

Universal Asynchronous Receiver/Transmitters:
A Software Implementation Approach

Universal Asynchronous Receiver/Transmitters: A Software Implementation Approach

Diploma Thesis
by
Herbert Valerio Riedel

submitted to the
Faculty of Computer Science
Vienna University of Technology

in partial fulfillment of the requirements for the degree
Diplom-Ingenieur (Dipl.-Ing.)

written at the
Institute of Computer Aided Automation
Research Group Industrial Software

supervised by
Ao. Univ. Prof. Dipl. Ing. Dr. Techn. Thomas Grechenig
Dipl. Ing. Dr. Techn. Philipp Tomsich

Vienna,
7th December 2005



Abstract

Universal Asynchronous Receiver/Transmitters (UART) are hardware components usually required to handle serial communication protocols; protocols that are widely in use, ranging from implementing low-cost communication paths between peripherals on embedded systems to connecting hosts to external peripherals or other host systems.

With the continued growth of CPU performance it becomes viable to emulate UARTs traditionally implemented as dedicated hardware components in software. Software implementations allow for more flexibility with respect to upgrade-ability, potentially shorter time to market, and cost savings at the expense of higher CPU utilisation. When the host system is expected to service other processes while the software UART is active, challenges are faced for the implementation, depending on the real time constraints imposed by the serial protocol.

This thesis gives an overview of various serial protocols in use and discusses the difficulties arising when emulating the corresponding UART in software. Furthermore, different common software UART approaches for the implementation of ISO 7816 serial protocol are presented in the context of the general purpose non-real-time Linux kernel, and discussed with respect to reliability and (whole system) performance. Finally, a novel algorithm is presented which meets the posed requirements of a portable, response-conserving, reliable, good performing, and protocol conforming implementation.

Zusammenfassung

Universal Asynchronous Receiver/Transmitters (UART) sind verbreitet im Einsatz befindliche serielle Schnittstellenbausteine, die üblicherweise zur Unterstützung von seriellen Kommunikationsprotokollen eingesetzt werden. Dies sind Protokolle, die breite Anwendungen finden: vom Einsatz auf Mikroprozessorplatinen zur kosteneffizienten Realisierung der Kommunikation zwischen den Komponenten, über die Anwendung zur Verbindung von Computern mit externen Periphäregeräten bis hin zur Verbindung von Computersystemen untereinander.

Mit dem kontinuierlichen Anstieg der Rechenleistung ist es praktikabel geworden, UARTs, die traditionellerweise als eigenständige Hardware Bausteine ausgeführt sind, als Software-Komponenten zu implementieren. Software-Implementationen bieten höhere Flexibilität in Bezug auf die Aufrüstbarkeit und Fehlerbehebung, potentiell kürzere Produkteinführungszeiten und nicht zuletzt Kostenersparnisse – alles auf Kosten höherer Beanspruchung der Zentraleinheit. Wenn das System – während der Software UART aktiv ist – noch weitere Aufgaben erfüllen können soll, so ergeben sich je nach Beschaffenheit der Echtzeit-Erfordernisse des seriellen Protokolls entsprechende Herausforderungen für die Implementation des Software UARTs.

Diese Diplomarbeit gibt einen Überblick über einige verbreitete serielle Protokolle und erörtert die mit der Software-Emulation verbundenen Schwierigkeiten. Darüberhinaus werden einige übliche Software UART Implementationsmethoden vor dem Hintergrund des Einsatzes mit dem Mehrzweck Nicht-Echtzeit Betriebssystemkerns Linux vorgestellt und in Hinblick auf ihre Verlässlichkeit und das Leistungsverhalten des Komplettsystems diskutiert. Schliesslich wird ein neuartiger Algorithmus vorgestellt, der die Anforderungen an eine leicht portierbare, reaktionszeiterhaltende, zuverlässige, leistungsstarke und protokollkonforme Implementation erfüllt.

Contents

List of Figures	v
List of Tables	vii
Listings	ix
Preface	xi
1 Introduction	1
1.1 Serial Communications and UARTs	2
1.2 Replacing Hardware with Software	7
1.3 Real-Time Correctness	9
1.4 The Challenge	10
1.5 Road-Map	11
2 The UART Problem	13
2.1 The Physical Layer	13
2.2 The Data Frame	14
2.3 The ISO 7816 Serial Transmission Protocol	17
2.4 Sampling the Bits	20
2.4.1 Sampling with Clock Drift	21
2.4.2 Sample Point Jitter	22
2.4.3 Start Bit Synchronisation	23
3 Approaching a Solution	25
3.1 Dedicated Polling	25
3.2 Timer Interrupt Triggered Processing	27
3.3 Edge Interrupt Triggered Bit Detection	29
3.4 Summary	32

4	The Implementation	35
4.1	The Host Environment	35
4.1.1	The Hardware	35
4.1.2	The Software	36
4.2	The Driver	38
4.2.1	Constraints	38
4.2.2	The Hybrid Approach	39
4.2.3	Actual Implementation	41
5	Experiments in Latency and Time	43
5.1	Interrupt Latency	43
5.1.1	Idle System	45
5.1.2	Network Stress	45
5.1.3	Cache-Disabled System	45
5.1.4	System Under Combined Stress	46
5.2	Achieved Data Rates	52
5.2.1	Idle System	54
5.2.2	System under Stress	54
6	Analysis of the Results	59
7	Conclusion	63
7.1	Future Work	64
7.1.1	Protocol Conformance	64
7.1.2	Latency Improvements	64
7.1.3	Performance Improvements	65
7.1.4	Reliability Improvements	65
7.1.5	Full-duplex support	65
A	ISO 7816 Software UART Driver Source Code	67
B	Latency Measurement Kernel Module Source Code	91
	Glossary	95
	Bibliography	99
	Colophon	103

List of Figures

1.1	PC16550D UART Block diagram	5
2.1	Binary line codes	15
2.2	ISO 7816 character frame	19
2.3	Binary signal $s(t)$ waveform with periods of undefined state	20
2.4	Bit cell with sampling point	22
3.1	Timer interrupt triggered receiver state machine	31
4.1	Block diagram of card terminal embedded system	36
4.2	Sample positions with hybrid implementation	41
5.1	Interrupt latencies for idle system	47
5.2	Interrupt latencies for system with network stress applied	48
5.3	Interrupt latencies for idle system with disabled i-cache	49
5.4	Interrupt latencies for idle system with disabled d- and i-cache	50
5.5	Interrupt latencies for combined stress	51
5.6	ISO 7816 T=1 block-frame sequence of <i>ISO READ BINARY</i> transaction	52
5.7	ICMP Ping round-trip-times for idle system during communication	56
5.8	ICMP Ping round-trip-times for stressed system during communication	57

List of Tables

1.1	Comparison of ISO 7816, RS232, and I ² C	8
2.1	Allowed values for D according to the ISO 7816 specification	17
2.2	Allowed values for F and max f according to ISO 7816	17
2.3	Bit-rates for combinations of D and F	18
2.4	Comparison of ISO 7816 T=0 and T=1 protocols	19
5.1	Interrupt latencies for idle system	47
5.2	Interrupt latencies for system with network stress applied	48
5.3	Interrupt latencies for idle system with disabled i-cache	49
5.4	Interrupt latencies for idle system with disabled d- and i-cache	50
5.5	Interrupt latencies for combined stress	51
5.6	Bit-rates for implemented combinations of DIVCLK, D , and F	53
5.7	Communication benchmark for idle system	55
5.8	Communication benchmark for stressed system	55

Listings

3.1	Dedicated polling transmitter implementation	26
3.2	Dedicated polling receiver implementation	27
3.3	Timer interrupt triggered transmitter	29
3.4	Timer interrupt triggered receiver	30
3.5	Edge interrupt triggered receiver implementation	33
4.1	Simplified frames_tx implementation	37
4.2	Simplified hybrid sampling implementation	40
A.1	ISO 7816 software UART driver source code	67
B.1	Latency measurement kernel module source code	91

Preface

“Humour and knowledge are the
two great hopes of our culture.”

(Konrad Lorenz)

This document has been the hopefully successful result of several months attempting to put some meaningful content down to paper, while trying to *eschew obfuscation*. The pain associated with this experience was alleviated by the use of L^AT_EX and other invaluable tools¹ distracting from the actual task of filling countless pages with words, which put in the correct order may form a so-called master’s thesis; a task which can be proved² that a monkey hitting keys at random on a keyboard could achieve as well, given an infinite amount of time...

However, I have to admit that I was surprised that the topic of software UARTs, which at first seemed to me like a topic already exhausted in the 1980s, still offered enough material for a master’s thesis. Far from it! I have discovered in the course of the work on this thesis that the topic of *software peripherals* is still a very active area of research, as the primary goal seems to be to get rid of most specialised silicon components in favour of general one-fits-all chips, which are produced in high volumes with the associated cost benefits. So in the end, it seems once again, it’s all just about the money. But still, imagine a world, where you only need to buy a device whose appearance matches your furniture at home—or at least your taste—and the functionality of this device is provided by uploading the corresponding software. Maybe, you could even reuse old hardware modules you have just lying around, and breath new life into them by uploading new software. On the other hand, I have to admit, this sounds a bit like playing Frankenstein...

Acknowledgements. The impact of working on this thesis was not confined only to yours truly; therefore acknowledgements are due to various people I had to involve—

¹See the Colophon on page 103 for more information about the document preparation and typeset process.

²Based on the second Borel-Cantelli lemma, of which the notorious “infinite monkey theorem” is a special case.

that is, annoy—in order to obtain highly honourable contributions such as proof-reading, feedback, or even emotional support. For this, thanks go to T. Hirsch, J. Hetzl, P. Tom-sich, T. Volpini, K. Goger, C. Brem, and to everyone else I forgot to mention here explicitly.

Finally, I hope I haven't been responsible for too many dead trees.

Herbert Valerio Riedel
December 2005

CHAPTER 1

Introduction

By using software routines to emulate dedicated logic circuitry, i.e. hardware, the function of the emulated circuit can be taken over by a general-purpose circuit, for instance a micro-controller. In other words, specific hardware (components) can be replaced by software (being executed on general-purpose hardware components).

The ability to replace hardware components with software routines allows for additional flexibility, since the logic behaviour can be modified without the need to replace hardware components. Furthermore, the omission of hardware components, such as communication controllers, simplifies the hardware design as a result of circuit paths and possibly auxiliary components not being required anymore. This simplification can lead to smaller Printed Circuit Boards (PCB) and cost-savings.

When a multitasking operating system is involved in the process of emulating hardware by means of software, the implementation becomes more complex as it has to be taken into account that usually the software execution has to be distributed sensibly among various tasks, one of which is the hardware-emulation routine. Since the function of most hardware-components involves reacting to logic signals within certain time-frames, the software emulation has to respect these timing constraints as well. To this end, so-called Real-Time Operating Systems (RTOS) are usually deployed as they provide response-time guarantees and therefore facilitate the implementation of real-time processes. However, real-time operating systems are not always available for the chosen target platform; and on the other hand, even if available, their use might not be appropriate if the provided Application Programming Interface (API) complicates the implementation of the complete application.

The goal of this thesis is to show how to replace a hardware component—specifically a component for communicating with smart-cards—by a software driver being executed in the context of a widely available general-purpose multitasking operating system such as Linux, i.e. without relying on real-time facilities.

This chapter gives a general overview about the problem domain by discussing se-

rial protocols, the respective serial communications controllers, and their application (among others) for communicating with smart-cards. Smart-cards are often associated with credit card shaped (although other form factors are defined as well) identification cards carrying under a metallic contact pad an embedded microprocessor which controls the access to the information stored therein. Next, the discussion will be narrowed to the class of universal asynchronous serial communication protocols, and furthermore to the smart-card related serial protocols.

Then hardware-to-software migration is discussed and the associated problem of real-time correctness and latencies is brought up. Finally, the problem tackled by this thesis and the motivation to do so is restated explicitly.

The chapter concludes with a road-map of the rest of this thesis.

1.1 Serial Communications and UARTs

Serial communication protocols were used long before computers were available. An example for such an early serial protocol developed in the 19th century for use in telegraphy is known as the Morse code[16], which uses variable length sequences of short and long pulses (traditionally called *dots* and *dashes* respectively) to encode characters.

Today in the age of computer technology, many communication protocols of serial nature are in use for connecting various kinds of devices ranging from:

- on-board components, e.g. I²C[27] or System Management Bus (SMBus)[31];
- local external devices, e.g. Universal Serial Bus (USB)[36]; up to
- long distance connections between computer host systems, e.g. RS232C or even Ethernet.

Furthermore, serial communication protocols exist for different topologies:

- point-to-point connections (RS232C);
- single-master/slave bus systems (RS422, RS485¹); and
- multi-master bus systems (I²C, SMBus, Ethernet).

According to the Glossary of Telecommunication Terms[23], the term *serial transmission* is generally defined as:

The sequential transmission of the signal elements of a group representing a character or other entity of data. Note: The characters are transmitted in a sequence over a single line, rather than simultaneously over two or more lines, as in parallel transmission.

¹Actually it is possible to implement a multi-master RS422 or RS485 bus, if bus arbitration is implemented in software, as RS422/RS485 is only specified for single-master configurations.

The following discussion will assume those signal elements to be *binary digits* (or simply *bits*), i.e. signal elements each representing one of two possible states; thus, with serial transmission, one *bit* at a time is transmitted. To put it differently, serial communication protocols are characterised by using one communication path per data transmission direction at most. Moreover, the data frame, i.e. a character or other entity of data, needs to be serialised into a single data stream, as opposed to parallel communication, where data is sent in parallel over multiple links, thus sending data through multiple data streams. Therefore, parallel data transmission extends into the space dimension (by using multiple data paths), whereas serial communication makes use of the time dimension (by sequential transmission).

With respect to the direction of data flow on the transmission path, communication protocols can allow for:

- one-way operation (unidirectional or simplex), having a fixed transmission direction (e.g. RC-5, a popular unidirectional remote control protocol for televisions);
- time interleaved one-way operation (half- or semi-duplex), allowing to speak only in one direction at a time (e.g. I²C, ISO 7816); while others allow for
- two-way (full-duplex) communication by having separate data links for each direction by use of Space Division Duplex (SDD), e.g. RS232C or Fast Ethernet (with respect to separate wires for reception and transmission), or Frequency Division Duplex (FDD) and Time Division Duplex (TDD) which are commonly used together for wireless communication protocols (e.g. WiMAX[6]).

When multiple masters are involved, bus access control becomes necessary in order to avoid multiple masters attempting to transmit on the physical transmission medium at the same time, which would cause data corruption through interference. In order to avoid such *collisions* on a shared medium techniques such as Carrier Sense Multiple Access (CSMA) exist, which basically work by listening on the shared medium for the presence of ongoing transmissions before initiating the transmission.

Another parameter of serial communications is the distinction between *synchronous* operation and *asynchronous* operation. This parameter defines whether there is a separate clock signal used for synchronising the transmitter and receiver clocks (i.e. synchronous); or whether the receiving party is required to detect the protocol timing only by the received data, e.g. by using the signal edges for synchronisation.

Finally, the data frame format is to be defined, which encapsulates the data payload. Data framing is especially important for asynchronous operation, to aid protocol synchronisation.

According to the OSI reference model[38], a communication system can be described by subdividing its functions into seven architectural layers. The layers describe all functions of a system from the physical medium up to the application, with each layer

interacting directly only with adjacent layers. The lowest layer is the *physical layer* and is responsible for all electrical and physical properties of the system. This includes properties such as the voltages used for data encoding on the communication medium (for an electrical medium), as well as the medium itself, i.e. the cabling and the physical connectors. The next layer is the *data link layer*, which provides the functionality for transferring data frames between the entities connected through the communication medium.

For handling the serial communication at the logic level (i.e. the data link layer) a digital component is required. For asynchronous serial communication protocols this is commonly called an Universal Asynchronous Receiver/Transmitter (UART). Furthermore, there are UARTs which also support synchronous communications, in which case they are referred to as Universal Synchronous Asynchronous Receiver/Transmitter (USART). Sometimes, the term Serial Communications Interface (SCI) is used instead of UART.

Serial communication standards such as the RS232C specification which define the physical layer specify other voltage levels as those usually provided by UARTs, as these are commonly implemented as Transistor-Transistor Logic (TTL) or Complementary Metal-Oxide-Semiconductor (CMOS) type digital circuits. In order to convert from the TTL/CMOS levels to the required voltage levels on the medium and provide additional amplification, electrical components called *line drivers* are used; examples for such driver components are the MAX232[34] or the TDA8023[28] Integrated Circuits (IC). When applying the OSI reference model, these components represent a part of the physical layer.

