 4.5) receives ADC samples from the ADC unit on the FlockBoard per SPI and puts them into an ADC packet. Further, the module generates packets with timestamps to relate the ADC samples to a time.

When the ADC unit on the FlockBoard is turned on, it sends the measured samples as 24 bit frames over SPI, as SPI master. The ADC packaging module implements an SPI slave interface (based on [8] [1]) to receive the ADC samples, shown in Figure 4.5. The ADC packaging module runs with a 20MHz clock, and over samples the 7MHz clock.

The received ADC samples are put into 32 bit packets with a header and written into the ADC FIFO. Before the first sample a packet with the current timeslot and a packet with the current seconds count are written to the FIFO. This is used as start time of the first sample. Then after every 100th sample the current timeslot count is sent in a packet again. That way we are able to interpolate a timestamp for all the ADC sample packets sent in between two ADC timeslot packets. Note that the ADC has a constant sampling rate. The timestamp always corresponds to the transmission start time of a sample, i.e. when the SPI signal $nfrm$ is set to low.

To reduce the data sent to the Gumstix a test user can specify which part of the samples is actually forwarded in ADC packets. Configured by the input *sample_divider* as a 11 bit wide integer. For example if *sample_divider* $= 1$ every sample is forwarded, if *sample_divider* $= 4$ every forth sample is forwarded and the other three discarded.

A last feature of the ADC packaging module, the signal to start/stop the ADC unit on the FlockBoard runs through the module. It does not affect the signal starting the ADC unit, but delays the stopping to ensure the last ADC sample is fully received and not cut off.

More details to the exact packet definitions can be found in Section 4.2.

4.1.4 SRAM and FIFOs

As seen in Figure 4.5 the Tracing and ADC packaging modules put their captured data as 32 bit packets in the correspondent FIFO. Those FIFOs are 1024 slots deep and 32 bit wide, and they have a read and write access time of one cycle per packet. The FIFOs are therefore really fast buffers. The next stage is the 1 MByte SRAM: It can store up to $250'000$ 32 bit packets. The SRAM is larger than the FIFO but slower in access time: 16 cycles are needed to write or read from the SRAM.

[1] We based our SPI slave implementation on the examples in [8]. The examples, written in Verilog, were translated into VHDL and adapted to our system.

Those two buffers (FIFO fast/small and SRAM slower/large) are used to relieve the last module, the SPI-Gumstix interface, which sends all the captured data to the Gumstix and which is, with 12 Mbit/s or $353k$ packets/sec respectively, the slowest module in the data flow path.

The two modules ADC packaging and SPI to Gumstix run with a different clock speed than the rest of the FPGA. Therefore we have to ensure a proper clock domain crossing of data between modules with different clocks, e.g. the SRAM and SPI module. For this purpose the FIFOs are used. All the FIFOs are generated with the ISE software suite and can have a two clock interface, therefore data can be written and read with a different clock signal and speed each.

The preference to read or write to the SRAM unit are as follows: In terms of writing the ADC FIFO is preferred over the tracing FIFO, since the tracing module could generate much higher throughput than the ADC packaging module. And the constant maximum throughput of the ADC packaging module still leaves spare time to handle the tracing FIFO between two ADC sample packets. For example, at the maximum setting the ADC produces 56000 samples/sec, which is roughly $\frac{1}{5}$ of the SPI bandwidth or 1% of the SRAM write bandwidth.

If there is no data to write or the SRAM unit is full, the SRAM module reads data and puts it into the SPI FIFO, unless the SPI FIFO is full, too.

In the worst case the Tracing and ADC packaging modules generate so much throughput over a certain time period, that the SPI module is not able to transmit at the same rate, and therefore the SRAM unit and FIFOs exceed their capacity one after each other. Note that the threshold of the continuous transmission rate is higher than in the FlockLab without the DAQ. Before events could be detected with a rate of up to 50k events/sec, which determined also the minimal time difference, of $20\mu s$, between two events to be detected. With the implementation of the DAQ a continuous rate of up to 350k events/sec can be reached, furthermore due to the buffer capabilities the minimal time difference could be decreased to $10ns$ (i.e. a 100MHz rate).

The SRAM module reads the data in the same order as it has written them (first in first out). It is ensured that, after leaving the SRAM, the tracing packets are still ordered chronologically and that ADC packets are still ordered chronologically. The two packets streams however are not merged in any particular order. But all the packets can be distinguished by their header.

4.1.5 Gumstix SPI

The SPI to Gumstix module implements an SPI master interface with a 12 MHz SPI clock signal (*spi_clk*), specified to communicate with the SPI setup of the Gumstix. The characteristics are given in Figure 4.6, where the FPGA signals *spi_clk* and *spi_tx* correspond to SSPSCLK and SSPRXD, respectively. Further spi_nfrm is the inversion of the signal SSPSFRM. The SPI interface transmits 32 bit data frames. The spi_nfrm signal is set to 'low' for one cycle, one cycle before the first data bit is sent. The bits are valid to read on the falling edge. The module runs with a 24 MHz clock, i.e. the double *spi_clk* speed.

When the Gumstix indicates to be ready by the *gum_nrdy* signal and if there is data in the SPI FIFO, the module transmits the 32 bit data packet. With a 12 MHz SPI clock, the maximum throughput of this interface is 12 Mbaud/s or 353k packets per second.

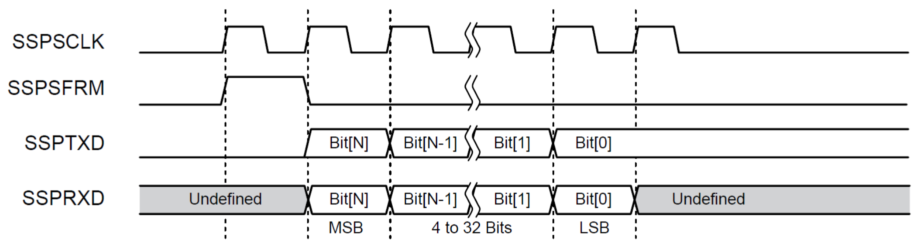


Figure 4.6: SPI timing characteristics of Gumstix from [9].

4.1.6 Actuation and ADC-Callback

The main task of the Actuation module is to actuate the 3 target signals (*tg_sig1, tg_sig2, tg_rst*) at user specified times. Further it has to start/stop the ADC unit with the *adc_off_act* signal, this signal is handled like the other actuation signals. To actuate a signal a packet from the actuation FIFO is read. Such a packet contains the pin(s), pin level(s) and time (seconds count and timeslot) to actuate. In each cycle the Actuation module compares the current internal time with the time from the packet. If the times match the corresponding pin(s) are actuated, and the packet removed from the FIFO. If a timestamp of a packet is more than 6 hours away, the packet is discarded. It is interpreted as a wrongly placed packet.

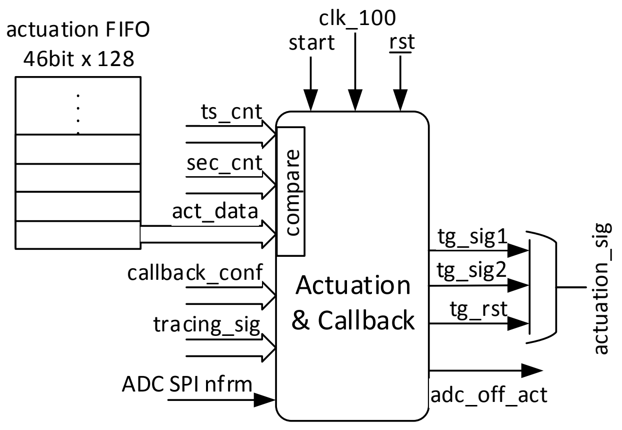


Figure 4.7: Actuation and Callback module

A second task of the module is the callback service. When the callback service is configured, a target node can start/stop the ADC unit by itself. Therefore the module listens to a specified tracing pin and sets the *adc_off_act* accordingly. The ADC unit is started immediately when the traced pin goes low, but the module waits, when the traced pin goes high, for a pre-configured number of

ADC samples before it turns the ADC unit off. By delaying the stop signal it is ensured that the ADC samples, sampled at the turn off time, arrive at the DAQ before the ADC shuts down.

The *adc_off_act* stays low (ADC unit on), if either the callback service or the actuation service started the ADC unit. Both need to stop the ADC unit before the ADC unit actually stops.

4.1.7 Controlling

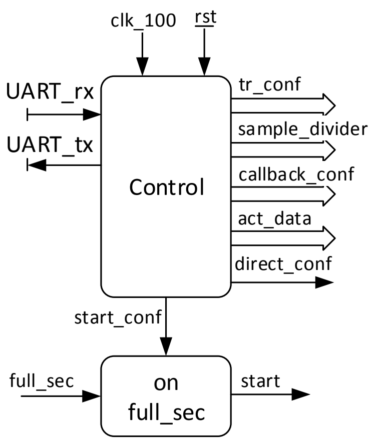


Figure 4.8: Control Interface module

The last but no less important module is the Control module. Over this module all the services implemented in the other modules can be configured.

The Control module implements a UART interface with a baud rate of $1M$ and a frame size of 8 bit, plus one stop and one start bit. The UART interface is an imported VHDL core from [10][2].

The Control module receives the configuration commands over UART. An acknowledgement is sent back over UART if a complete command is successfully received. The following commands and their payload can be sent:

- **conf_tracing:** specifies the tracing and actuation signals to be traced.
- **conf_actuation:** specifies pin and level to actuate, including a timestamp.
- **conf_callback:** specifies the callback pin or none; and the packet delay.
- **conf_sampling_divider:** specifies how many ADC samples to skip/send.
- **conf_direct_through:** routing all Gumstix signals direct trough to the FlockBoard or not.

[2]The main file 'uart.vhd' of [10] is deployed unchanged, the handling files 'loopback.vhd' and 'top.vhd' were integrated and adapted into the files 'loopbackplus.vhd' and 'controlling.vhd', respectively.

- **conf_start:** start and stop a test run.

To run a test with the DAQ the option conf_direct_through needs to be 'off'. All other configuration commands, except actuations, need to be set before the start of a test run. Actuation commands need to be sent chronologically, because they are processed one after each other. The actuation FIFO has 128 slots, therefore up to 128 conf_actuation commands can be sent prior to the actual actuations. The UART interface needs $70\mu s$ to transmit a full actuation command, hence an actuation needs to be sent at least those $70\mu s$ before the actual actuation.

By sending the command conf_start 'on', the test run is started on the next full second.

4.2 Packet definition

This section gives an overview over all the packets and their bit definitions sent over the two interfaces SPI to Gumstix and USB-UART.

4.2.1 SPI packet definitions

There are two basic packet types received over SPI: tracing packets and ADC packets. The first 3 bits serve as header: all ADC packets start with three ones as header, the other 7 header configurations specify different tracing events and/or different packets. As shown in Table 4.1 and Figure 4.9.(a-b), a regular tracing packet has as payload consisting of the pin levels and the timeslot of an event. And a full second packet contains the seconds count instead of the timeslot. Further the header encodes if certain errors happened on the DAQ: if the Time Calibration module had to hard reset the timeslot counter; if the tracing FIFO was full and events might be missing; or a combination there off occurred.

The first 3 bits (header) of an ADC packet are ones, the next 3 bits (sub header) specify the payload: a 21 bit timeslot counter; a 17 bit seconds counter; or an 24 bit sample, see Figure 4.9.(c-e) and Table 4.1. Further the sub header specifies whether the ADC time packet is the first, last or an intermediate packet in the stream of ADC sample packets.

Tracing packets			ADC packets		
Header	Type	Payload	Header	Type	Payload
000	regular tracing	a	111 000	first sample	c
001	full second	b	111 001	last sample	c
010	hard_rst	a	111 010	interval time	c
011	hard_rst + full second	a	111 011	second counter	d
100	fifo_was_full	a	111 100	sample	e
101	fifo_was_full + full second	a	111 101	-	-
110	fifo_was_full +hard_rst	a	111 110	-	-
111	ADC packets	-	111 111	Gumstix driver error	

Table 4.1: SPI packet headers and the corresponding payloads layouts, further specified in Figure 4.9

An exception is one header, specifying neither an ADC nor a tracing packet: When the SPI driver on the Gumstix no data has to return to a reading process, an empty packet is returned with 6 ones as header.

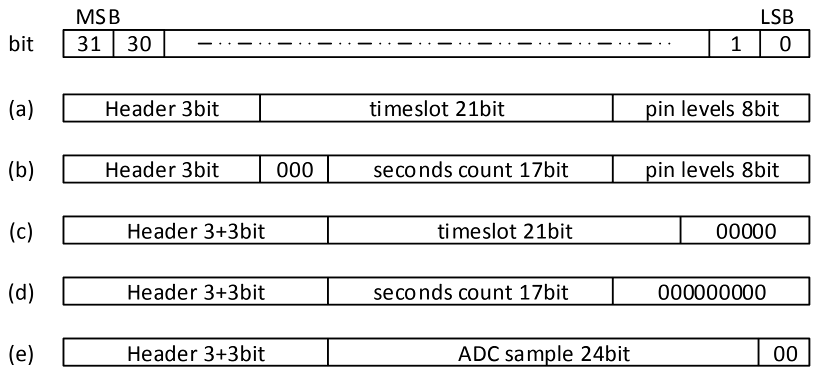


Figure 4.9: SPI packet layout

4.2.2 UART packet definitions

A configuration command can consist out of multiple 8 bit UART packets. The first of those packets carries the header and defines the payload, i.e. if more packets are needed to complete the command. The MSB of a header packet is always 0, the MSB of a plain data packet always 1. This ensures that after an incomplete command the next correct command will be detected. In the header 3 more bit are used to specify the payload. The actual remaining payload size in a header packet is 4 bit, in a data packet 7 bit. Table 4.2 shows which header is used for which command and which payload layout from Figure 4.10 a command has. The payload of the command packets is specified in more detail in the Appendix in Table A.2.