The principal components of hardware UARTs are shift registers, which are used for conversion between the serial and parallel representations of the data frames; data frames to be transmitted are placed into the shift register, and subsequently emitted bit-by-bit on the serial interface; bits received from the serial interface are shifted into the shift-register until the frame is received complete, at which point the CPU is notified.

Some UARTs, if required by the protocol, can handle flow control, e.g. request the other communication endpoint to temporarily stop sending data frames, or honour such requests signalled by the other endpoint. Also, when required by the serial protocol, UARTs can be required to perform error handling, e.g. request data frame retransmission on bad frame reception, or react to such a retransmission request. These tasks are typically handled by UARTs, as they usually require to react within short time periods in the range of bit-lengths in order to avoid data frame loss.

Often, UARTs also contain First In First Out (FIFO) buffers for reducing the frequency at which the CPU has to interact with the UART, by allowing the UART to handle multiple data-frames on its own before requiring intervention from the CPU again. The use of FIFO buffers allows the CPU to react with higher latencies, to avoid data frames to get lost if no flow control was available, or in general for higher data rates.

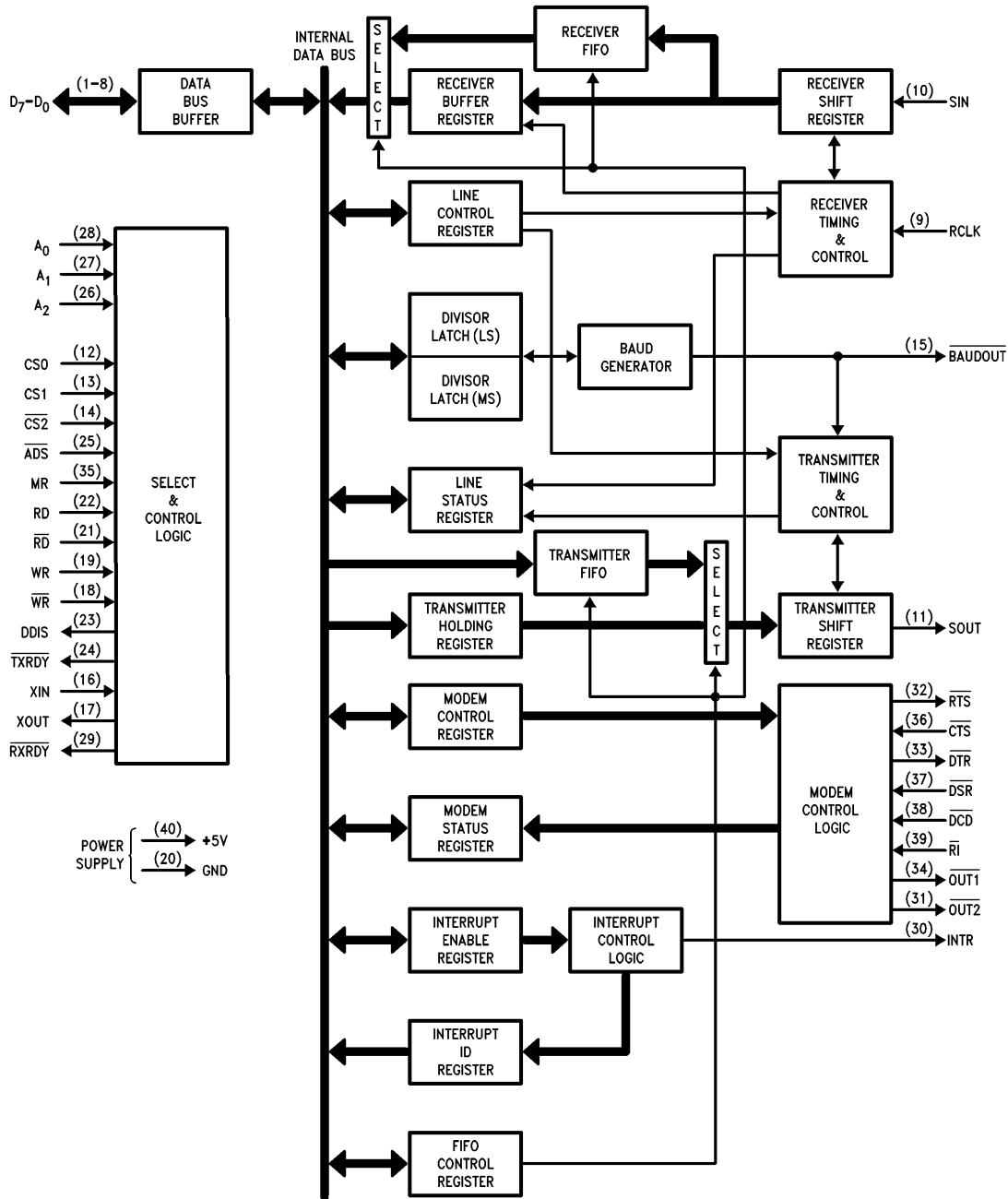


Figure 1.1: PC16550D UART Block diagram (taken from the PC16550D data-sheet[24])

An alternative to FIFOs for reducing the interrupt load is the use of Direct Memory Access (DMA) operations to transfer the data frames between main memory and UART independently of the CPU. With DMA, the CPU initiates the transmission by pointing the UART to the memory location where the data frames to be sent are located, and gets notified (e.g. by interrupts) on completion of the transmission; for reception, the UART is provided with a target location in memory to store the received data frames and notifies the CPU when data has arrived and/or the provided memory buffer has been completely filled. In other words, by using DMA the FIFO which would be part of the UART is moved to main memory. DMA is commonly deployed where high data-rates occur.

The block diagram of a typical standard UART hardware component is shown in figure 1.1 on the preceding page. The UART is controlled by the host CPU through various hardware registers, of which the *transmitter holding register* and the *receiver buffer register* (when the FIFO buffer is not enabled) are the most frequently used ones during actual communication; the data frames to be transmitted are placed in the *transmitter holding register* and then moved into the *transmitter shift register*, which is then shifted by the *transmitter timing & control* unit to produce the serialised bit-pattern of the data-frame on the *SOUT* pin. For data frame reception, the process is reversed and accomplished by sampling the bits on *SIN* and shifting them into the *receiver shift register*, which on completion moves its content to the *receiver buffer register*; from where it can be retrieved by the host CPU, which can poll the UART for the availability of new frames, or alternatively be notified by an interrupt signal. The shown block-diagram shows various other registers and logic units, of which some are used for managing the parameters of the serial communication, such as bit-rate or data-frame format.

Smart-cards and the ISO 7816 Specification

The ISO 7816 specification describes all standardised properties and parameters of smart-cards. Smart-cards, or Integrated Circuit(s) Card (ICC) as they are referred to in the ISO 7816 standards, are usually credit card shaped plastic cards with an embedded microprocessor accessible through metal contact pads located on the surface of the smart-card. The embedded microprocessor controls the access to the data stored on the smart-card, which therefore can be used to securely store passwords or other sensitive information, or even perform cryptographic operations such as signatures or data encryption.

The metal contacts on the ICC are used for communication with the embedded microprocessor and for supplying the necessary power for operation. The signals, as named by the ISO 7816 Specification, and their purpose are as follows:

GND Signal ground and reference voltage for the other signals/supplies.

VCC Primary power supply for the embedded microprocessor.

I/O Semi-duplex serial data signal.

CLK Clock signal; used as time-base for deriving the serial data bit-rate and for providing² a clock signal to the microprocessor.

RST Reset signal; used for activation and deactivation signalling.

VPP Additional programming power supply; required seldom, as ICCs have become less power consuming.

All signals are input to the smart-card, except for the I/O signal which is also output from the smart-card.

The ISO 7816 standard[15] specifies two variants of 1-wire semi-duplex asynchronous³ serial protocols used to communicate with the embedded microprocessor on the smart-card. These protocols are referred to as T=0 and T=1, specifying character-oriented and block-oriented variants of the asynchronous serial protocol respectively.

The physical layer is usually provided by ICC interface components such as the TDA8023 which handles the electrical requirements (e.g. voltage level conversion) for interfacing with smart-cards, including card power-up and power-down. Additionally, the TDA8023 takes over part of the digital handling during the card power-up phase to assist enforcing the ISO 7816 specified protocol timings by deactivating the Smart-card again in case of protocol failure.

For handling the data link layer, specific ISO 7816 UARTs have to be used, as the ISO 7816 specification has requirements with respect to the character framing format and timing which usually are unsupported by standard UARTs. Single-chip solutions exist which embed a micro-controller implementing the ISO 7816 UART and provide a convenient interface to the upper layer, possibly even resulting in an USB interface.

Table 1.1 on the following page shows a comparison between the ISO 7816 protocols with two other serial communication systems—RS232 and I²C.

1.2 Replacing Hardware with Software

The currently available general purpose Central Processing Units (CPU), as used in personal computers or even embedded systems, have enough spare CPU cycles left over, which can be put to use for taking over the tasks traditionally assigned to dedicated

²In the past, the provided clock source directly drove the microprocessor, which allowed for overclocking the smart-card. However, current smart-cards often generate their own clock signal independently of the supplied clock signal, in which case the signal is only used as a time-base.

³Actually, it is not a strictly asynchronous protocol as there is a separate clock signal wire from which the bit length is derived; as will be explained in more detail in section 2.3 on page 17.

Property	ISO 7816 T=0/T=1	RS232	I ² C
Scope	Smart-cards	computer peripherals	on-board electronic components
Topology	point to point	point to point	bus
Master/Slave	single fixed master	both endpoints can be master	multi master, multi slave
I/O signals	1	2	1
Clock signal	yes	no	yes
	asynchronous	asynchronous	synchronous
	half-duplex	full-duplex	half-duplex
Max. bit rate	344086 bit/s	115200 bit/s	3.4 Mbit/s

Table 1.1: Comparison of the serial protocols ISO 7816, RS232, and I²C with respect to their application domain and characteristics

hardware components[21, 22]. This has already been done in the past, with a prominent example being software modems, where the tasks of DSP chips is taken over by the CPU of the host system[19, 37]. Other examples include General Purpose Input Output (GPIO) based drivers for serial protocols such as I²C or Philips RC-5[3, 26] as seen in the Linux kernel[10].

When designing a product with traditional hardware components, this usually leads to hardware designs with specialised microprocessors and/or ICs which serve the need of the specific embedded application. On the other hand, by relaying on software emulated hardware, one can choose a more general microprocessor for multiple products and exploit the associated volume discount. Moreover, each new hardware component causes additional hardware development costs, which can be avoided by reusing known-to-work hardware designs with minor modifications, as opposed to, for instance, complete re-designs incurred when switching to another microprocessor.

In the following text the specific hardware class of UARTs will be discussed whereas some of the arguments used apply to other classes of “software implemented hardware” as well.

Depending on the real-time requirements imposed by the serial protocol, the implementation of the functionality in hardware while retaining responsiveness of the host system becomes a challenging task—especially with multitasking/processing operating systems such as Linux.

The main benefit of implementing an UART in software—apart from saving the costs of an additional hardware component—is the resulting flexibility. Especially when complex protocols such as ISO 7816 are involved, or when the protocol is subject to changes/amendments, or if the functionality is only required during manufacture, or even due to lack of availability of hardware components combined with time-to-market

pressure, implementing the serial protocol logic in software might become a necessity. With components implemented in software it is possible to “upgrade the hardware” by means of a simple software (firmware) upgrade—as opposed to replacing actual electrical components, if possible at all. While this kind of flexibility could also be achieved to some extent by using Field-Programmable Gate Arrays (FPGA) or Complex Programmable Logic Devices (CPLD), which would represent a compromise between a pure hardware implementation and a pure software implementation, a pure software approach might still be more cost effective in many cases.

1.3 Real-Time Correctness

The term *real-time* refers to the requirement of the correctness of an operation to be performed at a certain time; furthermore, a distinction between *hard real-time* and *soft real-time* can be made; the former denotes operation for which meeting the deadline is of crucial importance for the correctness, while the latter refers to operations which may incur performance degradation when missing the deadline, but still being able to recover from the deadline-miss.

When assigning the burden of an UART to the host CPU while it is running a general purpose multi-tasking operating system—especially one lacking the real-time facilities to provide deterministic response times and scheduling latencies—the host is expected to perform other tasks as well besides serving as an UART, i.e. keeping response latency low. This implies that the code providing the UART functionality should block the CPU (and thus keeping the CPU from servicing other processes, such as network or user input processing, while UART operation is ongoing) only as long as absolutely needed in order to provide the required protocol accuracy and reliability.

In preemptive multi-tasking operating systems such as Linux most events to which a real-time response is required are notified to the operating system by triggering hardware interrupts. Interrupts are a mechanism provided by the CPU to react to external signals by suspending the execution of the currently executing code and execute instead a specifically written *interrupt handler*, after whose completion the previously executing task is resumed. Events causing such an interrupt can be the reception of an Ethernet frame requiring a buffer to be read before it gets overwritten by the next frame, or a timer underflow calling for a keep-alive signal to the hardware watchdog. Thus the *interrupt latency*, i.e. the time between the event triggering the interrupt and the interrupt handler being executed, is a key quantity in determining the overall response latency.

The interrupt latency is caused by hardware related issues such as signal propagation delays within logic circuits as well as software issues, a few of which are listed below:

- The signal propagation delay between the event source and the interrupt controller caused by the electronic circuits in-between.

- In case of multiple interrupt sources, the interrupt controller might have to serialise (possibly according to defined priorities) and thus cause delays.
- Interrupts can be disabled by software, usually as a measure to protect currently executing critical code regions.
- The platform dependant interrupt entry routine might need to save the currently executing context and in the case of non-vectorred interrupts determine the actual interrupt number.
- Memory accesses—instruction as well as data—might stall the CPU, especially when the required memory content is not cached and needs to be fetched from slower memory.

The ability of the system to process interrupt events quickly also determines how many events the system can handle within a certain time-frame. Therefore, when it is required to have multiple concurrently active software UARTs running, the number of UART processes is limited by the amount of CPU time slices the software implementation uses up and what the exact real-time requirements are.

Usually the receiving part of the UART is the more demanding part with respect to CPU resources, as it might be necessary to react to events originating outside the control of the host CPU within hard real-time constraints; otherwise, communication protocol failures can occur, e.g. frame reception errors caused by missing the start of the frame or failing to request frame retransmission within the protocol-specified time-window.

Also the task of sampling (and storing) input data and eventually performing error detection is usually more demanding than writing output data. The availability of flow control in the protocol can help lowering the real-time constraints, but it has also the potential to hurt the overall protocol performance; however, the constraint of correctness outweighs that of performance.

Error handling and flow control can cause the sending part to become a receiver as well, as it results in the need to react to external events, such as the error and flow control signalling of the listening party.

1.4 The Challenge

As already mentioned at the beginning of this chapter, the problem tackled by this thesis is to implement a software ISO 7816 UART for the smart-card communication protocol for a general-purpose multi-tasking non-real-time operating system such as Linux. The implementation shall conform to the following requirements:

- Be suitable for the target-platform; and yet sufficiently generic to be portable with little effort to other architectures (portability);

- perform (persistent-)error free communication with the smart-card, if necessary by exploiting error recovery facilities provided by the protocol (reliability);
- conform to the ISO 7816 protocol, e.g. respect intra- and inter-frame timings (correctness);
- support the default ISO 7816 bit rate, i.e. 9600 bps (performance); and
- keep system responsive for other services with hard or soft real-time requirements (responsiveness).

The motivation for replacing a hardware component by its software emulation can be one or more of the following:

- Simplified or smaller PCB design, enabled by omitting the hardware component and associated circuit paths.
- Cost-savings, by omitting the hardware component.
- Flexibility, as the logic behaviour can be modified later, contrary to a traditional hardware component.
- Shorter time-to-market, especially when the hardware component in question is not available (yet) or would have to be developed in the first place.

The choice of targeting a non-real-time operating system is motivated by the fact, that for a given target-platform a real-time operating system may have to be ported to first, whereas a non-real-time operating system is more likely to be readily available.

1.5 Road-Map

The remainder of this thesis is outlined below:

Chapter 2 Starting out with the basic principles of serial data transmission, this chapter provides a presentation of the tasks performed (and the problems involved) by UARTs, with special attention to the ISO 7816 protocol.