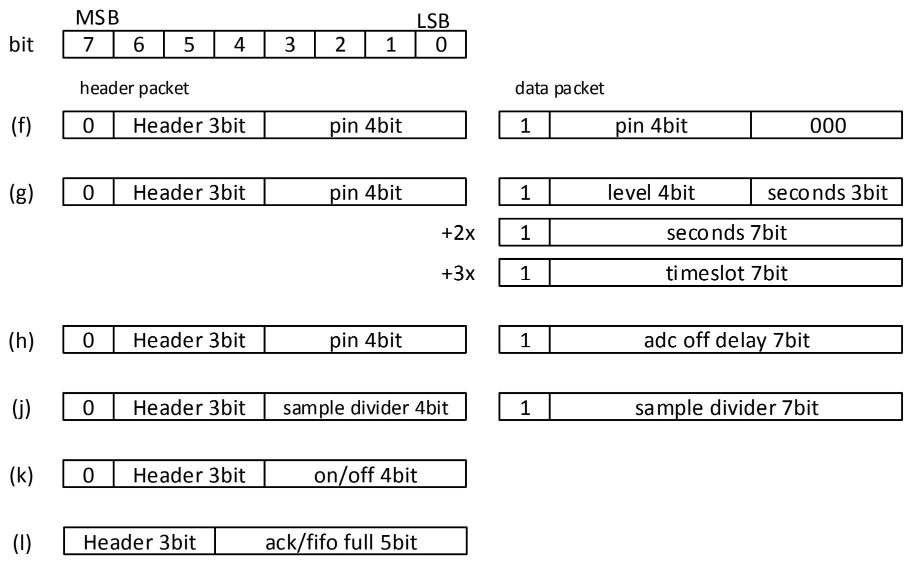
An acknowledgement is an 8 bit packet with a 3 bit header. The acknowledgement header contains the same header bits as the successfully received command and the remaining 5 bits are zeros. Further when an actuation command is successfully received, but the actuation FIFO is full, the remaining 5 bits are set to ones, and the command is discarded. When an error occurs all 8 bits are ones.

4.3 Adapted Gumstix Software (ads1271 driver)

The SPI driver on the Gumstix was originally not designed for high data rates. Therefore packets get lost, when the DAQ sends continuously data at the full rate. We modified the driver to process higher data rates more stable. To handle the data a DMA transfer, from the SPI interface to two memory buffers, was initialized. Every time a buffer was full and the DMA stopped, the kernel module set up a new DMA transfer to the other buffer and an users process

UART packets		
Header	Command	Payload
000	direct through	k
001	tracing	f
010	actuation	g
011	sample divider	j
100	start	k
101	callback	h
110	-	
111	ack error	l

Table 4.2: UART packet headers and the corresponding payloads layouts, further specified in Figure 4.10



MSB
LSB
bit
7 6 5 4 3 2 1 0
header packet
data packet
(f) 0 Header 3bit pin 4bit | 1 pin 4bit 000
(g) 0 Header 3bit pin 4bit | 1 level 4bit seconds 3bit
+2x 1 seconds 7bit
+3x 1 timeslot 7bit
(h) 0 Header 3bit pin 4bit | 1 adc off delay 7bit
(j) 0 Header 3bit sample divider 4bit | 1 sample divider 7bit
(k) 0 Header 3bit on/off 4bit
(l) Header 3bit ack/fifo full 5bit

Figure 4.10: UART payload definition

could read out the previous full buffer. One throughput limiting problem was the time needed to swap those two buffers, which was done by the kernel module.

Our solution set the DMA with a linked descriptor list up, so the DMA would swap the buffers and initiate a new DMA by itself. The descriptor list contains just two elements pointing to one another as next element. Such a list element contains source and destination (buffer) address, configuration flags and the address of the next list element.

With the new kernel module the next problem came up. The process reading out the data from the memory is to slow on the highest possible throughput. The data actually is received properly by the Gumstix and written into the memory buffers without losing any packets. But then the buffers were swapped faster than they could be read. Note that forwarding the data over Ethernet is faster than writing it locally on a SD card, see Table A.1 in the Appendix for SD card speeds.

With all the optimization receiving the SPI data works best by slowing down the SPI interface on the DAQ by $150ns$ per packet transmission, as further explained in Section 5.2.

In either case all the data produced by the DAQ is successfully sent over SPI to the Gumstix and written into a memory buffer. The bottleneck then is the reading process on the Gumstix.

4.4 Setup the DAQ Prototype

This section shows step by step how to run a test including the DAQ with the development environment and a single FlockBoard. The test is executed from a Linux PC, which initializes the Gumstix through ssh and configures the FPGA through a USB to UART connection (see Figure 4.1). The test data received by the Gumstix is forwarded per ssh to the PC.

Its assumed the Gumstix is turned on and connected to the FlockLab intranet, as well as the modified ads1271 driver is installed. The Anvyl developer board is programmed with the DAQ code (v1.3) and DAQ_RESET is on high. Further the developer board needs to be connected to the tracing, actuation and SPI signals of the Gumstix and FlockBoard, as shown in the Table A.3 in the Appendix.

A test run is carried out with the following steps:

1. DAQ_RESET is on high, which implies that the signals are routed directly through (*direct through* on) and no test is running (*start* is off). The FIFOs and the SRAM is cleared and all modules set their outputs to the initial state. In this step the target node can be configured.

2. DAQ_RESET is turned low, *direct through* and *start* remain unchanged. At this point the Time Calibration module starts running, the internal time is calibrated to the PPS signal. The Control module is turned on and ready to receive commands for the next steps.

3. Send the UART command to turn *direct through* off (*start* remains unchanged). Now the FIFOs, SRAM and the SPI to Gumstix are turned on and could forward data packets to the Gumstix. But the modules Tracing, ADC packaging, Actuation and Callback are still turned off and no packets are produced.

4. In this state send further UART commands to configure the services: tracing, actuations of up to 128 signals, callback and sample divider.

5. Now the Gumstix needs to be configured to receive packets from the DAQ. Therefore the driver for the SPI is initialized:

   ```
   $ ssh root@flocklab-observerXXX modprobe ads1271
   ```

   This command can only be executed when no other module is accessing the ads1271 device. Dependent modules can be removed with "modprobe -r *module_name*".

6. Then the SPI device is turned on and received packets forwarded to the local PC[3]

   ```
   $ ssh -F ~/.ssh/config root@flocklab-observerXXX
     dd bs=8188 if=/dev/ads1271-2 > some.file
   ```

   Replace 'XXX' with the correct observer number (e.g. 029) and specify the file to save the received data (some.file).

7. By sending the UART command '*start* on' the test is started. Now all the modules are running: as configured signals are traced, ADC samples received, actuations set. The produced data packets are sent over the Gumstix SPI through ssh to the PC.

8. The test can be stopped by sending the UART command '*start* off'. The FIFOs, SRAM and SPI to Gumstix are still running, hence the remaining of the acquired data is sent to the Gumstix.

9. Now the ads1271 device needs to be stopped on the Gumstix, by canceling the command in step 6. For a next test the ads1271 does not have to be removed, but can with "modprobe -r ads1271".

10. Before a next test run is performed, it is best to clear the FIFOs and SRAM, by turning *direct through* on through UART. Optionally to reset the whole DAQ, DAQ_RESET can be turned on high.

[3]The ssh config file contains the parameters: Compression: no; Ciphers: arcfour,blowfish-cbc; Cipher: blowfish. The file specifies encryption algorithms with low computation time, to speed up the transmission.

Chapter 5

Evaluation

We performed tests and evaluated them for different properties of the DAQ, mainly concentrating on the time precision and event/data throughput. Concerning time precision we looked into two properties:

- The offset of the estimated *full_second* and the actual PPS signal.
- The drift of the estimated time, when there is no PPS signal over a long period of time.

In terms of throughput we evaluated:

- The burst size/time on maximum event rates, until the tracing FIFO is full.
- The burst size/time and rate, until the SRAM is full.
- The maximum constant throughput, in terms of *bytes/s* and tracing events per second.

Further we look into burst sizes and constant rates of actuation events. We perform a testcase, where we monitored a blinking Tmote Sky node, and provide the tracing data and power profile of the target.

5.1 Time Precision

To measure time offset and drift a debug option was added to the DAQ. When activated the DAQ sends after each PPS signal several timing messages to the PC over UART, instead of sending acknowledgments from the Control module. These timing messages contain properties such as: the actual cycle count between two PPS signals; the current seconds count; the cycle offset; and the new and average cycle difference of the PPS signals.

To be able to filter weak PPS signals and its influences on the test, status messages of the GPS unit were continuously logged. A GPS status message contains mainly information about the current coordinates and time and more importantly as filter criteria: the number of available GPS satellites; a value for signal quality (hdop: horizontal dilution of precision); and the GPS fix status. The status message could also be empty or not received at all, due to weak GPS

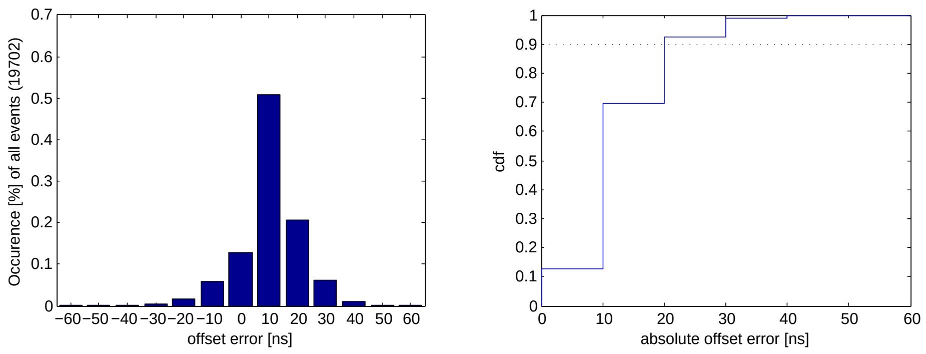


Figure 5.1: Left: distribution of the time offset after one second, total: 19702 data points. Right: the same data set in an empirical cumulative distribution function (cdf)

signals. The GPS unit is configured to send a PPS signal only if it runs in GPS time locking mode. Therefore longer periods without any PPS signal can occur during a test.

5.1.1 Time offset after one second

To determine the precision of our Time Calibration module, i.e. internal clock precision, we measure the offset between the estimated full second and the actual PPS signal.

For the offset measurements the PPS signal was applied to the Time Calibration module as it comes from the GPS unit. The test run for 8 hours during the day and the timing messages were collected. The raw data contained 24547 test points with time offsets in cycles.

Due to weak GPS reception of the test antenna (behind a window, indoors), test points after weak or no GPS/PPS signals had to be removed. The signal is considered weak if: (a) the GPS status message was received empty or not at all; (b) the PPS signal was lost for more than 3 seconds; (c) the GPS unit had less than 6 satellites in range; or (d) if the $hdop < 3$. Because the DAQ needs multiple cycles to adjusts the internal time to a new or changing reference signal, 9 test points after a weak signal were removed. The cleaned test set had 19702 test points left.

The distribution of the time offsets is shown in Figure 5.1. In the bar chart on the left can be seen that in 40% of the cases an offset of 1 cycle, i.e. 10 ns, occurred. The average is +1.0493 cycles , and the standard deviation is 1.08 cycles. The chart on the right shows the corresponding empirical cumulative distribution (cdf), computed with the absolute offset values. It shows that 90% of the offsets have an absolute value $\leq$ 30 ns, and 99% are $\leq$ 40 ns.

This means the accuracy of the DAQs internal time relative to a high precision PPS signal is well below 100 ns.

A minimum error could be expected, due to jitter on the PPS signal and to the sampling error of the PPS signal. The PPS signal of the u-blox LEA-6T module, has a standard deviation of $6.7ns$ [7], average is zero, since it is the reference signal. Additionally there is a sampling error. By sampling the PPS signal once per $10ns$ interval, the measured difference of two PPS signals can have an error of ± 1 cycle, i.e. $\pm 10ns$. Our evaluation reaches a similar error deviation with 1.07 cycles, but the average is shifted one cycle.

In the previous FlockLab implementation different observers had a pairwise timing error in average of $166\mu s$ and maximal of $1179\mu s$, considering observers connected by Wi-Fi. We derive the pairwise error of our implementation without measurements, due to the availability of just a single prototype to test. We assume the pairwise error is (maximal) twice the offset error. In our setup with a good GPS PPS signal, the average pairwise error is therefore below $30ns$ and the maximum, considering the results of the measurements, is below $120ns$. This is a improvement by 4 magnitudes, providing every observer node has an accurate PPS signal input.

5.1.2 Time Drift after losing PPS signal

To measure the time drift, the PPS signal from the GPS unit was cut off intentionally after the Time Calibration module calibrated the internal time. We implemented a clock drift routine on the FPGA, which would apply the PPS signal for 120 seconds and then cut it off for 180 seconds. To the debug unit, however, still used the PPS signal to measure the drift. Therefore the debug unit continuously sends timing messages containing the offset between the estimated *full_second* and the PPS signal. The test run for 22 hours and logged 316 periods of PPS signal cut offs. Those were filtered for incomplete periods, due to PPS signal and therefore timing message losses. The test set we investigate finally contains 309 periods. The distribution of the offsets after $1-9$ cycles of losing the reference signal are shown in the left empirical cdf plot in Figure 5.2. On the right the distributions of the offsets after $10, 20, \ldots, 100$ cycles are plotted.

As expected the drift of the internal time, in respect to the PPS signal, increases the longer the reference signal is lost. The distributions show that after x seconds without a reference signal at least 90% of the offsets are within $20x$ nano seconds. For example after 100 s in at least 90% of the cases the clock drift is less than $(20 \times 100)ns$, i.e. less than $2\mu s$.

This results show, that even after losing the reference signal for multiple seconds, the pairwise timing error of different observers should be still below

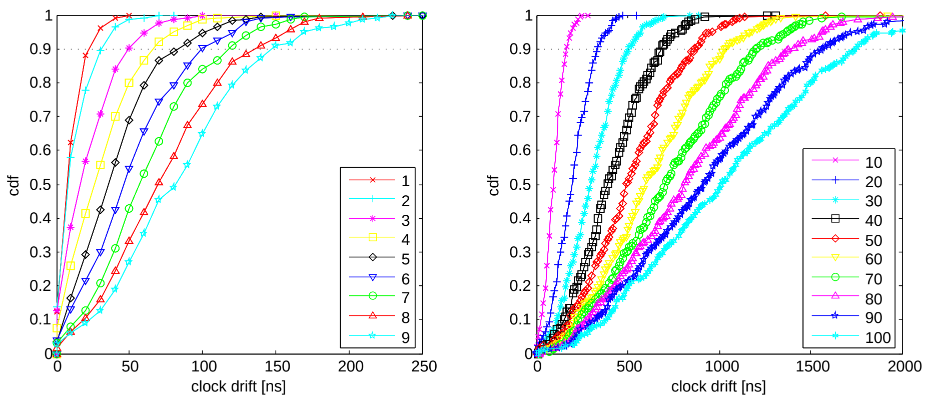


Figure 5.2: CDF of the time drift after 1-100 seconds without a reference signal.