Chapter 3 This chapter explores three commonly used generic implementation techniques for sampling and emitting bits and (serial protocol) character frames composed thereof. The properties (requirements, limitations, and strengths) of each algorithm are described and compared to each other.

Chapter 4 In this chapter the target host environment (hardware and software) is presented, for which the software UART implementation was implemented.

The additional requirements imposed on the implementation are described, and the generic techniques from chapter 3 are evaluated according to these requirements.

Finally, an hybrid approach based on the algorithms discussed in chapter 3 is presented.

Chapter 5 This chapter presents measurements showing the behaviour of the interrupt latency under various stress conditions on the target hardware. Based on the measured data, estimates for the theoretically expected achievable bit-sampling rates for the software UART implementation are provided.

Finally, the actual software UART implementation is bench-marked with respect to the achievable bit-rates and the respective error-rates.

Chapter 6 The data presented in chapter 5 is analysed.

Chapter 7 After a short summary of the thesis, conclusions are drawn based on the results from the previous two chapters, and the limitations of the implementation are exposed. Finally, this chapter concludes with suggested areas for future work, where the implementation could be improved.

Appendix A This appendix contains the source code for the software UART implementation presented in this thesis.

Appendix B This appendix contains the source code for the kernel module used for the latency measurements in chapter 5.

Glossary, Bibliography, Colophon Following the appendices, the terms and abbreviations used throughout this thesis are explained, the thesis references are listed, and finally a colophon describing the document preparation completes this thesis.

CHAPTER 2

The UART Problem

As outlined in the introduction, serial communication protocols represent an important class of communication protocols due to their widespread use in computer systems. Hardware components called UARTs are commonly employed to aid in the protocol handling of universal asynchronous serial protocols at the data link layer.

This chapter provides an in-depth discussion of the tasks and problems handled by UARTs.

2.1 The Physical Layer

From the mathematical perspective, the serial-data communication channel can be seen as a medium transporting a continuous bit-stream of binary zeros and ones. However, physically on the medium, the information is transported by means of a waveform; for the case of electrical conductors the voltage waveform over time, $v(t)$, is commonly used to convey information, but one can also use the waveform of other state variables such as electrical current, $i(t)$. The format used for encoding binary bits to the actual physical signal is called *line code*[20].

Line codes can be divided into the following two major categories according to their waveform structure:

Return to zero (RZ) line codes, meaning the waveform signal returns regularly to the zero amplitude (usually for at least half a bit-period), in order to avoid long periods of staying at a non-zero value, which can lead to build-up of DC voltages, and more importantly to potential loss of bit-synchronisation if no separate clock signal is provided for synchronisation.

Non return to zero (NRZ) line codes, which in contrast to RZ coding can result in arbitrarily long sustained non-zero signal levels. In order to cause more level

transitions, variants such as NRZ-I are used where equal subsequent logic values cause a transition of the signal level.

With respect to the polarity of the waveform, the encoding can be distinguished according to the following two interpretations:

Positive-logic defining that the positive or higher signal level is regarded as a logic one, whereas the negative, zero or lower signal state maps to the logic zero value; or

Negative-logic which represents the opposite interpretation of positive-logic.

Together with the distinction whether the signal returns to zero and the polarity interpretation, the actual encoding can be further subdivided into the following incomplete list of line code formats:

Unipolar signalling A zero and a non-zero signal state are used for encoding. For example, an unipolar positive-logic NRZ coding would represent a straightforward mapping of the logic “waveform” alternating the logic states zero and one, to the signal waveform alternating between the zero and the non-zero signal state respectively.

Polar signalling A positive and a negative signal amplitude of equal magnitude is used for representing binary states. When using RZ coding, this leads to three signal levels, which provides a perfectly self-clocked signal, as each bit can be clearly distinguished from each other by the zero signal level.

Manchester signalling As opposed to the previous line codes, the Manchester encoding uses level transitions to represent the logic value; whereas for positive logic a transition from the high to the low signal level represents the binary logic one value, and vice versa. Therefore, Manchester signalling is sometimes called *edge signalling* or *phase signalling*, in contrast to *level signalling*, expressing that the information is not represented by the signal level but rather by the signal transitions.

Figure 2.1 on the next page provides a visual side-by-side comparison of the resulting waveforms resulting from encoding the same binary data with different line codes.

The encoding can be subject to signal inversion due to inverting line drivers, effectively switching between positive-logic and negative-logic. To simplify matters, the following discussion will assume an idle communication state, signifying that no data is being transmitted, represented by a bit-stream consisting of logic ones.

2.2 The Data Frame

The smallest information unit which can be transmitted is usually the binary digit, except for line codes which encode multiple bits to form an atomic data unit called

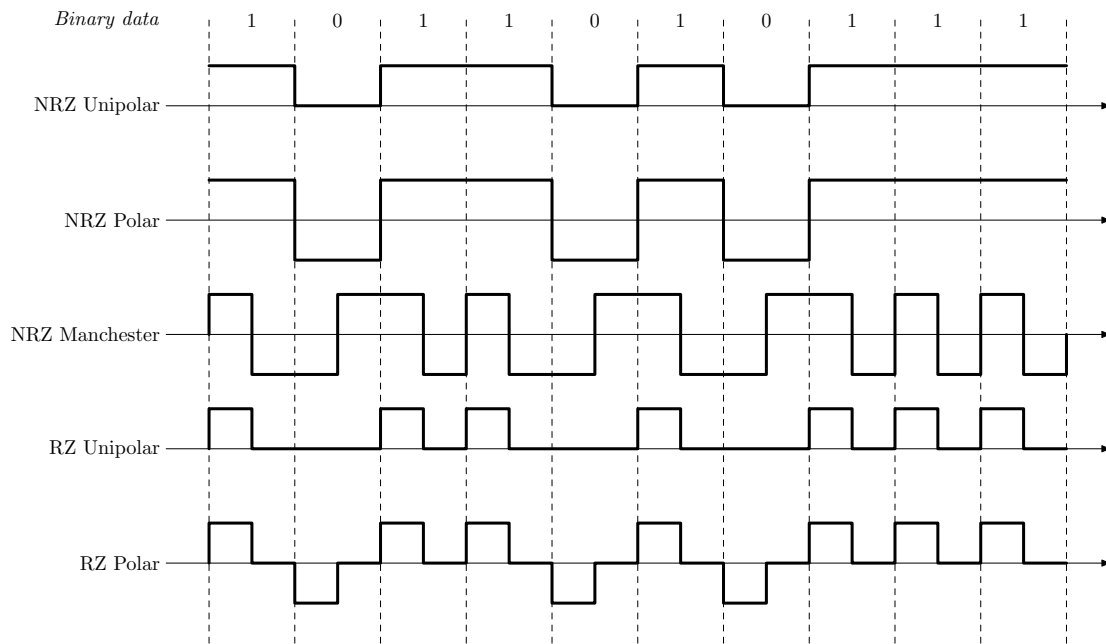


Figure 2.1: Comparison of the resulting signal waveforms of some binary *line codes*

symbol. In order to reduce communication overhead, multiple bits are grouped to form a so-called data frame. The data frame format is part of the protocol specification. Frames can range from sizes as large as a few KiB, e.g. Ethernet frames, to the size of a *character* of only a few bits.

For the purpose of synchronisation, the beginning of a data frame is marked by a start bit(pattern) and terminated by an end bit(pattern). In order to detect transmission errors at the frame level, a simple parity bit or check-sum field can be part of the data frame.

The following discussion will concentrate on the class of universal asynchronous serial communication protocols using a non-return-to-zero level-signalling line code.

The serial line is said to be in *idle* state (which is logically valued one) when no data frames are currently being transmitted. In order to mark the beginning of a new data frame, the start-bit needs to be represented by a state different from the one representing the idle state; consequently, the start-bit is represented by a logic zero which “interrupts” the constant bit-stream of ones representing the idle state.

After the start bit follows the sequence of data bits, in turn followed by an optional parity bit; finally a number of stop bits terminate the data frame. Stop bits are logic ones, just as the idle-state. Therefore, the idle-state can be regarded as a prolongation of the stop bits. Alternatively, the mandated amount of stop bits can be seen as

the minimum amount of time the line should rest in idle state before starting the transmission of the next data frame. An example for a data frame composed of one start bit, eight data bits, one parity bit, and finally two stop bits is shown in figure 2.2 on page 19.

A common short format specification scheme for universal asynchronous data frames is: “*number_of_data_bits* {N|E|O|M|S} *number_of_stop_bits*”. When using this notation, one start bit is assumed implicitly. The single character between the numbers represent the parity modes none, even, odd, mark, and space respectively, which are explained below:

None parity, i.e. the parity bit is omitted.

Even parity bit and data bits are required to contain an *even* number of ones.

Odd parity bit and data bits are required to contain an *odd* number of ones.

Mark parity bit is always *mark*, i.e. one.

Space parity bit is always *space*, i.e. zero.

One of the most frequently used data frame formats is 8N1, specifying one start bit (implicitly), eight data bits, no parity, and finally one stop bit.

Communication Failures

Communication failures are the lack of ability to reliably transmit the data in question; this can have various causes, ranging from noise distorting the signal too much to the communication equipment being unable to handle the amount of information at the required speed.

Errors at the receiving endpoint that can occur during serial communication can be subdivided into the following types:

Overrun errors happen when data frames have to be dropped, i.e. ignored, due to limited resources such as processing power and/or buffer space.

Framing errors emerge when data-frames are received with invalid *framing*, i.e. the start- and stop-pattern.

Parity errors occur when the data-frame contains an invalid parity bit.

The errors listed above are those which can be detected at the receiver, although the detection of those is not guaranteed; for instance, if an even number of data bits are received incorrectly, the parity is unchanged and consequently no parity error will occur even though the data frame has been received incorrectly. Thus, even when no

D	1	2	4	8	16	32	12	20
-----	---	---	---	---	----	----	----	----

Table 2.1: Allowed values for D according to the ISO 7816 specification

F	372	558	744	1116	1488	1860	512	768	1024	1536	2048
max f [MHz]	5	6	8	12	16	20	5	7,5	10	15	20

Table 2.2: Allowed combined values for F and max f according to the ISO 7816 specification

frame error is detected the data frame received might still be corrupted. In order to improve the error detection efficiency, additional measures such as generating Cyclic Redundancy Check (CRC) check-sums over blocks of data frames can be employed.

2.3 The ISO 7816 Serial Transmission Protocol

The ISO 7816 standard defines a semi-duplex asynchronous serial protocol for information exchange over the I/O contact.

Each ISO 7816 data frame is made up of one start bit, 8 data bits, one even parity bit, and at least 2 stop bits¹ (8E2 in short notation) for the T=0 transmission protocol (see figure 2.2 on page 19). The T=1 protocol allows reduction to only one stop bit, i.e. 8E1, as no per-character error signalling is possible.

Two *coding conventions* are defined for the data bits and parity bits, *direct convention* and *inverse convention*; the encoding of the start, guard bits, and the idle-line “bits” remain unaffected. The direct convention defines the high signal state to map to logic ones (i.e. positive-logic), and the least significant data bit to be transmitted first in the data frame, while the inverse convention assigns the high state to logic zeros (i.e. negative logic) and requires the most significant bit to be transmitted first.

The duration of a bit², denoted as Elementary Time Unit (etu), is defined by the ISO 7816 standard as

$$1 \text{ etu} = \frac{F}{D} \frac{1}{f}$$

with f being the clock frequency supplied to the smart-card through the CLK contact, with specified values ranging from 1 MHz to a maximum of 20 MHz and a typical default value of approximately 3,7 MHz. The variables F and D are negotiable parameters called the *clock rate conversion factor* and the *baud rate adjustment factor* respectively, with a default value of 372 for F and 1 for D , resulting in a typical initial communication

¹The standard uses the term *guard-time* to refer to what is referred to in this thesis as stop bits.

²The ISO 7816 specification uses the term *moment* to refer to the duration of a bit at the current bit-rate.

F	$D = 1$	$D = 2$	$D = 4$	$D = 8$	$D = 12$	$D = 16$	$D = 20$	$D = 32$
512	7200	14400	28800	57600	86400	115200	144000	230400
372	9910	19819	39639	79277	118916	158554	198194	317110

Table 2.3: Resulting bit/s for combinations of D and F at frequency $f = 3.6864$ MHz

bit rate near 9600 bits per second. The values allowed by the specification for the parameter D range from 1 to 20 (see table 2.1 on the preceding page). For F and max f the combined values supported by the specification are shown in table 2.2 on the previous page. The resulting bit-rates for a selected set of combinations of D and F at a frequency of about 3.7 MHz are provided in table 2.3.

When a smart-card is *reset*, which includes powering up, the card *answers* with the so-called Answer-To-Reset (ATR). This sequence of maximum 32 character frames is transmitted at the already mentioned default initial bit-rate of about 9600 bits per second using the T=0 protocol. The content of the ATR describes various properties of the smart-card, such as the supported communication protocols (e.g. T=0 or T=1), supported maximum values for D and F , programming voltage and current, etc. If the ATR states that the card supports protocol negotiation, a Protocol-and-Parameters-Selection (PPS) request can be sent to the smart-card, consisting of up to 6 character frames, specifying the preferred protocol and transmission factors D and F . The smart-card then expresses its inability to operate at the requested parameters by not replying at all, affirmatively acknowledges the request by echoing the received PPS sequence, or suggests alternative parameters by sending a PPS sequence containing an alternative proposal of parameters acceptable to the smart-card. Upon reception of the PPS reply (or the lack thereof), the card-terminal can in turn reject the received parameters by resetting the smart-card, thus causing a restart of the PPS negotiation process. After a successful PPS handshake, the subsequent communication with the smart-card is performed with the negotiated values.

For the T=0 transmission protocol, error detection is provided at character level by verifying the parity bit. In case of a detected parity error, this is signalled to the sender by the receiving party during the guard time (i.e. during stop bits) by pulling the signal line to the low state, thus requesting a retransmission of the failed data frame from the sender.

As regards the T=1 protocol, errors are detected and signalled on block frame level, by verifying the check-sum (which can be a simple 1-byte exclusive-or check-sum or a 2-byte CRC check-sum) contained at the end of the block frame, and if necessary requesting the block received containing errors again. The T=0 protocol requires a faster response to reception errors compared to the T=1 protocol since signalling needs to be performed while the first stop bit is still being received, whereas for the T=1 protocol the receiver simply re-requests the same block frame without any upper time

Property	T=0	T=1
Transmission	asynchronous, half-duplex, character oriented	asynchronous, half-duplex, block oriented
Protocol complexity	low	high
Timing critical	high	medium
Error detection	character parity bit	character parity bit, block frame checksum
Error signalling	during character guard time window	by request same block again
Segmentation	not available	available
OSI layer separation	no clear separation	physical, data link, application layer

Table 2.4: Comparison of the ISO 7816 specified T=0 and T=1 protocol variants with regard to characteristic properties

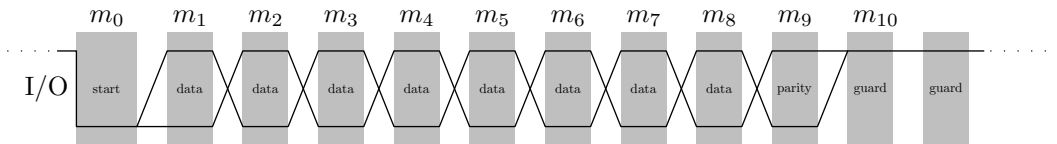


Figure 2.2: ISO 7816 character frame composed of one start-bit, eight data-bits, one parity bit, and two guard-bits

constraint. On the other hand, when only a single frame is corrupted, the T=1 protocol causes the entire block to be retransmitted until the complete block is received error-free (at least according to the check-sum and parity bits), while with T=0 only the bad frame is retransmitted until no parity error is detected.

The actual communication, after card initialisation has been successfully performed, follows a master-slave scheme. The card terminal initiates the communication by sending a series of frames and waits for a response from the ICC. For the T=1 protocol, the T=1 block frame is the smallest sent data unit after which a response from the ICC is required. Whereas for the T=0 protocol, the smallest data unit to require an acknowledgement from the ICC can become as small as a single character frame—if requested so by the ICC.