$1\mu s$. Which is an improvement, compared to the average pairwise timing error of $166\mu s$ in a setup without the DAQ and with continuous time synchronization (over NTP).

5.2 Throughput

In this section we look into the actual throughput rates of the DAQ, and compare it to the requirements from Chapter 3.

To evaluate the throughput of the DAQ, we implemented an event generator module. With this module we toggle internally the tracing pins and generate thereby tracing events. This feature allows to fully control the events generated and it is easy to compare them to the received packets at the Gumstix.

The drawback of this approach is the alignment of the generated events with the same clock source as the DAQ runs with, and are therefore optimally aligned to detect. In later operation the target clock might not be aligned with the DAQ clock, and side effects influencing the detection accuracy could result. But for throughput test the setup is adequate.

Maximum FIFO rate In the first throughput test we evaluate how many events can be successfully traced before the tracing FIFO is full. We generated an event per cycle, which is the maximum resolution in detection. The theoretical limit are the 1024 slots of the FIFO plus the amount of packets the SRAM module can read out of the FIFO in the meantime. The SRAM module needs 16 cycles to write a packet to the memory, so it can read 64 packets from the tracing FIFO in the 1024 cycles. The measured maximal burst size without exceeding the FIFOs slots is 1070 packets, which is within 2% of what we expected.

Maximum SRAM rate The second throughput test evaluates the maximal burst size before the SRAM exceeds its space. Every 16 cycles an event is generated, which corresponds to the write time of the SRAM module. The SRAM can store 262144 packets and write theoretically up to $6.25M$ packets per second, at that rate the SRAM is full within $40ms$. We measured that actually 270000 events can be detected, one every 16 cycles, before first the SRAM and then the tracing FIFO is full and has to reject packets. There is more space measured, because additionally to the SRAM the FIFOs get filled up too.

Maximum continuous rate The third throughput test investigates the maximum continuous event rate, at which neither the FIFO nor the SRAM will exceed their capacity. The bottleneck on the DAQ is the SPI connection to the Gumstix. With a baud rate of $12M\frac{baud}{s}$ and a packet size of 34 baud (32 bit data + 1 frame + 1 setup), theoretically every $2.8\mu s$ a packet can be sent, i.e. $353k$ packets per second. The theoretical maximum can be reached and the data is actually sent from the DAQ over SPI to the Gumstix. With the new driver the Gumstix is able to receive the packets and write them through DMA into memory buffers. However the process which reads those buffers is too slow so far to handle the throughput, and packets get lost (i.e. buffers are overwritten again before read). Therefore a delay of a 15 cycles was introduced between

cycles between two events	max. burst size
1 (10ns)	1070
16 (160ns)	270000
350 (3.5μs)	continuous

Table 5.1: measured throughput burst sizes and maximum continuous rate

two SPI packets. A user process on the Gumstix reads the data from SPI and forwards it over Ethernet to the PC. We measured 285k packets per second as the maximum stable throughput rate, which corresponds to 3.5μs per packet. The throughput measurements are summarized in Table 5.1.

With a continuous throughput of 285k packets per second, the DAQ is able to satisfy the required rates of 3 out of 4 target platforms (see Table 3.2 from Chapter 3). However we miss to comply with the rates needed by the Opal platform by $\frac{1}{4}$.

In terms of burst rate we meet the requirements off all target platforms. The DAQ is able to detect events at 100MHz, which includes 19.2MHz burst of the Opal platform.

Adding power profiling data By running the power profile service at the same time, the maximum event rate is decreased, however the through put in data (ADC packets and tracing packets combined) remains the same. At high-speed mode the ADC unit generates 56000 samples per second. Which is roughly $\frac{1}{5}$ of the maximum SPI bandwidth. Therefore by running the ADC unit too, the maximum continuous event detection rate changes to 229k events per second. At the default configuration at the moment, though, power profiling runs in high-resolution mode and the sample divider is set to 2. In this case the power profiling service needs $\frac{1}{20}$ of the bandwidth.

5.3 Actuation events

As seen in Section 4.1.7 an actuation command transmission takes 70μs. Then some cycles are needed to execute the command, which is insignificant compared to the transmission time. This means that in theory 14285 signals per second could be actuated. It is hard to actually reach that rate,because an actuation command contains the time when to set a signal. Therefore host system sending the commands would need to send exactly every 70μs an actuation containing the previous time + 70μs. If it sends actuation faster the actuation FIFO will be full at some point. Is it slightly slower, then the commands will arrive after their due time.

In a simple and imprecise setup we managed to send 2340 actuations one every 70μs before the commands arrived too late.

But the actuation service can set the signals much faster than every 70μs over a short period. One actuation can be executed every 500ns at beginning of a timeslot, up to 130 after each other due to the actuation FIFO. The timings can be improved by enlarging the actuation FIFO.

5.4 Testcase

In this section we present the results of an actual test run on the FlockLab with the DAQ unit integrated. A Tmote Sky node is the target node on the FlockBoard. It runs a simple program (Blink), implementing a 3 bit counter on 3 onboard LEDs and on the 3 tracing signals LED1-3. The DAQ is set up to trace the 3 LED signals and to turn on power profiling 20 seconds after start. Figure 5.3 shows the acquired data plotted on a time line: The levels of the signals LED1-3 on the 3 bottom graphs ($\in [0, 1]$); the sum off all LEDs with level 1 ($\in [0, 1, 2, 3]$); and the power profiling data in $[mA]$. The scale of the y-axis only applies to the power profile data.

As expected the state of an LED influences the current drawn by the target node, the more LEDs burning the more current is needed. It can also be observed that not all LEDs need the same amount of power, due to different colors of the LEDs. Further there are peaks in the LED count graph at every second LED toggle. This is correct, because one LED is turned off before the next is turned on, with a time difference of up to $200\mu s$. This becomes visible by zooming into the data set as in Figure 5.4. The figure shows a time period when all LEDs are turned of and we notice that one LED is turned off after each other.

To compare the measuring accuracy to the FlockLab without the DAQ, we measured the time differences between the rising edges of LED 1. The distribution of the different times measured either with or without the DAQ is shown in Figure 5.5. Over 90% of the measurements with the DAQ are within the same $8\mu s$ time period. Whereas the measurements without the DAQ are spread over roughly $200\mu s$. The wider distribution without the DAQ, is due to the variating delays of interrupts handling. With the DAQ still some distribution is measured. Which might be, because the program running on the target is not as precise either.