At the application layer, the communication with the ICC is seen as sending Application Protocol Data Units (APDU) back and forth between card terminal and smart-card; the card terminal sends a command APDU, which is processed by the smart-card, and on completion of the request the card terminal receives a response APDU from the smart-card. The APDU command-response pairs follow a message structure defined in the ISO 7816 specification[14]. For the actual transmission with T=0, the APDUs get

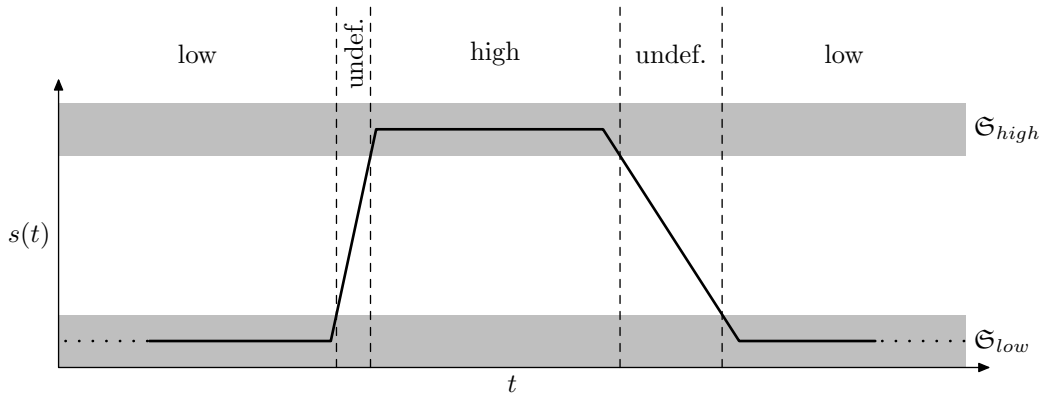


Figure 2.3: Binary data signal $s(t)$ waveform, demonstrating periods of undefined state, resulting from transitions between the low- and high-state representing areas (denoted as \mathfrak{S}_{low} and \mathfrak{S}_{high} respectively)

transformed into Transmission Protocol Data Units (TPDU); this step is required, since the T=0 protocol maps the APDU message structure almost directly to the transmission message structure to avoid size overhead incurred by wrapping the APDU into an enclosing transmission data structure. By contrast, the T=1 protocol does incur that overhead, and wraps the APDU verbatim within a T=1 block-frame structure, splitting the APDU into multiple block-frames if necessary. The T=1 block-frame consists of a 3-byte *prologue field*, an (optional) up to 254-byte *information field*, and is terminated by a 1- or 2-byte *epilogue field*. In figure 5.6 on page 52 an example for a T=1 transmission transaction is shown.

See table 2.4 on the preceding page for a side-by-side comparison of the properties of the T=0 and T=1 protocol variants.

2.4 Sampling the Bits

The data serialised into a bit-stream can be seen as a continuous signal over time, $s(t)$. In order to reconstruct the original (discrete) bit-stream it is necessary to *sample* the continuous signal. Formally, this is expressed by forming the inner product of the signal $s(t)$ with an orthonormal set of functions $\varphi_n(t)$ representing the different bit cells, yielding the orthogonal series coefficients s_n :

$$s_n = \int s(t)\varphi_n(t)dt \quad (2.1)$$

We are restricting the discussion to binary digital signals, which means that s_n in order to be well defined needs to be in one of the two distinct value ranges, \mathfrak{S}_{high} or \mathfrak{S}_{low} , representing the high or low state logic level respectively. Thus in order to

obtain binary values, the resulting coefficients s_n need to be mapped to the bit values d_n according to the rule:

$$d_n = \begin{cases} 0 & \text{if } s_n \in \mathfrak{S}_{low} \\ 1 & \text{if } s_n \in \mathfrak{S}_{high} \end{cases} \quad (2.2)$$

In the case that s_n is neither in one of \mathfrak{S}_{high} or \mathfrak{S}_{low} , the value of d_n is not defined. When dealing with actual hardware an undefined state results in a non-deterministic assignment of the values 0 or 1 to d_n .

The functions used for $\varphi_n(t)$ are usually pulse-shaped functions. In the case of UARTs, the sampling itself takes far less time compared the duration of a bit period, therefore pulse-shapes of infinitesimal width as provided by Dirac's delta function will be assumed for the following discussion:

$$\varphi_n = \delta(nt_p + \xi - t) \quad (2.3)$$

which substituted in (2.1) results in

$$s_n = s(nt_p + \xi) \quad (2.4)$$

Thus assigning s_n the value of the signal $s(t)$ at the time-position $t = nt_p + \xi$.

In actual electronic circuits, the state transition between low and high logic state levels takes a certain amount of time, thus causing $s(t)$ (and therefore d_n) to be of undefined state for a certain amount of time as is shown in figure 2.3 on the preceding page. In order to avoid undefined states, the signal is sampled at the midpoint of the bit interval. Figure 2.4 on the following page shows a bit cell with a sample point at offset ξ from the start of the bit cell (for which $0 < \xi < t_p$ holds), ε representing the time at both ends of the bit-cell the signal needs to stabilise (for which $0 \leq 2\varepsilon < t_p$ holds), and finally t_p being the bit length. In order to reliably sample a bit during its defined-state window, the following condition needs to hold:

$$\varepsilon < \xi < t_p - \varepsilon \quad (2.5)$$

2.4.1 Sampling with Clock Drift

In general, when considering universal asynchronous communications, the communications endpoints are clocked by separate clock sources, causing a drift rate ρ :

$$\rho = \frac{t_{p,receiver}}{t_{p,transmitter}} - 1 \quad (2.6)$$

For perfect clock synchronisation ρ would have a value of 0. By generalising (2.5) to take into account clock drift (2.6) and assuming that both clocks are synchronised at

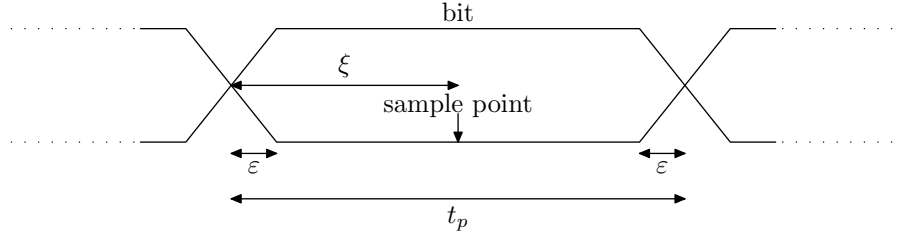


Figure 2.4: Bit cell with sampling point at offset ξ and transition-width ε

$t = 0$ and both endpoints use the same values for ε and t_p in their respective time scales, we get for the n^{th} bit (with the bit index starting at 0):

$$\underbrace{nt_p + \varepsilon}_{\text{transmitter}} < \underbrace{(nt_p + \xi)}_{\text{receiver}}(1 + \rho) < \underbrace{(n + 1)t_p - \varepsilon}_{\text{transmitter}} \quad (2.7)$$

By resolving (2.7) for ρ we get:

$$\frac{\varepsilon - \xi}{nt_p + \xi} < \rho < \frac{(t_p - \varepsilon) - \xi}{nt_p + \xi} \quad (2.8)$$

Requiring (2.8) for all bits up to the N^{th} bit, i.e. for the case of $N + 1$ bit frames with synchronisation at start bit, while assuming that (2.7) still holds:

$$\bigwedge_{n=0}^N \left(\frac{\varepsilon - \xi}{nt_p + \xi} < \rho < \frac{(t_p - \varepsilon) - \xi}{nt_p + \xi} \right) \iff \frac{\varepsilon - \xi}{Nt_p + \xi} < \rho < \frac{(t_p - \varepsilon) - \xi}{Nt_p + \xi} \quad (2.9)$$

If multiple sample points ξ_n within a *test zone* delimited by ξ_α and ξ_β are required, i.e. $(\forall n)(\varepsilon < \xi_\alpha < \xi_n < \xi_\beta < t_p - \varepsilon)$, then (2.9) becomes:

$$\frac{\varepsilon - \xi_\alpha}{Nt_p + \xi_\alpha} < \rho < \frac{(t_p - \varepsilon) - \xi_\beta}{Nt_p + \xi_\beta} \quad (2.10)$$

If either of $Nt_p \rightarrow \infty$, $\varepsilon \rightarrow t_p/2$ or $\xi_\beta - \xi_\alpha \rightarrow t_p - 2\varepsilon$, the allowed range for ρ gets smaller while always including 0, with the limit being only 0 allowed for ρ , i.e. $\max|\rho| \rightarrow 0$.

2.4.2 Sample Point Jitter

Due to technical reasons the actual sampling point is only *close* to the ideal sampling point—especially when the clocks are not synchronised. The deviation of the actual sampling instant ξ' from the ideal sampling instant ξ is commonly called jitter (denoted by σ_ξ), and usually its upper bound is of interest.

Starting from (2.5) given an ideal sampling point ξ one can ask for an upper bound for the jitter, $\sigma_\xi = \max|\xi' - \xi|$, in order to guarantee sampling to occur within stable zones:

$$\varepsilon + \sigma_\xi < \xi < (t_p - \varepsilon) - \sigma_\xi \quad (2.11)$$

This leads to the upper bound for σ_ξ :

$$\sigma_\xi < \min\{\xi - \varepsilon; (t_p - \varepsilon) - \xi\} \quad (2.12)$$

By expressing the sample instant as the absolute distance from the middle of the bit cell, $\lambda = |\xi - t_p/2|$, the following inequality results:

$$\sigma_\xi < \frac{t_p}{2} - \varepsilon - \lambda \quad (2.13)$$

Consequently, by moving the sample point as close as possible to the bit-cell midpoint, thus letting $\lambda \rightarrow 0$, the maximum allowable jitter σ_ξ can be increased. On the other hand, the upper bound for allowed jitter σ_ξ decreases the larger the signal-transition-width ε becomes.

2.4.3 Start Bit Synchronisation

The previous two sections were concerned with sampling in the middle of bit-cells, assuming the knowledge of when the bit-cell actually starts; this knowledge is available if the receiver's and the sender's clock are perfectly synchronised with regard to frequency and phase, and furthermore the bit-cells' boundaries are aligned with the signal edges of the clock signals. In contrast, for asynchronous communication clocks, the clock frequency might be reasonably synchronised, whereas the clock phase might still be out of sync.

In order to synchronise the phases of the receiver clock with the phase of the transmitter clock, it's necessary to detect the leading edge of the start bit as accurately as possible. The detection can be performed by *oversampling*—i.e. sampling the bit cell at multiple instants—which performed at k -fold nominal bit rate, leads to a sample period of t_p/k . When detecting a possible leading edge by having a different logic state at the current sample compared to the previous sample, the edge has occurred some time between these two sample instants. By assuming the event to have occurred exactly in the middle of the two sample instants, this leaves us with an upper bound σ_0 for the uncertainty (neglecting any sample instant jitter, which would add to σ_0):

$$|t_{assumed} - t_{real}| < \sigma_0 = \frac{t_p}{2k} \quad (2.14)$$

When detecting the beginning of the start-bit within an accuracy of σ_0 , this needs to be taken into account when elaborating an upper bound for σ_ξ . For the findings

of the previous sections, the formulae can be adapted by adding σ_0 to the occurrences of ε , which can be interpreted as the uncertainty σ_0 additionally contributing to the transition-width ε . For instance, (2.13) is extended to:

$$\sigma_\xi < \frac{t_p}{2} - \sigma_0 - \varepsilon - \lambda \quad (2.15)$$

which by substituting σ_0 leads to

$$\sigma_\xi < \frac{t_p}{2} \left(1 - \frac{1}{k}\right) - \varepsilon - \lambda \quad (2.16)$$

and finally resolving for k yields

$$k > \frac{1}{1 - 2\frac{\sigma_\xi + \varepsilon + \lambda}{t_p}} \quad (2.17)$$

From the last inequation follows that oversampling is always required, i.e. k needs to be larger than one, even for almost ideal conditions where the term $\sigma_\xi + \varepsilon + \lambda \rightarrow 0$. This result is in accordance with the *Nyquist-Shannon Sampling Theorem*[18, 32, 25] which states that the sampling frequency must be greater than twice the input signal bandwidth (i.e. the data bit-rate in our case) in order to be able to reliably reconstruct the original signal from the sampled data.

CHAPTER 3

Approaching a Solution

At the end of the previous chapter, the formal timing requirements for the correctness of sampling character frames in serial protocol communications were presented.

In this chapter three commonly used algorithms for sampling and emitting bits are presented and discussed with respect to their properties and requirements.

3.1 Dedicated Polling

The simplest method for implementing a software UART is to perform the reception and transmission of the character frames regardless of any other operations the system might have to perform; if necessary, even prevent the system from performing other tasks while the communication is ongoing.

The basic algorithm for character reception is provided in listing 3.2 on page 27; the algorithm for character transmission in listing 3.1 on the following page.

For simplicity, it is assumed that all instructions—except for `usleep()`—have negligible execution times compared to `BIT_PERIOD`. Otherwise, this would lead to continuous shifting of the sampling point with each iteration if no compensation by reducing the time spent in `usleep()` was done. The transmission algorithm signals the value of each bit in the frame for the length of a bit-cell, `BIT_PERIOD`. For the reception, the algorithm works by busy-waiting for the occurrence of a start-bit, i.e. the leading signal edge of the frame, after which the sampling points are performed in the middle of the subsequent bit-cells. If it is necessary to compensate for signal noise, the reception algorithm can be modified to sample multiple times during a bit-cell, and use a majority vote to reduce to a resulting bit-value.

The shown algorithms are suitable for environments which provide real-time properties, e.g. exclusive CPU access by disabling interrupts and preemption in order to guarantee the above-mentioned timing assumption, or when a dedicated microprocessor is available solely for the task of handling serial communication.

```
void transmit_frame(unsigned frame)
{
    set_tx(0); // signal start-bit
    usleep(BIT_PERIOD); // wait for the duration of one bit-cell

    // transmit the DBITS data bits
    for(unsigned TxBitCnt = 0; TxBitCnt < DBITS; ++TxBitCnt)
    {
        set_tx(frame & 1); // put least significant frame bit on the wire
        frame >>= 1; // shift frame by one bit
        usleep(BIT_PERIOD);
    }

    set_tx(1); // signal stop bit
    usleep(BIT_PERIOD);
}
```

Listing 3.1: Dedicated polling transmitter implementation

The given implementation is rather simple, since it is intended for exclusive execution, and therefore the state is kept implicitly in the local execution context. This makes the implementation suitable for single channel half-duplex operation, where at most one of the two routines is executing at the same time.

On the other hand, when full-duplex operation is required, or even multiple serial communication paths need to be served concurrently, the implementation becomes increasingly complex since multiple threads need to be intertwined into a single function. For the purpose of intertwining multiple threads of control, techniques such as Asynchronous Software Thread Integration (ASTI) can be used[1].

Since this UART algorithm requires to be executed exclusively by the CPU, this causes the algorithms' routines to be cache-resident with a high probability and additionally not subject to preemption; this leads to highly time-deterministic and fast instruction execution. Therefore, this implementation variant represents the highest possible bit-rates and most accurate signal timings that can be achieved with a software UART implementation on the given platform; this good performance is achieved at the cost of reduced responsiveness of the rest of the system during execution of the UART routines.

To summarise, the *dedicated polling* technique presented in this section is simple in its implementation (for single-channel half-duplex operation), and shows good performance at the same time. On the other hand, the responsiveness of the system is effectively disabled for other services while transmission is on-going (including start-bit detection) because of the required execution-exclusivity of the UART routines; especially when frames can arrive at any time, this would require the system to be busy-waiting for a start-bit most of the time, effectively causing the system to be unusable for other tasks beyond serving the serial communication channel.

```

int receive_frame(unsigned &frame)
{
    frame = 0; // clear receive buffer

    while(get_tx() != 0) {}; // busy waiting for start-bit

    usleep(BIT_PERIOD/2); // wait until midpoint of start-bit

    if(get_tx() != 1) // verify start-bit
        return -1; // start-bit error

    usleep(BIT_PERIOD);

    // sample the DBITS data bits
    for(unsigned TxBitCnt = 0; TxBitCnt < DBITS; ++TxBitCnt)
    {
        frame |= get_tx() << TxBitCnt;
        usleep(BIT_PERIOD);
    }

    if(get_tx() != 1) // verify stop bit
        return -2; // stop bit error
}

```

Listing 3.2: Dedicated polling receiver implementation

3.2 Timer Interrupt Triggered Processing

This approach mimics the usual hardware implementation of UARTs which consists of executing a process triggered at each tick of a clock running at a multiple frequency of the communication bit-rate. Let k denote the multiplier by which the UART's clock runs faster than the nominal bit-rate. Standard hardware UART implementations such as the PC16550D[24] have typical values of up to $k = 16$ in order to achieve the required start-bit synchronisation timing accuracy (see section 2.4.3 on page 23).

The task of transmitting a character frame is straightforward. In terms of a simplified description, the character frame to be transmitted is placed in the transmit register, including the start-bit. The end of the transmit register, where the start-bit is located, is connected to the transmit line, therefore causing the transmit line to always get the value of the bit currently being at the initial start-bit-position. On every k^{th} clock tick (relative to the clock tick when the transmit register was loaded to ensure the proper start-bit length), a logic one is shifted into the register at the opposite end to the one connected to the transmit line, causing all bits previously in the register to be shifted towards the transmit line. Finally, when all bits have been transmitted, the transmit register is consequently filled with logic ones and stays that way (emitting logic ones) until a new frame is to be transmitted.

In contrast to the *dedicated polling* technique from the previous section, the timer interrupt triggered process needs to keep the state of the character frame transmission

or reception across process invocations. For the process of frame transmission the state is simply the transmit register which gets shifted on each process activation (assuming the process being activated only every k^{th} clock tick), and additionally a bit-count register for detecting when the character frame transmission has finished. Listing 3.3 on the facing page shows such a simple implementation in VHSIC Hardware Description Language (VHDL) for the frame transmission process. The CLK_DIV_K signal denotes the clock signal divided by k . The code requires simultaneously the register TxReg to be loaded and TxStart to be set at the rising edge of the CLK_DIV_K. The end of frame transmission, and thus the readiness for the next character frame to be loaded into the transmit register, is signalled by TxReady getting set to one.

Data reception is more complicated due to start-bit detection and error handling. As a result, the state to be kept across process activations is more complex as well. See listing 3.4 on page 30 for a simple implementation in VHDL for the receive process together with the related finite state machine depicted in figure 3.1 on page 31. In the implementation, the constant BIT_TICKS denotes the multiplier k and DBITS the number of data bits. The actual sampling occurs at the middle of each bit cell.

As mentioned for the *dedicated polling* algorithm already, in order to compensate for line noise, the receiving process can take additional samples—i.e. oversample—around the midpoint of each bit cell at clock-tick distance, and use a simple majority vote over the uneven number of samples in order to decide the bit value.

When implementing a software UART with this design pattern in a multitasking environment, the VHDL process would be mapped to a routine which would be called at regular intervals—for instance an interrupt service routine—being triggered by a periodic timer interrupt.

As an optimisation, the timer interrupt frequency could be adapted; when not oversampling, only a timer interrupt frequency matching the nominal bit-rate is required. Therefore, only while waiting for the start-bit a sampling frequency higher than the nominal bit-rate is required. On the other hand, if character frame reception can occur at any time, this would mean that the system is waiting most of the time for a start-bit at the higher interrupt frequency, which would limit the usefulness of this optimisation.

Since interrupts are involved for triggering the bit sampling, the interrupt latency becomes an issue, as the sampling point depends on it. The relation between the nominal bit-rate and the resulting constraints on the interrupt latency for error-free frame reception have been discussed in section 2.4 on page 20.

By using timer interrupts for activating the UART routines, the CPU can perform other tasks between the interrupt service routine calls and thus, compared to the previous polling algorithm, allowing the system to be responsive to a certain degree while the UART routines are active. On the other hand, the interrupt handling overhead, the interrupt latency, and the need to oversample cause this algorithm to perform significantly worse than the *dedicated polling* algorithm.

In summary, this timer interrupt based algorithm solves the responsiveness problem

```

signal CLK_DIV_K, Tx, TxReady: std_logic;
signal TxReg, TxCnt: std_logic_vector(BITS downto 0);

Tx      <= TxReg(0); — connect LSB of TxReg to Tx
TxReady <= TxCnt(0); — connect LSB of TxCnt to TxReady

process (CLK_DIV_K, TxStart)
begin
  if TxStart = '1' then
    TxStart <= '0'; — reset flag
    TxCnt <= (others => '0'); — reset bit-count shift-register

    elsif rising_edge(CLK_DIV_K) then
      TxReg <= '1' & RxReg (RxReg'high downto 1); — shift register
      TxCnt <= '1' & RxCnt (RxReg'high downto 1); — shift register
    end if;
end process;

```

Listing 3.3: Timer interrupt triggered transmitter

of the *dedicated polling* algorithm at the cost of additional complexity and significantly poorer performance, i.e. lower achievable bit-rate.

3.3 Edge Interrupt Triggered Bit Detection

The primary problem of the previous two algorithms is the start-bit detection, which consumes considerable amounts of CPU time, and requires a higher sampling rate than for the remaining part of the character frame. The algorithm presented in this section tries to improve the start-bit detection. It handles only the character frame reception, and must therefore be combined with one of the frame transmission routines presented in the previous sections.

This variant for the character reception implementation tries to improve the start-bit detection by having the hardware detect signal edges by means of interrupts triggered by signal transitions; thus requiring that the hardware supports this mode of operation. Additionally a timer is required which allows to associate signal edges with a time-stamp; the timer is also required to be able to trigger an interrupt on timeout, in order to detect the end of frame since the trailing stop bit(s) are set to high state just as the idle state and therefore no terminating signal transition occurs at the end of the frame.

Listing 3.5 on page 33 shows a possible implementation of this sampling scheme. This algorithm works by “sampling” the signal transitions, rather than the bit-cell midpoints. The `rx_handler()` is to be registered as the interrupt handler for the signal transitions as well as the timer timeout triggered interrupts. At each detected signal transition the preceding bits since the previous transition are known to be of the previous signal value and therefore updated accordingly in the receive variable holding the current frame.

```
signal CLK, Rx, RxReady: std_logic;
signal RxReg: std_logic_vector(DBITS downto 0);

process (CLK)
  variable RxDelay : integer range 0 to (BIT_TICKS - 1) := 0;
  variable RxState : RxStates := RxStateIdle;
begin
  if rising_edge(CLK) then
    if RxReady = '1' and RxState /= RxStateIdle then
      RxReady <= '0'; -- RxReady only held up while in RxStateIdle
    end if;

    if RxDelay /= 0 then -- countdown RxDelay until 0
      RxDelay := RxDelay - 1;
    end if;

    case RxState is
      when RxStateIdle =>
        if Rx = '0' then
          RxState := RxStatePreStart;
          RxBitCnt <= 0;
          RxDelay := BIT_TICKS/2-1; -- wait for start-bit midpoint
        end if;

        when RxStatePreStart =>
          if RxDelay = 0 then RxState := RxStateStart; end if;

        when RxStateStart =>
          if Rx = '1' then -- start-bit failure
            RxState := RxStateIdle; -- back to square 1
          else
            RxReg <= (others => '0'); -- clear register
            RxDelay := BIT_TICKS-1; -- wait for next bit midpoint
            RxState := RxStatePreBit;
          end if;

        when RxStatePreBit =>
          if RxDelay = 0 then RxState := RxStateBit; end if;

        when RxStateBit =>
          RxDelay := BIT_TICKS-1;
          RxReg <= Rx & RxReg (RxReg'high downto 1); -- shift register
          if (RxBitCnt = DBITS-1) then
            RxState := RxStatePreStop;
          else
            RxBitCnt <= RxBitCnt + 1;
          end if;

        when RxStatePreStop =>
          if RxDelay = 0 then RxState := RxStateStop; end if;

        when RxStateStop =>
          if Rx = '1' then -- only signal reception when valid stop bit
            RxReady <= '1';
          end if;
          RxState := RxStateIdle;
        end case;
      end if;
    end process;
```

Listing 3.4: Timer interrupt triggered receiver

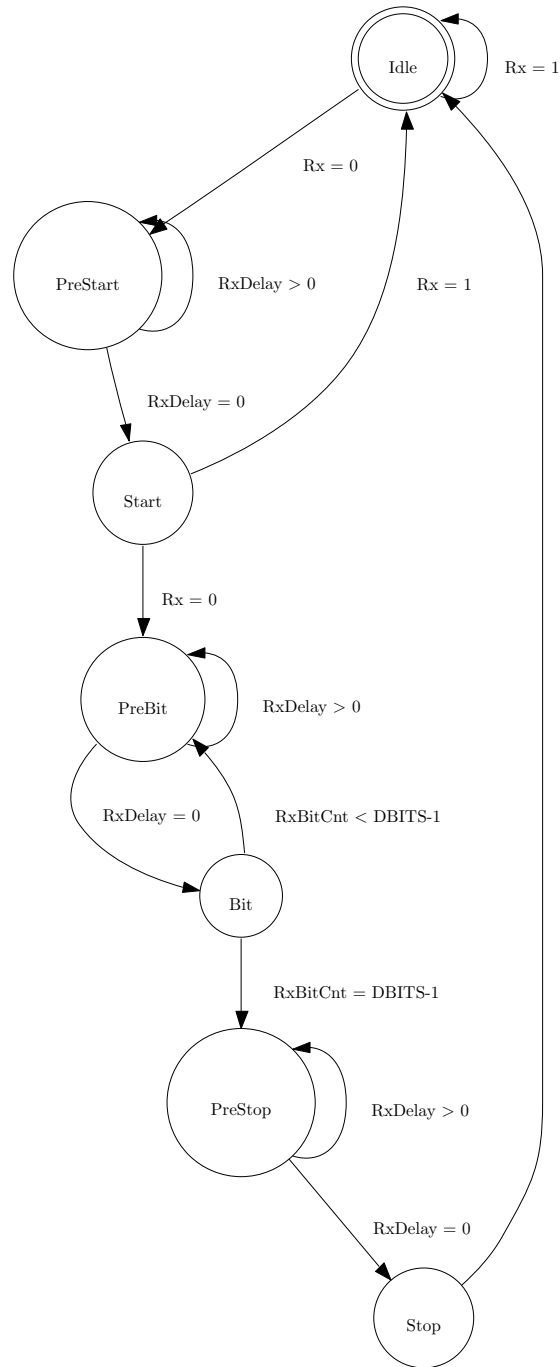


Figure 3.1: Timer interrupt triggered receiver state machine

Finally, when the timeout interrupt is triggered, the frame reception is completed and the algorithm returns to waiting for the next data frame. In order to reliably associate a detected signal edge to the corresponding bit-cell boundary, the interrupt jitter needs small than $\frac{t_p}{2} - \varepsilon$ (see section 2.4.2 on page 22).

3.4 Summary

This chapter presented three common implementation schemes for software UARTs; all of which had advantages and disadvantages which are concisely visualised below:

Dedicated Polling	Timer Interrupt Triggered	Edge Interrupt Triggered
Advantages		
<ul style="list-style-type: none"> • simple implementation • state kept implicitly in execution context • highest possible bit-rates achievable 	<ul style="list-style-type: none"> • execution takes place only at sample point 	<ul style="list-style-type: none"> • protocol handler routines gets executed the least possible
Disadvantages		
<ul style="list-style-type: none"> • requires exclusive execution or real-time facility from the underlying OS • busy waiting 	<ul style="list-style-type: none"> • state needs to be managed explicitly • requires bounded interrupt delay jitter 	<ul style="list-style-type: none"> • state needs to be managed explicitly • requires bounded interrupt delay jitter
Characteristic Properties		
	<ul style="list-style-type: none"> • requires periodic timer interrupt 	<ul style="list-style-type: none"> • applicable only to data reception • requires edge sensitive hardware interrupt • requires readable timeout timer for timekeeping

However, none of the presented variants meets all of the requirements stated in section 1.4 on page 10 for the target implementation; the shortcomings of each of the presented algorithms with respect to the stated requirements are discussed in more detail in section 4.2.1 on page 38 and furthermore a novel algorithm is presented in chapter 4 on page 35.

```

void rx_handler(void)
{
    static unsigned frame; // buffer for constructing currently receiving frame

    static bool idle_state = true; // current state selector
    static unsigned last_rx = 0; // previous states' get_rx() value
    static unsigned BitCnt = 0; // last states' bit index

    if(idle_state)
    {
        if(get_rx() == 0) // verify start-bit
        {
            last_rx = 0; // start-bit

            timer_set_alarm(BIT2TICKS(DBITS+2)); // set timeout on end of stop-bit
            timer_restart(); // start timer from 0

            frame = 0; // clear buffer
            BitCnt = 0; // reset bit counter

            idle_state = false; // leave idle state
        }
    }
    else
    { // in frame state
        const unsigned NewBitCnt = TICKS2BIT(get_timer()); // rounding function

        while(BitCnt < NewBitCnt) { // set bit indices in [BitCnt, NewBitCnt)
            frame |= last_rx << BitCnt; // to last_rx
            ++BitCnt;
        }

        last_rx = get_rx(); // update state

        if(BitCnt == DBITS) // i.e. timer alarm caused this handler execution
        {
            idle_state = true; // back to idle state

            if(last_rx) // verify stop bit
                // signal reception of masked data bits
                rx_complete(frame >> 1 & (1<<DBITS) - 1);
        }
    }
}

```

Listing 3.5: Edge interrupt triggered receiver implementation

CHAPTER 4

The Implementation

After having presented various possible generic implementation techniques for simple serial protocol implementations in chapter 3, this chapter will concentrate on the actual implementation for the T=0/T=1 protocol specified in ISO 7816, after having described the target host environment.

4.1 The Host Environment

4.1.1 The Hardware

The goal was to implement an ISO 7816 software UART using GPIO pins on a Cirrus EP9301[4] based embedded system, see figure 4.1 on the next page.

The Cirrus EP9301 is a 32-bit RISC 166MHz ARM920T processor with a 5-stage pipeline, a Memory Management Unit (MMU), and a non-unified 32 KiB cache. The core-internal memory system follows the Harvard architecture, i.e. separating the instruction paths from the data paths to allow parallel access to instructions and data. Therefore, the Cache is divided into separate 16 KiB instruction and data caches (programmable as write-through or write-back) with a cache-line length of 32 bytes, resulting in 512 cache-lines for each cache. Also the Translation Look-aside Buffer (TLB) are a divided, providing 64 entries for each of the data TLB and the instruction TLB. The TLBs as well as the caches allow for individual lock-down of their entries, in order to keep selected entries from being replaced.

The EP9301 provides various GPIO pins which can be programmed to trigger hardware interrupts, these can be configured either edge or level sensitive.

Furthermore, the EP9301 features four hardware timers of which the most precise is the 40-bit upward-counting *time stamp debug timer*, being clocked directly by the 14.7456MHz clock source and divided by 15 to yield 983040 clock ticks per second. The time stamp debug timer is free-running, i.e. automatically wraps over on overflow; it

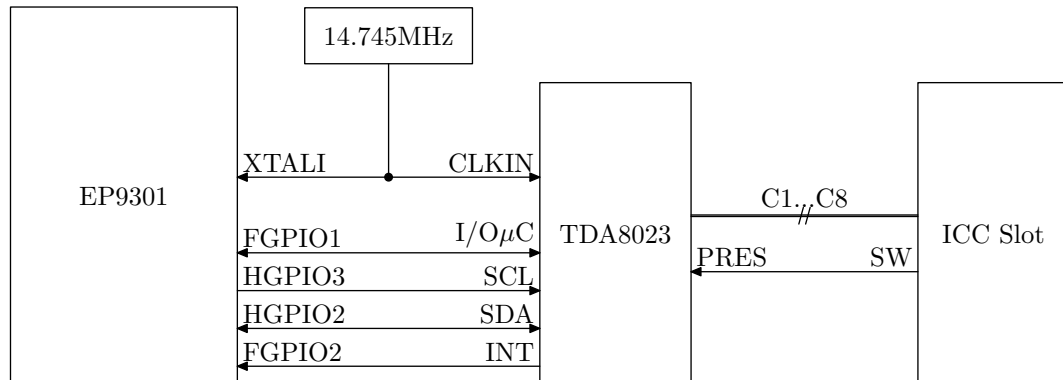


Figure 4.1: Block diagram of card terminal embedded system

can be enabled and disabled, which clears it back to zero; however, it cannot be set to trigger interrupts.