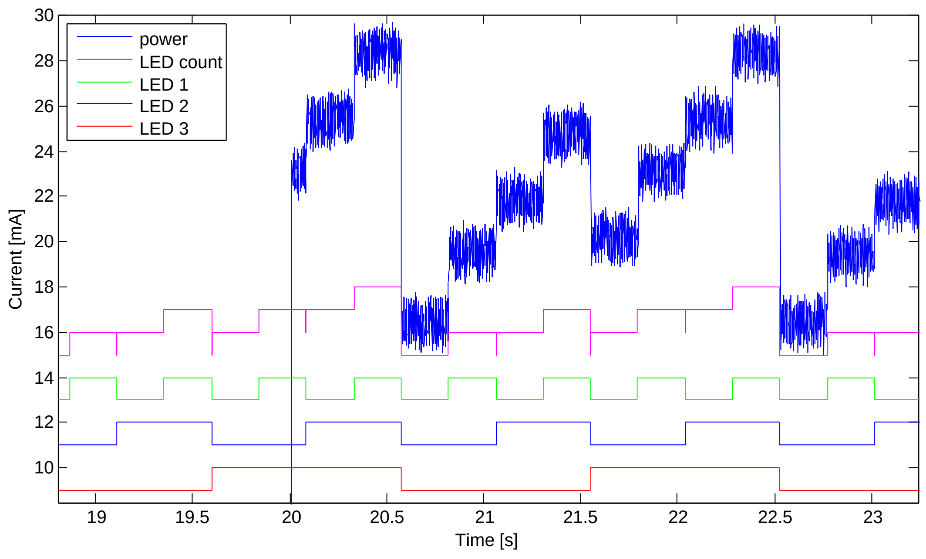


Figure 5.3: Results of the Blink testcase. The LED count correlates with the power profile.

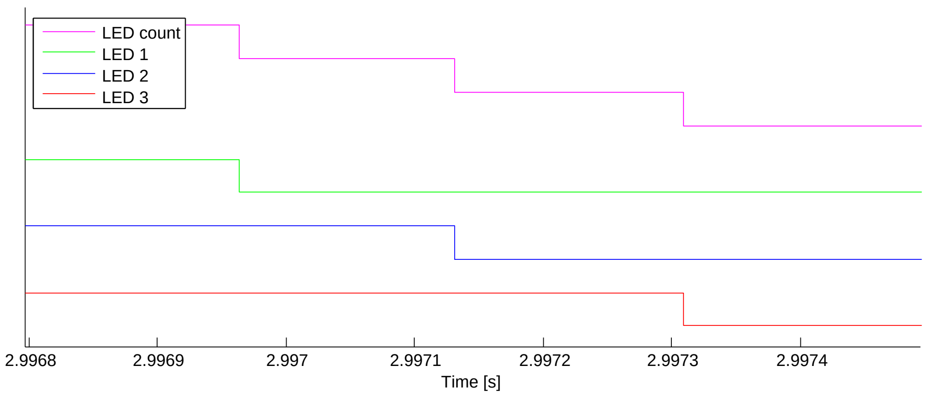


Figure 5.4: Zoom into the Blink testcase, where all LEDs turn off at 'the same time'. Shows the delays due to execution time and sequence on the target node.

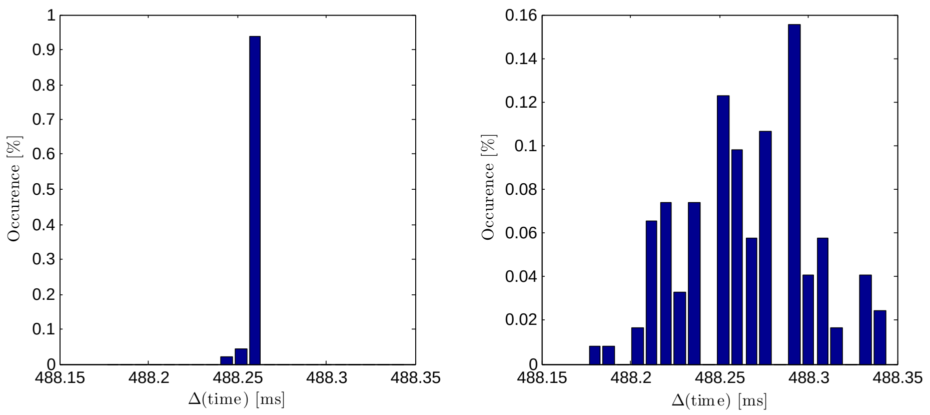


Figure 5.5: Measured time differences between the rising edges of LED1 in the Blink Testcase. Left side: FlockLab with DAQ, right side: without DAQ

5.5 Final FPGA chip

After finishing the implementation of the DAQ on the Anvyl Developer Board with an Spartan-6 LX45 FPGA, we investigated on which of the Spartan-6 FPGAs we could transfer our project. The smallest chip of the family, the LX4, has not enough logic cells to implement our design. But the next larger LX9 meets our requirements. The Spartan-6 LX9 FPGA has 144 pins thereof 120 user I/Os, 9152 logic cells and costs around 15-20. A detailed logic utilization is given in Table 5.6, as well as a power estimation computed by the Xilinx XPower Analyzer with standard values.

On-Chip	Power[W]	Used	Available	Utilization[%]
Clocks	0.027	1	-	-
Logic	0.014	3160	5720	55
Signals	0.014	5127	-	-
BRAMs(8K, 16K)	0.004	4, 6	64, 32	6, 19
DSPs	0	2	16	13
PLLs	0.087	1	2	50
IOs	0.071	86	102	84
Leakage	0.017			
Total	0.235			

Figure 5.6: Logic Utilization and estimated power (by Xilinx XPower Analyzer).

Chapter 6

Conclusion and Outlook

6.1 Conclusion

With the introduction of the data acquisition unit (DAQ), we improved the existing FlockLab in terms of time synchronization and event detection rate. The pairwise timing error has been reduced from $166\mu s$ to $30ns$, with an accurate pulse per second (PPS) signal as reference. The event detection rate could be increased from 12.5 kHz up to 10MHz. Moreover, each reported event contains a timestamp with a resolution of $500ns$ (2MHz), which can be further improved to actually utilize the low timing error. Those enhancements improve all services of the FlockLab: The tracing signals can be timestamped more precisely and at a higher rate. The actuation signals can be set more accurately, at a higher rate and with a smaller pairwise time error over different observers. Further, the continuous data rate of the power profiling services slows the other services no longer down.

6.2 Outlook

With the capabilities of a FPGA more can be achieved. A major improvement could be to reduce the timeslots of now $500ns$. This can be implemented without changing the size of the 21 bit timestamp, by relating the timestamp not to a packet sent every second, but more often. In terms of ADC samples, the sample divider could instead of discarding the surplus of samples return an average over multiple samples.

Then on the Gumstix the SPI driver can be improved, to handle the full 12 Mbit/s data stream without loosing buffers and to support reading trailing bytes from the DMA buffer.

But the next step certainly is to transfer our prototype from the Developer Board to a custom PCB, which can actually be placed on the FlockBoard. Therefore a concept is required to provide the DAQs on all the observers with a accurate PPS signal and avoiding the need of a GPS device on all observers. Furthermore a software integration of the DAQ is needed. The FlockLab test management has to be adapted to the DAQs control options and data handling on the server needs to be adjusted to the new data format.