The only component apart from the contacting unit used for interfacing with the smart-card is the TDA8023 IC[28], the *smart-card analog interface* which handles the physical layer—i.e. power supply, protection and power-up/down functions. The TDA8023 is provided with the same 14.7456 MHz clock source as the CPU in order to avoid clock drifts. The CPU is connected to the TDA8023 over an I²C bus in addition to the passed-through smart-card I/O signal, and an interrupt line for signalling events such as card insertion/removal.

4.1.2 The Software

The software UART has been developed as a kernel device driver for the 2.4.x branch of the Linux kernel[10], a free Unix-like operating system kernel. The specific kernel version used was a vanilla 2.4.26 version of Linux, with ARM specific patches[30] and EP9301 specific patches from Cirrus applied.

A by-product of the software UART implementation process was the creation of a generic ISO 7816 T=0/T=1 protocol stack and card terminal device driver framework for Linux. The kernel/user-space interface is accomplished by providing character devices on which the standard I/O Portable Operating System Interface (POSIX) system calls are to be applied, which are subsequently mapped to smart-card operations. The semantics for the character device based smart-card user-space interface have been formalised in the UX/SC specification[35].

In order to register a card terminal with the protocol stack, a low-level device driver has to be written, which implements at least the following methods:

```
int start(struct ifd_device *dev, int warm, ifd_device_class_t class);
int stop(struct ifd_device *dev);
```

```

int frames_tx(struct ifd_device *dev, const u8 frames[], unsigned len, int more)
{
    assert(!more);
    // assume more is always false, thus frames[] contains all frames to send

    // send all frames
    for(unsigned idx = 0; idx < len; ++idx)
        send_frame_to_icc(frames[idx]);

    // wait for answer and pass up frames one by one when received
    do {
        u8 frame;

        if(receive_frame_from_icc_with_timeout(&frame, TIMEOUT))
            break; // leave loop on timeout

        // report received frame to protocol stack
        // ifd_frame_rx() returns non-zero when further frames are expected
    } while(ifd_frame_rx(frame));

    // either timeout occurred or end of message signalled by protocol stack

    // signal timeout/acknowledge end of reception to protocol stack
    ifd_notify(dev, IFD_EVENT_RX_TIMEOUT);
}

```

Listing 4.1: Simplified frames_tx implementation

```

int set_rate(struct ifd_device *dev, unsigned F, unsigned D, int try);
int frames_tx(struct ifd_device *dev, const u8 frames[], unsigned len, int more);

```

The functions `start()`, `stop()` and `set_rate()` are used for session initialisation and finalisation. Function `frame_tx()` is the primary entry point for the protocol stack for initiating a transmission cycle, consisting of a request being transmitted to the ICC and a reply being received. The reply is signalled asynchronous frame by frame to the protocol stack concluded by an end of frame notification by means of the API functions `ifd_frame_rx()` and `ifd_notify(,IFD_EVENT_RX_TIMEOUT)` respectively.

Listing 4.1 shows a simplified implementation of the `frames_tx()` method. This transaction scheme exploits the property of smart-card communication that the ICC only answers when asked, since the ICC is a communication slave endpoint; therefore, it is only needed to expect character frames from the ICC within defined time-windows, and thus the reception routine is only activated for limited time windows.

4.2 The Driver

4.2.1 Constraints

Apart from having to fit into the described software and hardware host environment, the software UART implementation was required to meet the following requirements (as already stated in section 1.4 on page 10):

- Suitable for the target-platform; and yet generic enough to be portable with little effort to other architectures (portability);
- perform (persistent-)error free communication with the smart-card, if necessary by exploiting error recovery facilities provided by the protocol (reliability);
- conform to the ISO 7816 protocol (correctness);
- support bit rate of at least 9600 bps (performance); and
- keep system responsive for other services (responsiveness).

When applying the techniques presented in chapter 3 to the problem, the following issues arose:

dedicated polling Real-time context created by disabling interrupts. Using third party real-time patches to Linux would have required major modification to the standard Linux source tree; in addition, at the time writing there was no support for the used platform yet. For the purpose of transmission tolerable, since each character takes up only about 1.2 ms at the default rate, and scheduling could be allowed between each frame sent to the ICC. As for frame reception disabling interrupts is not acceptable, since for the time waiting for the start bit of the first reply frame, and also the time between each subsequent frame, can extend to a couple of seconds; to make that worse, the protocol allows for extending the time the ICC has to respond, by using so-called *waiting time extension*, which can lead to several minutes spent waiting for a start-bit.

Since this variant has the least overhead, it also provides the best performance, at the cost of responsiveness.

timer interrupt triggered This technique effectively avoids the problem of spending large amounts of time with disabled interrupts busy waiting for the next start bit to show up. But, it requires the interrupt latency to be small enough compared with the sampling interval. Also the sampling needs to be a sufficient multiple of the bit rate, in order to detect the start bit with enough precision to be able to sample subsequent bit cells at the midpoint. The overhead incurred by using high

frequency interrupts can easily grow beyond the time spent within the interrupt handler, effectively causing the system to use up all CPU time for serving the frequent interrupts and maybe even miss interrupts.

edge interrupt triggered bit detection This variant would cause less frequent interrupts than the timer interrupt triggered variant, but still relies on low and constant latency for proper operation. Furthermore, the T=0 protocol error signalling requires low latency jitter in order to meet the timing constraints. Requires hardware to be able to detect negative as well as positive signal edges.

Therefore, satisfying a subset of these constraints could be easily accomplished by using one of the approaches presented in chapter 3, satisfying all of them at once, given the host environment at hand, requires a new approach.

4.2.2 The Hybrid Approach

The approach taken to fulfil the requirements was to combine ideas into the following implementation:

- Dedicated polling for character transmission
- Hybrid algorithm based upon edge detection for start bit and subsequent auto-aligning dedicated bit sampling.

By auto-aligning, the act of compensating for the start-bit detection delay, caused by the interrupt latency, is meant, by means of waiting for the next bit-change within the frame; in the best case this will occur with the first data bit after the start-bit, in the worst case this happens with the stop-bit. See listing 4.2 on the following page and figure 4.2 on page 41. This approach leads to the following combined properties:

- responsiveness, due to asynchronous start-bit detection by use of negative edge triggered interrupt
- good performance and timing accuracy thanks to dedicated polling

However, the unsynchronised sampling point offset ξ_0 needs to stay within ξ_α and ξ_β (0.2etu and 0.8etu respectively for ISO 7816) in order to guarantee correctness; otherwise the timer could be aligned on signal edge detection. For the lower boundary, ξ_α , this can be avoided by delaying a certain amount of time, `LATENCY_OFFSET`, at entry of the interrupt handler based on the absolute minimum interrupt delay possible and the current bit-rate dependent ξ_α value; This leaves us with a timing constraint for the correctness of the algorithm, requiring the interrupt latency to be less than $\xi_\beta + \text{LATENCY_OFFSET}$.

```
void rx_edge_irq_handler(void) // negative edge interrupt handler
{
    // assume all interrupts are disabled, i.e. code below runs exclusively

    if(get_rx () != 0) // verify start bit
        return; // latency too high or glitch — ignore

    udelay(LATENCY_OFFSET); // move initial sampling point

    if(get_rx () != 0) // re-check start bit
        return; // latency too high or glitch — ignore

    timer_restart_at(0); // restart timer at initial offset

    unsigned curr_frame = 0; // frame format: SDDDDDDDDPG
    unsigned idx;

    // unsynchronised sampling time base

    // scan for first edge
    for(idx = 0; idx < 1+8+1+1; ++idx)
    {
        while(timer_get() <= idx*ETU + ETU)
            if(get_rx() == 1) // edge occurred, so leave while loop
                goto edge_detected;
    }

edge_detected:
    // bit 0 up to bit idx are cleared
    timer_restart_at(idx*ETU + ETU); // align timer value

    // synchronised sampling time base

    // since we now know where the bit starts, we can sample at mid-point
    // next bit to be sampled is at index idx+1
    for(++idx; idx < 1+8+1+1; ++idx)
    {
        int curr_bit = 0;

        // delay until in the middle of the bit cell
        while(timer_get() <= idx*ETU + ETU/2) {}

        curr_bit = get_rx(); // now sample it

        curr_frame |= curr_bit << idx; // insert bit into frame
    }

    // curr_frame now contains complete frame including framing

    // verify framing/parity, do error signalling if needed,
    // submit curr_frame to reception buffer
    ...

    // interrupts will be enabled again outside the handler
    return;
}
```

Listing 4.2: Simplified hybrid sampling implementation

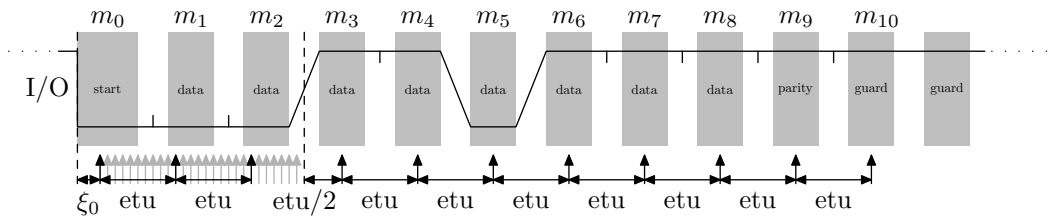


Figure 4.2: Hybrid sampling algorithm: Signal waveform with sample positions shown as arrows; the numerous light arrows depict the polling phase leading to the detection (indicated by the dashed line) of the first low-high signal transition, at which the algorithm is synchronised, and consequently switches over to sampling at the subsequent bit-cell midpoints; ξ_o denotes the leading edge detection delay caused by interrupt latency

4.2.3 Actual Implementation

The actual driver implemented according to the algorithm presented in section 4.2.2 on page 39 to provide a proof of concept for this thesis is reproduced in appendix A on page 67.

Platform Dependencies

The driver implementation requires the following capabilities from the hardware, which were assumed to be available on most platforms:

- GPIO Port for the I/O data line.
- (Negative) signal-edge triggered I/O data interrupt.
- Timer counter for time-keeping with the resolution required the targeted bit-rates.
- Smart-card controller interfaced with I²C and interrupt line, and clock source synchronised with time counter.

The driver was implemented with the intent of being easily ported to other platforms, nevertheless, due to the low-level nature of the task and the lack of hardware abstraction facilities in the Linux kernel at this level, it is still necessary to make modifications beyond simply changing a few constants to port the driver. Therefore, in order to port the given implementation to other platforms, the following changes would be required, assuming the same smart-card controller interface chip was used:

- Modify the functions controlling the interrupt handling used for communicating with the smart-card controller: `tda8023tt_ctl_irq_on ()`, `tda8023tt_ctl_irq_off ()`, and `tda8023tt_ctl_irq_clear ()`.

- Adapt the functions for managing the I/O edge interrupt: `tda8023tt_io_irq_on()`, `tda8023tt_io_irq_off()`, and `tda8023tt_io_irq_clear()`.
- Adapt the functions for initialising and managing the state of the I/O data line: `_iouc_init()`, `_iouc_get()`, and `_iouc_set`.
- Adapt the functions for retrieving and managing the hardware timer counter value: `ep93xx_timer_init()`, `ep93xx_timer_reset()`, and `ep93xx_timer_reset()`.
- Adapting the integer arithmetic to convert from the timer time-base to the current bit-rate time-base expressed in half-ETUs in the functions: `ep93xx_ticks2timer`, `ep93xx_timer2ticks`, and `tda8023tt_timer_setup()`.

Limitations

As the primary goal of this thesis is to provide a proof of concept, rather than to implement a production-grade certified ISO 7816 software UART, the presented implementation suffers from deliberately chosen limitations:

- As the smart-cards are assumed to show much better timings than required by ISO 7816, the implementation behaves as if compensation delay `LATENCY_OFFSET` was always 0.
- Only the protocol timings absolutely required for allowing the basic character reception and transmission to work at all are (approximately) implemented.
- The `DIVCLK` parameter supports only the dividers 4 and 5, and can only be selected at compile time.
- The implementation announces only support for cards providing (at least) class A (5 Volt) operation in order to workaround potential reentrancy issues with the card initialisation routines.
- Signalling badly received character frames, thus requesting character frame re-transmission, and subsequently re-sampling the resent frame for the `T=0` protocol is not implemented.

Eliminating these deficiencies and making further improvements is reserved for future work, which is addressed in section 7.1 on page 64.

CHAPTER 5

Experiments in Latency and Time

In order to evaluate the presented algorithm for practical applications, it is necessary to acquire empirical data. This chapter presents measurements performed on the target system described in the previous chapter related to the devised software UART implementation.

5.1 Interrupt Latency

The interrupt latency is one of the main limiting factors for the performance of the software UART. Therefore, it is of interest to determine its magnitude under various conditions in order to estimate what performance to expect from a software UART implementation based on interrupt-based signal-edge detection.

The interrupt latency, i.e. the time between the interrupt being triggered and the interrupt service routine being executed, was measured using the Linux kernel module listed in appendix B on page 91. The basic idea was to use a periodic timer to generate interrupts, and to determine the time taken to reach the interrupt service routine—i.e. the interrupt latency—with the help of a second high precision timer.

By programming the TC3 timer with a count-down frequency of 1993.9 Hz to periodically trigger on counter underflow, interrupt trigger frequencies f between 1 Hz and 1 kHz could be achieved. The interrupt latency was determined by measuring with the free-running TC4, which is counting up with a frequency of 983040 Hz, the time until TC3 changes its value again; this all happens in the interrupt service routine with interrupts disabled, thus causing interrupts being disabled by this module for up to approximately 1 ms. This procedure allows for detecting the latency with a precision of about 1 μ s. The interrupt service routine's assembly text size is 160 bytes, which would take up five 32-byte cache-lines if cached as a whole.

For each trigger frequency, f , of 1 Hz, 10 Hz, 100 Hz, and 1 kHz, the latencies, t , were measured and collected for n consecutive interrupt triggers. Due to the discrete

resolution of the TC4 counter, the latency values were assigned into time-buckets with a length of $1/983040 \text{ s} \approx 1.017 \mu\text{s}$. For each test-condition, the resulting distributions of the interrupt latencies have been condensed in tabular form, and plotted as normalised histograms over the interrupt latencies.

To estimate the maximum achievable bit-rates, the interrupt latency is assumed to be the main determining factor for the maximum achievable error-free sampling bit-rate. It has been shown in section 2.4 on page 20, that in order to ensure correct bit sampling, the sampling point time-offset from the signal edge, ξ , has to satisfy $\varepsilon < \xi < t_p - \varepsilon$. For ISO 7816 this results in $0.2 \text{ etu} < \xi < 0.8 \text{ etu}$ for the sampling time-offset, which directly results from the interrupt latency for the leading edge detection, i.e. $\xi = t$. The data tables contain four columns related to the expected achievable bit-rates based on the determined latencies. The columns are calculated as follows:

$$\begin{aligned} \text{min bitrate} &= \frac{\varepsilon_{\text{ISO7816}}}{t_p \min t} = \frac{0.2}{\min t} \\ \text{max bitrate} &= \frac{t_p - \varepsilon_{\text{ISO7816}}}{t_p \max t} = \frac{0.8}{\max t} \\ \text{max}_{\text{ofs}} \text{ bitrate} &= \frac{t_p - 2\varepsilon_{\text{ISO7816}}}{t_p (\max t - \min t)} = \frac{0.6}{\max t - \min t} \\ \text{max}_{\varepsilon \rightarrow 0} \text{ bitrate} &= \lim_{\varepsilon \rightarrow 0} \frac{t_p - \varepsilon}{t_p \max t} = \frac{1}{\max t} \end{aligned}$$

The columns `min bitrate` and `max bitrate` represent the theoretical minimum and maximum bit-rates possible (without adopting latency offset compensation measures) for the measured t distribution, based on the worst-case timings as permitted by the tolerances in the ISO 7816 specification.

The `maxofs bitrate` column gives the upper limit for the bit-rate, this time by using an additional bit-rate dependant sampling offset, `ofs` = $\varepsilon - \min t$, which modifies the inequality for the edge detection latency to $\varepsilon < \text{ofs} + \xi < t_p - \varepsilon$.

Finally, the column denoted by `max $\varepsilon \rightarrow 0$ bitrate` shows the theoretical maximum bit-rate, when assuming perfect incoming signal bit timings without jitter, i.e. $\varepsilon = 0$.

The common setup for the measurements consisted of the target-platform (see section 4.1.1) connected to a workstation through a null-modem cable to the targets' serial console, and through a Fast Ethernet switched network. The target was booted over the network, and used a NFS root file-system containing a minimal Debian GNU/Linux 3.1/ARM[33] system served from the workstation.

The System was controlled by logging in on the serial console. The measurement was performed by loading the latency measurement kernel module, sleeping for a certain amount of time by use of the `sleep(1)` standard Unix tool, and finally removing the kernel module. The kernel messages appearing on console after removing the kernel module, which contained the latency measurement results, were monitored and saved for later processing, which resulted in the tables and plots presented in this chapter.

The diagrams use a logarithmic vertical density axis, as the density around the latency density peak declines fast, and would otherwise look like a 0-line around the density peak.

5.1.1 Idle System

An idle system, that is a system which has no user tasks to execute, is expected to represent the ideal case with respect to low latency, since only few context switches happen and therefore few cache replacements and high cache hit-rates are expected; also expected is little driver activity, which could cause interrupt processing, which in turn would lead to additional context switches, and also to time periods where other interrupts would be delayed due to temporarily disabled interrupt processing.

Table 5.1 on page 47 and figure 5.1 on page 47 show the data acquired for such an idle system. The range for the interrupt latencies is roughly the same for all frequencies¹, i.e. from about 2 μs up to about 29 μs . Most latencies measured to be below 10 μs , with typical latencies between 3 μs and 4 μs .

5.1.2 Network Stress

For the purpose of analysing the impact of interrupt-stress on the interrupt latency, the system was bombarded with network packets, which caused the network driver's interrupt handler to be called for each received packet.

These measurements were taken while the System was exposed to *ping flooding* with a default packet-size of 56 bytes, i.e. sending *ICMP ECHO_REQUESTs* as fast as possible to the target system, with a flood rate between 1 and 2 packets per ms. During the whole measurement no packet-loss was detected.

The measured data is reported in table 5.2 on page 48 and figure 5.2 on page 48. The latency distribution resembles the one determined for an idle system, except for being dilated towards slightly higher (about a few μs) latencies. There is an isolated density point at 32 μs for the 105 Hz distribution whose apparent isolation is caused by an insufficient amount of samples for the latency range above 14 μs : that density point corresponds to a single latency sample.

5.1.3 Cache-Disabled System

The previous measurement determined the impact of interrupt processing on the latency; This time the objective is on another latency inducing effect, namely cache-misses.

¹In the case of the 1 Hz interrupt trigger frequency, the maximum value is expected to be reach the same maximum as the other frequencies, given enough samples; but due to limited time resources it was not possible to acquire more samples.

The most demonstrative way to determine the effect of cache-misses is to cause all memory operations to be cache-misses which can be achieved by simply disabling the caches. On the target-platform it was possible to disable the instruction and data cache individually, therefore allowing to observe the individual effect of each of the caches on the latencies.

To begin with, only the instruction cache was disabled, which resulted in the data in table 5.3 on page 49 and figure 5.3 on page 49. For the next measurement, the data cache was disabled as well, and lead to the results in table 5.4 on page 50 and figure 5.4 on page 50. As expected, the results show a latency distribution very different from the ones previously measured; the distribution is shifted and dilated significantly towards higher latencies. The largest increase in latency, about $50\ \mu\text{s}$, was caused by disabling the instruction cache, the data cache only added around $10\ \mu\text{s}$ to the latency minima detected; most latencies were determined around $110\ \mu\text{s}$ for disabled instruction cache, whereas disabling the data cache again added around $10\ \mu\text{s}$ to the latency density peak.

5.1.4 System Under Combined Stress

In order to get values for realistic real-world worst-case scenarios, a continuous memory test over 1 MiB was performed while running the measurements, which is expected to stress the data cache. Moreover, the frequently updating memory testing progress displayed on the standard error file descriptor was transmitted over the network by the ssh session connected to the embedded system; this is expected to cause the instruction cache to be *polluted* by the network handling code and the executed encryption routines in the ssh daemon. Therefore, this setup was expected to cause frequent cache-line replacements as well as frequently executing the network driver code and causing network-induced interrupt processing.

Depending on whether the cache-pressure caused by the additional stress is enough to cause the few cache-lines required by the interrupt handling path for the latency measurement module to get expunged from the cache, the latency should be either unaffected, i.e. comparable to the one on an idle system, or in case of a completely non-cached code-path, somewhere near the cache-disabled latency values.

The measured data is reported in table 5.5 on page 51 and figure 5.5 on page 51, showing the distribution lying (as expected) between the one of an idle system and the one of a totally non-cached system. The measured data shows, as expected, increasing maximum latencies for increasing trigger frequencies; however, at the same time, a clearly visible decrease in the average and peak latencies is observed, which is caused by cache effects.

rate f [Hz]	samples n [1]	latency t [μ s]					achievable bit-rate [bps]			
		min	μ	max	σ^2	peak	min	max	max _{ofs}	max _{$\epsilon \rightarrow 0$}
1	3541	2.0	4.2	15.3	2.6	3.1	393216	52429	45371	65536
10	17742	1.0	3.6	25.4	1.1	3.1	786432	31457	24576	39322
105	30999	2.0	3.9	27.5	0.7	4.1	393216	29127	23593	36409
997	294583	2.0	3.1	28.5	0.2	3.1	393216	28087	22686	35109

Table 5.1: Distribution of the interrupt latency t for idle system at different interrupt trigger frequencies f , together with the predicted ranges of achievable bit-rates

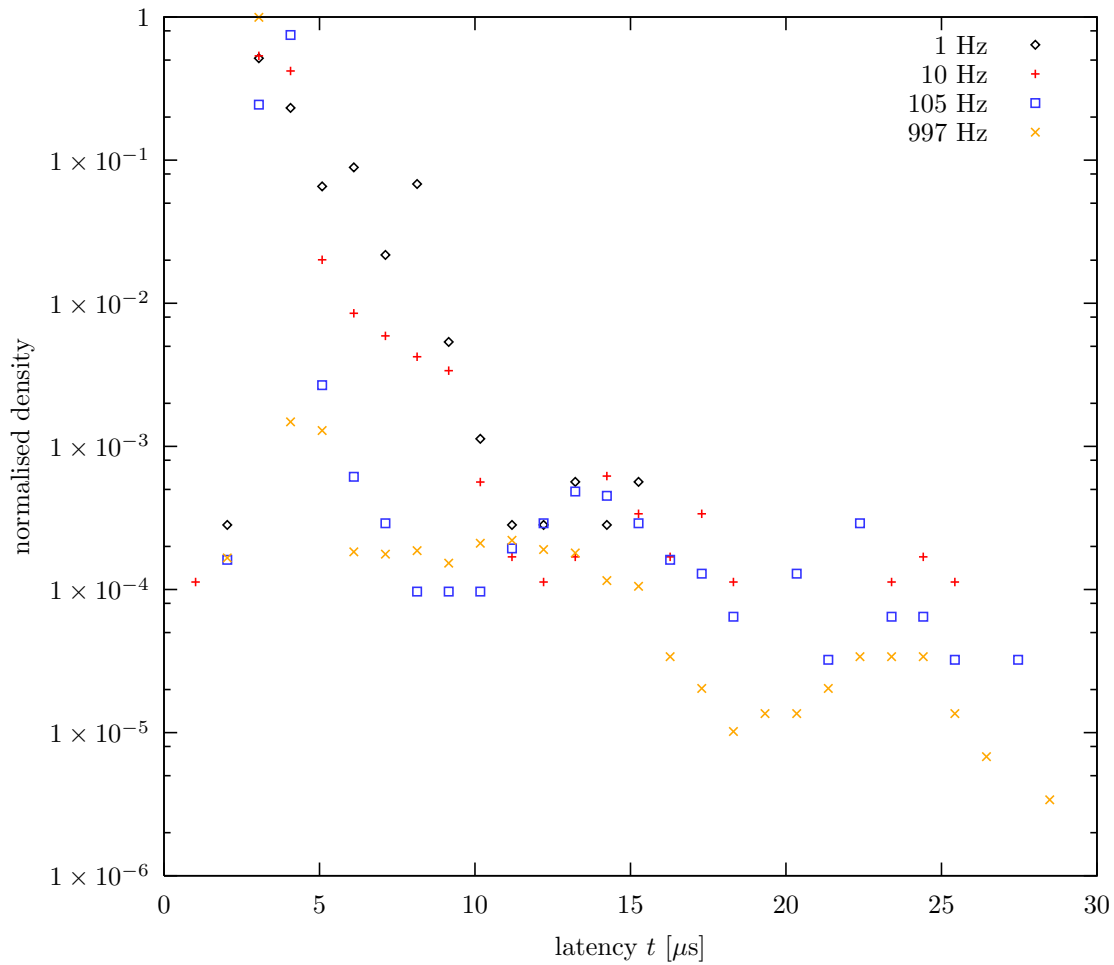


Figure 5.1: Resulting logarithmic histogram of the interrupt latency distribution for idle system at different interrupt trigger frequencies

rate f [Hz]	samples n [1]	latency t [μ s]					achievable bit-rate [bps]			
		min	μ	max	σ^2	peak	min	max	max _{ofs}	max _{$\epsilon \rightarrow 0$}
1	3540	3.1	5.3	14.2	1.8	5.1	262144	56174	53620	70217
10	3549	2.0	4.4	17.3	0.7	4.1	393216	46261	39322	57826
105	30985	1.0	4.2	32.6	0.6	4.1	786432	24576	19027	30720
997	294654	2.0	4.2	26.4	0.3	4.1	393216	30247	24576	37809

Table 5.2: Distribution of the interrupt latency t for system with network stress applied at different interrupt trigger frequencies f , together with the predicted ranges of achievable bit-rates

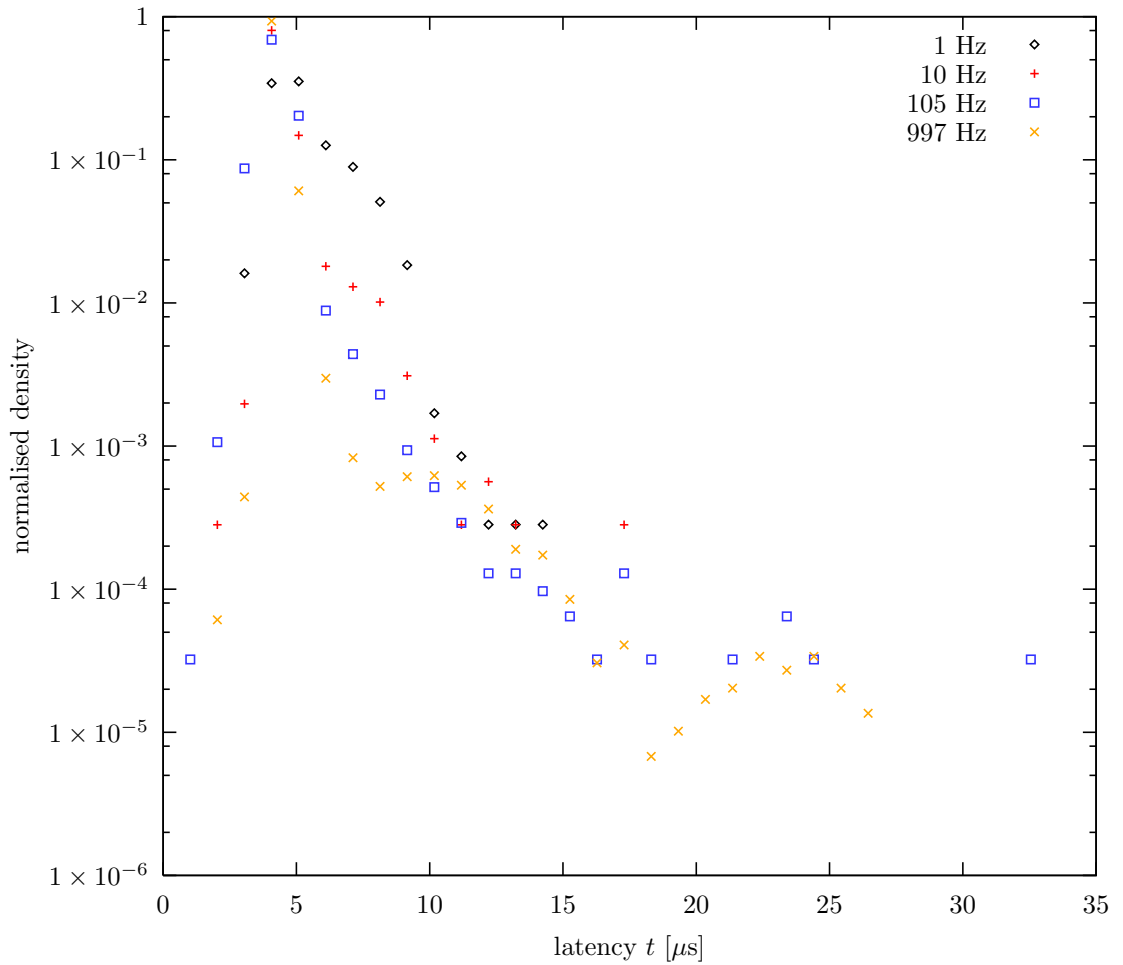


Figure 5.2: Resulting logarithmic histogram of the interrupt latency distribution for system with network stress applied at different interrupt trigger frequencies

rate f [Hz]	samples n [1]	latency t [μ s]					achievable bit-rate [bps]			
		min	μ	max	σ^2	peak	min	max	max _{ofs}	max _{$\epsilon \rightarrow 0$}
1	3541	57.0	119.1	224.8	59.6	117.0	14043	3559	—	4448
10	17748	52.9	117.2	217.7	55.2	116.0	15124	3675	3641	4594
105	31065	52.9	118.3	215.7	54.3	117.0	15124	3710	3686	4637
997	296607	52.9	118.4	267.5	54.1	118.0	15124	2990	2795	3738

Table 5.3: Distribution of the interrupt latency t for idle system with disabled i-cache at different interrupt trigger frequencies f , together with the predicted ranges of achievable bit-rates

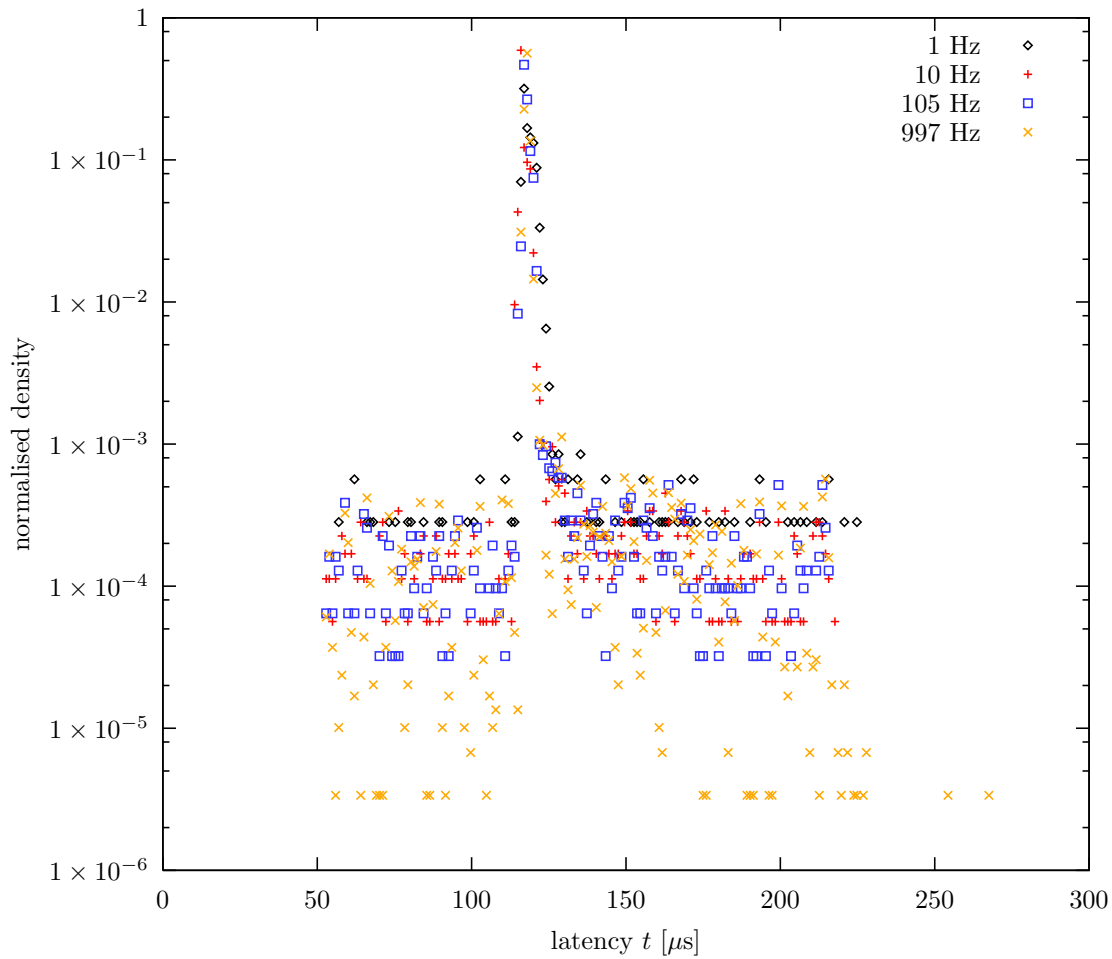


Figure 5.3: Resulting logarithmic histogram of the interrupt latency distribution for idle system with disabled i-cache at different interrupt trigger frequencies

rate f [Hz]	samples n [1]	latency t [μ s]					achievable bit-rate [bps]			
		min	μ	max	σ^2	peak	min	max	max _{ofs}	max _{$\epsilon \rightarrow 0$}
1	3541	63.1	132.3	275.7	146.3	129.2	12684	2902	2822	3627
10	17750	63.1	131.8	279.7	148.3	130.2	12684	2860	2769	3575
105	31088	62.1	131.4	278.7	148.6	129.2	12892	2870	2769	3588
997	297423	62.1	132.8	378.4	154.5	129.2	12892	2114	1897	2643