Appendix A

Additional Tables

A.1 SD card write speed test

We tested SD card write speed with (Table A.1):
dd count=X bs=Y if=/dev/zero of=/media/card/root/flocklab/test/test.file
Read speed tested with *hdparm -t /dev/mmcblk0* results in 2.1 MB/sec.

count/bs	1M	500k	10k	512
10	17	6.1-17	12	2.2
100	0.89	0.84	15	5.4
1000	0.52	0.55	14	6.3

Table A.1: SD card write speeds [MB/sec]. bs: bytes per block; count: no. of blocks. (FlockBoard 029)

A.2 UART packet payload definition

tracing pins

bit 7	6	5	4	3	2	1	bit 0
LED1	LED2	LED3	INT1	INT2	SIG1	SIG2	RST

actuation pins and levels

bit 3	2	1	bit 0
SIG1	SIG2	RST	ADC

callback pin or off

0XXX	1000	1001	1010	1011	1100
off	LED1	LED2	LED3	INT1	INT2

other

on/error	off/ack	sample divider	adc off delay
1111	0000	$[1 - 2047]$ 11bit	$[1 - 127]$ 7bit

Table A.2: UART payload bitwise definition

A.3 60pin Connector

The DAQ unit will be connected through an existing 60 pin connector to the Gumstix and the target nodes (FlockBoard). Therefore the DAQ is placed between the existing 60 pin connection of the Gumstix to the FlockBoard. Not all 60 pins are required, the DAQ captures those needed and connects the other pins directly through. Further a mechanism is required to connect all 60 pins directly from the Gumstix to the FlockBoard, hence to omit the DAQ at all. This is used to initialize the FlockBoard from the Gumstix. An overview of all captured pins is given in Table A.3.

On the FlockBoard side the DAQ uses 4 different interfaces with multiple pins each to monitor the targets. Those are connected from the board to the DAQ and include: 3 Target Inputs (nr: 13,16,41), 5 Target Outputs (pins: 8,9,10,22,49), 4 ADC SPI pins (nr: 21,44,45,50) and 3 USB pins (nr: 11,58,59). Further all GND and all VCC pins are connected.

To communicate with the Gumstix the DAQ can use the SPI and/or the USB interface. If USB is used, the DAQ needs to provide a hub, so the other USB devices on the board are still connected to the Gumstix (USB master). Additionally to the already occupied pins a reset signal (DAQ_RESET) is needed to put the DAQ into an initial state, therefore an unused pin from the Gumstix is needed. In the DAQ prototype the DAQ_RESET is a hardware switch on the Anvyl developer board.

Service	FPGA Pin Nr.		Gumstix Pin Nr.	Gumstix name	Board name
	FB	Gum			
tracing	JB7	JE7	8	GPIO71	TARGET_LED1
tracing	JB4	JE4	9	GPIO70	TARGET_LED2
tracing	JB3	JE3	10	GPIO69	TARGET_LED3
tracing	JB2	JE2	49	GPIO113_W	TARGET_INT1
tracing	JB1	JE1	22	GPIO87	TARGET_INT2
actuation	JD1	JD7	13	GPIO60	TARGET_RST
actuation	JD3	JD9	16	GPIO75	TARGET_SIG1
actuation	JD2	JD8	41	GPIO74	TARGET_SIG2
SPI	JA1	JA7	21	GPIO19_SSPSCLK2	SPI_SCLK
SPI	-	-	23	GPIO13_SSPTDX2_W	-
SPI	JA2	JA8	44	GPIO14_SSPSFRM2_W	RTC_nINT
SPI	JA3	JA9	45	GPIO11_SSPRDX2_W	SPI_MISO
SPI	JA4	JA10	50	GPIO59	ADC_nCS
USB	-	-	58	GPIO28	USB_HUB_nRESET
USB	-	-	59	USBH_N1	USB_HUB_UPLINK_N
USB	-	-	11	USBH_P1	USB_HUB_UPLINK_P
DAQ_RESET	switch 0		-	-	-
-	GND		1,7,31,42,60	GND	GND
-	-	-	28,29,30	V_BATT	VCC_5.0

Table A.3: Important Gumstix 60pin assignments

A.4 Compare FPGA Developer Boards

Device	**Cyclone III DE0**	**Cyclone III Starter Dev**	**Cyclone IV DE0nano**	**Igloo nano**	**Cyclone IV DE2-115**
Manufactor	Terasic	Altera	Terasic	Microsemi	Altera
LEs	15K	24K	22K	6K	114K
Memory	8MB SDRAM 4MB Flash SD slot	256Mb DDR 1MB SRAM 16MB Flash	32Mb SDRAM 2Kb EEPROM	-	2MB SRAM 128MB SDRAM Flash, EEPROM
I/O Board	72+8	HSMC: 84	72+8	68	54+172 HSMC
I/O FPGA	346	215	153	68	528
Serial Config $	4-50 (EPCS)	4-50 (EPCS)	4-50 (EPCS)	-	'SPI'
Board Cost $	119 (79)	200	79 (59)	116	600 (300)
FPGA Cost $	30	40-70	44-66	10-20	
Ausstattung	0	+	0	0	++
Oszillator (MHz)	50	50	50	20	3x50
Pin level board			3.3V		1.5-3.3V
Pin Level FPGA			1.5-3.3V		1.5-3.3V
SW (at ITET)	Quartus Web Edition, 10k lines (100 Licences)				Quartus
FPGA int Mem			590 Kb		3800 Kb
USB (inerfaces)					Y
Configuration					'SPI'/schematics
Availability					unclear

Device	**Igloo2**	**Spartan 3 Board**	**Spartan 6 Nexys 3**	**Artix 7 Nexys 4**	**Spartan 6 Anvyl**
Manufactor	Microsemi	Digilent	Digilent	Digilent	Digilent
LEs	12K	4K/17K	14K	15K	44K
Memory	64Mb Flash 512MB DDR	216 Kb RAM 2Mb Flash 1MB SRAM	3x 16MB	16MB C.RAM	2MB SRAM 128MB DDR2 Flash
I/O Board	64	96	48	48	56+10
I/O FPGA	169-377	60-170	100-570	170-500	100-570
Serial Config $	-	Y	Y	Y	Y
Board Cost $	400 (99)	150	230 (120)	300 (160)	540 (350)
FPGA Cost $	20-50	10-50	30 (10-130)	115-160	10-130
Ausstattung	++	+	++	++	++
Oszillator	50	50	100	100	100
Pin level board					-3.3V
Pin Level FPGA					-3.3V
SW (at ITET)		ISE Web Pack, 50k lines (50 licences)			
FPGA int Mem					2 Mb
USB (inerfaces)					
configuration					SPI/schematics
Availability					Yes

Table A.4: FPGA Developer Boards comparison

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Herbstsemester 2013

MASTERARBEIT

für
Benjamin Dissler

Betreuer: Roman Lim
Stellvertreter: Christoph Walser

Ausgabe: 30. September 2013
Abgabe: 11. April 2014

High-Speed Data-Acquisition for FlockLab

Einleitung

FlockLab [1] ist ein Testbed für drahtlose Sensornetzwerke. Darunter versteht man eine Installation die es erlaubt Programme direkt auf den physikalischen Knoten des Netzwerks zu testen. Typische Dienstleistungen (Services) eines Testbeds sind das Programmieren der Knoten, ein Kommunikationskanal (Serielle Schnittstelle) und das Messen des Stromverbrauchs. Zusätzlich kann mit FlockLab der Zustand von GPIO-Pins erfasst und auch gesetzt werden (GPIO tracing/actuation). In FlockLab wird jeder Sensorknoten von einem Beobachterknoten (Observer) überwacht. Auf diesen Knoten wird ein eingebettetes Linux-System verwendet um Testabläufe zu verwalten und Messdaten zu sammeln.

Da die Daten der Tests verteilt gesammelt werden, müssen sie miteinander synchronisiert werden um sie in einen globalen Kontext zu bringen. Dazu wird zur Zeit das Network Time Protocol benutzt, welches eine Zeitsynchronisation innerhalb von einigen 100 μS erlaubt. Das Linux-System ist verantwortlich um die gemessenen Daten mit einem Zeitstempel zu versehen. Da darauf aber mehrere Prozesse gleichzeitig aktiv sein können kommt es zu nicht-deterministischen Verzögerungen, was die Ungenauigkeit weiter erhöht.

Die beiden GPIO Services können sehr hohe Datenraten generieren. Damit ist es sehr leicht möglich an die Grenzen des Observers zu kommen und somit unvollständige Messungen zu generieren.

Das Ziel dieser Masterarbeit ist es den Beobachterknoten zu erweitern um die beiden erwähnten Nachteile zu beseitigen. Konkret soll eine Platine gebaut werden mit einem dedizierten Datenverarbeitungselement. Diese Platine soll sich, wie in Abbildung 1 gezeigt, in die bestehende Architektur einbetten und ein exaktes Zeitsignal (z.B. GPS) als Referenz benutzen.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest. Insbesondere soll genügend Zeit für die Schlusspräsentation und den Bericht eingeplant werden.

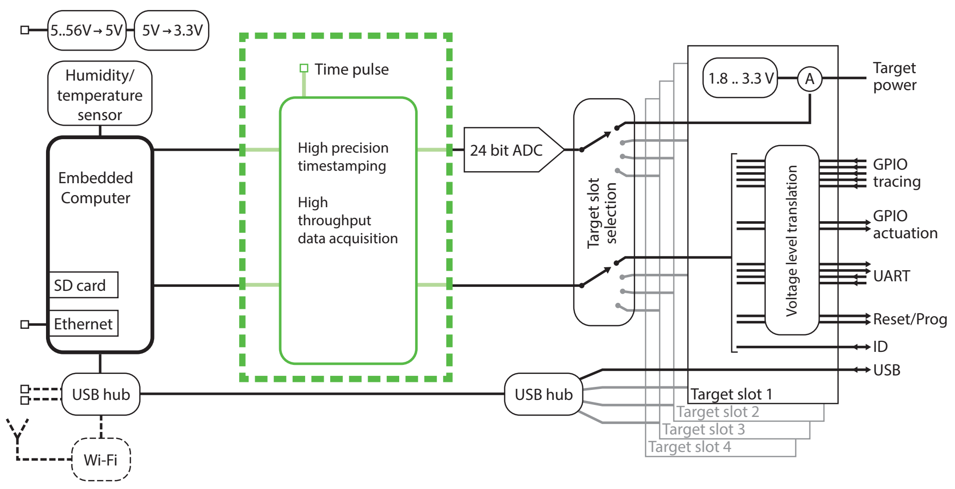


Abbildung 1: Überblick des FlockLab-Observers: Der eingebettete Computer steuert den Testablauf und sammelt verschiedene Daten (Stromverbrauch, GPIO Zustände und serielle Kommunikation). Gestrichelt dargestellt ist das neue Datenverarbeitungselement.

2. Verschaffen Sie sich einen Überblick über die Architektur von FlockLab und dessen Komponenten und Services.
3. Machen Sie sich mit den relevanten Arbeiten im Bereich Testbeds, Zeitsynchronisation und Messsysteme vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie gezielt nach relevanten Publikationen. Prüfen Sie welche Ideen/Konzepte Sie aus diesen Lösungen verwenden können.
4. Erstellen Sie eine Übersicht der Anforderungen an das Datenverarbeitungselement für FlockLab.
5. Entwerfen Sie einen lauffähigen Prototyp und implementieren Sie diesen.
6. Testen Sie den Prototyp.
7. Evaluieren Sie die Leistung des Prototyps.
8. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag sowie mit einem Schlussbericht.

Durchführung der Masterarbeit

Allgemeines

- Der Verlauf der Masterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über einen PC mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Masterarbeit in einem Kurzvortrag (maximal 5 Minuten) vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums (maximal 20 Minuten).

- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (email).
- Weiterführende Angaben finden Sie in [2].

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes sowie alle relevanten Source-, Object und Konfigurationsfiles bis spätestens am *11. April 2014* dem betreuenden Assistenten oder seinem Stellvertreter ab. Diese Aufgabenstellung soll im Bericht eingefügt werden, genauso wie das unterschriebene Unterschriftenblatt *Plagiat* des Rektorats. Die entsprechenden Richtlinien des Rektorats sind einzuhalten.
- Die Arbeit wird benotet anhand der Kriterien beschrieben in [3].
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Literatur

[1] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, “Flocklab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems,” in *Proc. of the 12th Intl. Conf. on Information processing in sensor networks (IPSN)*, 2013.

[2] TIK, “Studien- und Masterarbeiten: Merkblatt für Studenten und Betreuer.” Computer Engineering and Networks Lab, ETH Zürich, Switzerland, Apr. 2009.

[3] TIK, “Notengebung bei Studien- und Diplomarbeiten.” Computer Engineering and Networks Lab, ETH Zürich, Switzerland, May 1998.

Bibliography

[1] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proceedings of the 12th international conference on Information processing in sensor networks*, IPSN '13, pages 153–166, New York, NY, USA, 2013. ACM.

[2] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 73–84, Chicago, IL, USA, April 2011. Best paper award.

[3] Niels Penneman, Luc Perneel, Martin Timmerman, and Bjorn Sutter. An fpga-based real-time event sampler. In Phaophak Sirisuk, Fearghal Morgan, Tarek El-Ghazawi, and Hideharu Amano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 364–371. Springer Berlin Heidelberg, 2010.

[4] Wen-Qin Wang. Gps-based time phase synchronization processing for distributed sar. *Aerospace and Electronic Systems, IEEE Transactions on*, 45(3):1040–1051, July 2009.

[5] L. Gasparini, O. Zadedyurina, G. Fontana, D. Macii, A. Boni, and Y. Ofek. A digital circuit for jitter reduction of gps-disciplined 1-pps synchronization signals. In *Advanced Methods for Uncertainty Estimation in Measurement, 2007 IEEE International Workshop on*, pages 84–88, July 2007.

[6] Texas Instruments. 24 bit, wide bandwidth analog to digital converter (sbas306f). Technical report, Texas Instruments Incorporated, October 2007.

[7] ublox. Gps-based timing - considerations with u-blox 6 gps receivers (gps.g6-x-11007). Technical report, u-blox AG, March 2011.

[8] Jean P. Nicolle. Spi - a simple implementation. `http://www.fpga4fun.com/SPI2.html`. Accessed: 2014-04-04.

[9] Intel. Intel pxa27x processor family, developers manual. Technical report, Intel Corporation, October 2004.

[10] Peter Bennett. Vhdl uart. `https://github.com/pabennett/uart`. Accessed: 2014-04-04.