Table 5.4: Distribution of the interrupt latency t for idle system with disabled d- and i-cache at different interrupt trigger frequencies f , together with the predicted ranges of achievable bit-rates

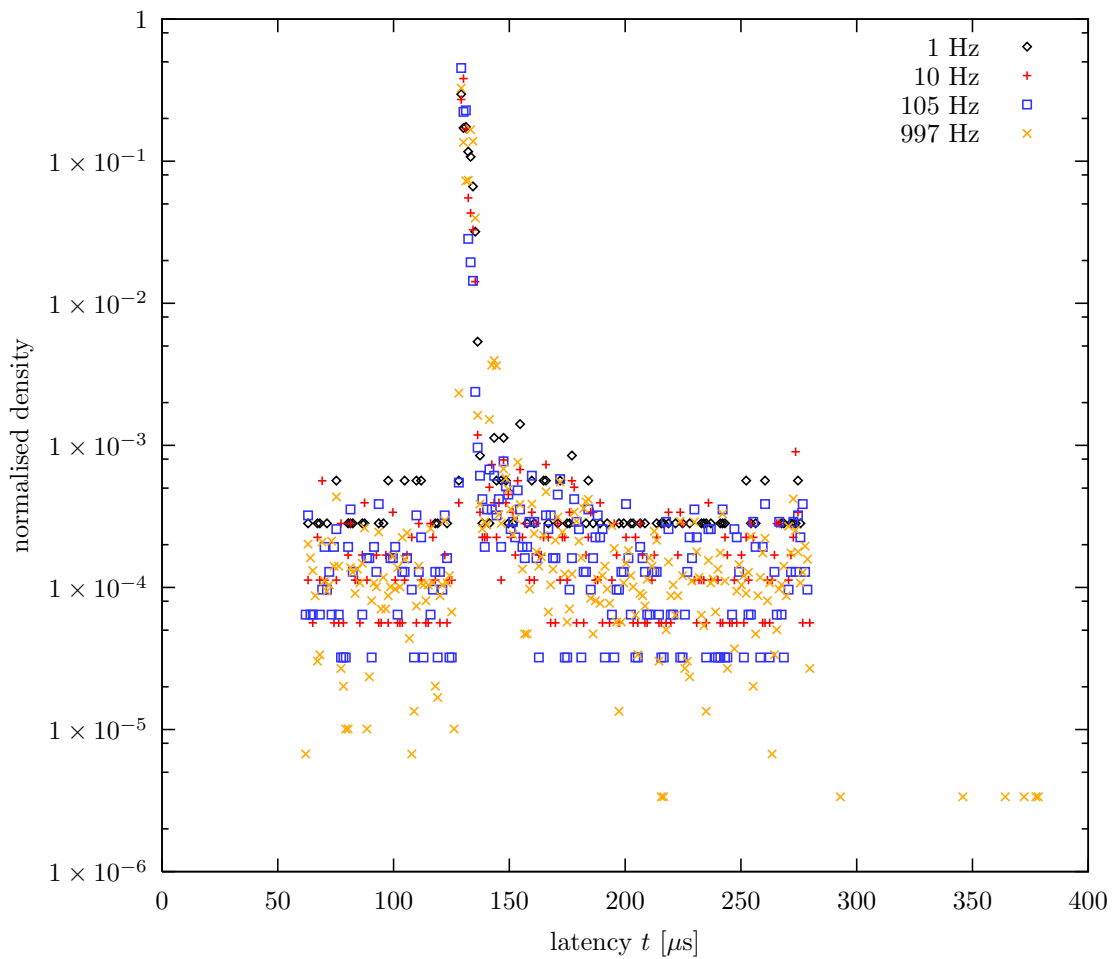


Figure 5.4: Resulting logarithmic histogram of the interrupt latency distribution for idle system with disabled d- and i-cache at different interrupt trigger frequencies

rate f [Hz]	samples n [1]	latency t [μs]					achievable bit-rate [bps]			
		min	μ	max	σ^2	peak	min	max	max _{ofs}	max _{$\varepsilon \rightarrow 0$}
1	3541	8.1	18.8	54.9	25.1	15.3	98304	14564	12822	18204
10	17742	5.1	18.9	84.4	26.4	15.3	157286	9475	7562	11844
105	31010	3.1	17.6	84.4	29.9	13.2	262144	9475	7373	11844
997	294754	2.0	9.1	68.2	11.8	8.1	393216	11738	9074	14672

Table 5.5: Distribution of the interrupt latency t for combined stress at different interrupt trigger frequencies f , together with the predicted ranges of achievable bit-rates

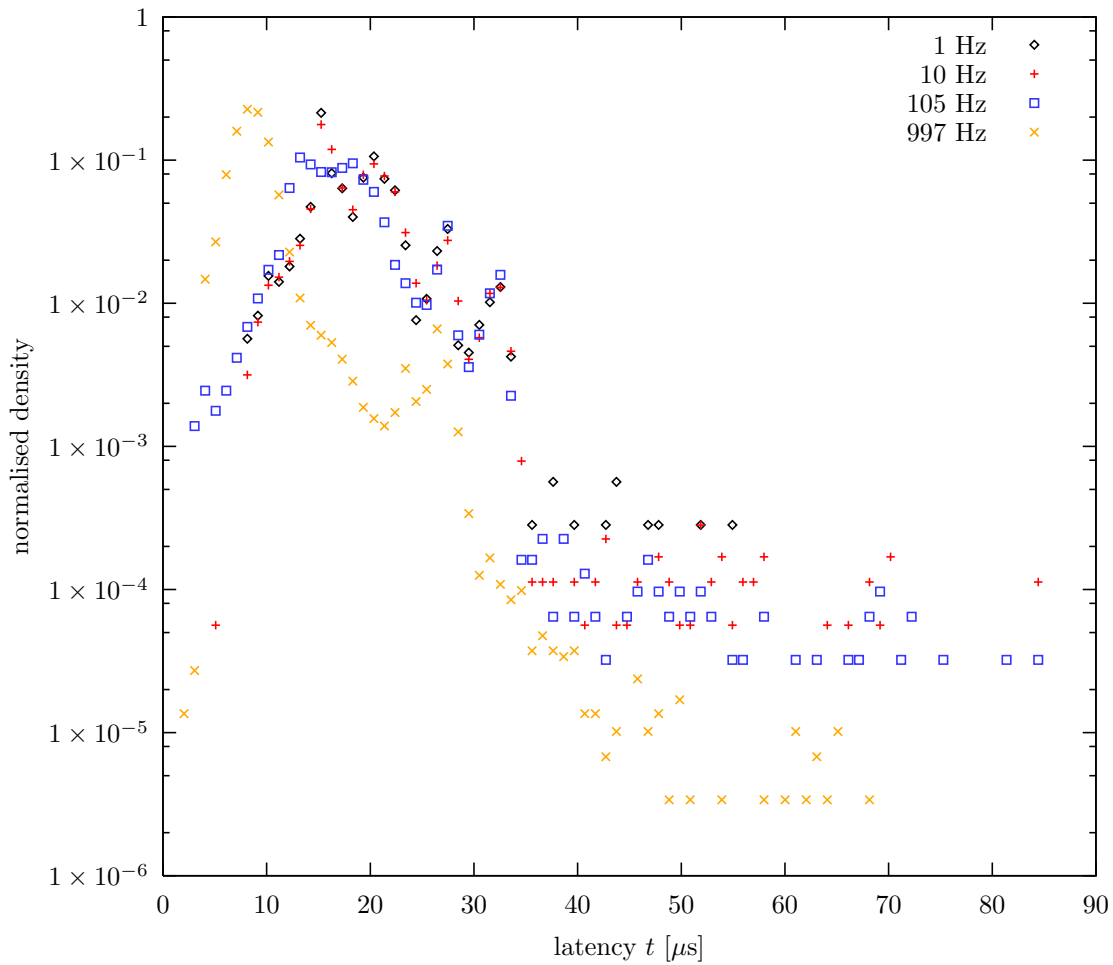


Figure 5.5: Resulting logarithmic histogram of the interrupt latency distribution for combined stress at different interrupt trigger frequencies

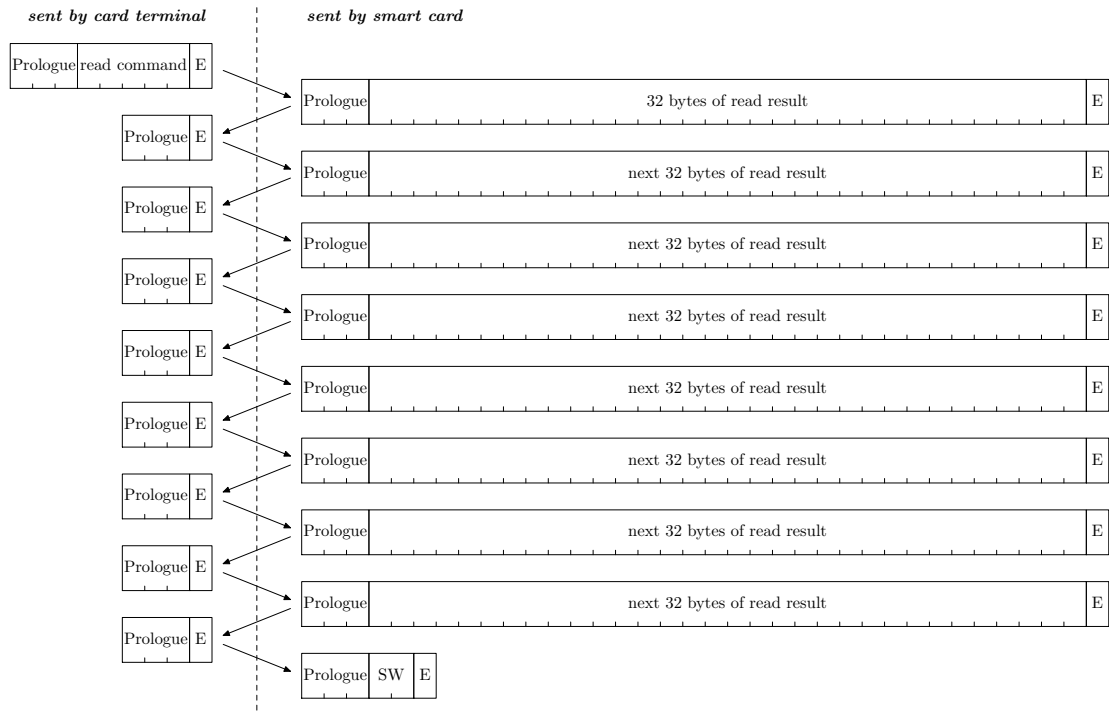


Figure 5.6: ISO 7816 T=1 block-frame sequence representing a 256-byte *ISO READ BINARY* transaction

5.2 Achieved Data Rates

The measurements in this section have the purpose to determine the data rates from table 5.6 on the next page at which error-free communication can be achieved with the implementation presented in this thesis. Additionally, the responsiveness of the system was determined—during ongoing communication—by requesting echo replies from the target system over the network and measuring the round-trip-times for the responses.

In this measurements only the character reception has been focused upon, since the frame sending part is implemented by disabling interrupts processing for the whole duration of sending a block-frame to the ICC and no errors were detected while sending frames for data rates at which receiving was error free as well.

The benchmark was performed by repeatedly issuing the *ISO READ BINARY* command to the ICC, yielding a response of 256 frames together with the 2-frame status-word. The transaction was encapsulated and packetised by the T=1 protocol layer as shown in figure 5.6. Thus an iteration of the test consisted in sending one 9-byte block-frame and eight 4-byte block-frames, interleaved with the reception of eight 36-byte block-frames and one final 6-byte block-frame. Therefore, the maximum duration for

F	DIVCLK	$D = 1$	$D = 2$	$D = 4$	$D = 8$
512	5	5760	11520	23040	46080
	4	7200	14400	28800	57600
372	5	7928	15855	31711	63422
	4	9910	19819	39639	79277

Table 5.6: Resulting bit-rates per second for the combinations of DIVCLK, D , and F supported by the implementation

which interrupts were disabled was for the single 9-byte block-frame sent to the ICC in each iteration; the duration of which is visualised in the response latency plots as an additional curve.

The smart-card used for the measurements was the author’s personal Austrian electronic national health insurance card, also known as the *e-card*, which is manufactured by Giesecke & Devrient. The *e-card* supports $T=1$ and all combinations supported by the ISO 7816 UART implementation under test—see table 5.6—except for $F=512/D=1$ and $F=372/D=8$.

The number of frames received whose parity did not match the data bits’ parity was counted during all iterations, and then divided by the total number of frames received to yield the relative error-rate.

Additionally, the response latency at the tested communication bit-rates was measured by using the standard UNIX tool `ping(8)` from the *iputils* package[9]. The operation principle of `ping(8)` is to send *ICMP ECHO_REQUESTs* to a given host over the network and then wait for *ICMP ECHO_REPLYs* from the contacted host; for each received reply, `ping(8)` reports the round-trip-time to the user. The resulting round-trip-time is the sum of all propagation and processing delays involved, including

- the delay incurred at the originating host for actually placing the echo request packet on the network;
- the time required for the network packet to travel through the network to the destination host;
- the actual response time at the destination host, i.e. the time from the reception of the packet to the emission of the reply packet;
- the time it takes the packet to travel back to the requesting host;
- the delay incurred on the requesting host for receiving and processing the packet.

Thus, the measured value of interest, i.e. the response time of the target system, is only one of the factors making up the round-trip-time measured. In order to minimise the influence of the factors other than the target system response time, it was taken care

to avoid any disturbing load on the network and the host performing the round-trip-time measurements. For the purpose of providing a base-line for the response-times, the round-trip-times were also measured for an idle system with no active software UART communication (hence denoted as “0 bit-rate” round-trip-time in the diagrams). Nevertheless, the round-trip-times should be taken only as a rough estimate for the actual responsiveness of the target system.

The measured round-trip-time distribution for each tested communication bit-rate is summarised in tabular form; together with the number of total frames read and the relative amount of the badly received ones thereof. The diagrams show the round-trip-times distributions as error-bars based on the minimum, mean, and maximum times, together with the herein before mentioned nine 12-etu curve.

5.2.1 Idle System

To begin with, measurements were performed with an unloaded system. Table 5.7 on the facing page and figure 5.7 on page 56 show the results for a system with no load except for the program exercising the *ISO READ BINARY* requests. This setup should provide a picture for the performance that can be achieved with the presented algorithm under ideal conditions.

5.2.2 System under Stress

Again, in order to get values for worse case scenarios, a continuous memory test over 1 MiB was performed while running the measurements. Moreover, the frequently updating memory testing progress display on standard error was transmitted over the network by the ssh session connected to the embedded system. The measured data is reported in table 5.8 on the facing page and figure 5.8 on page 57.

bit-rate				frames read		ping round-trip-times [ms]			
[bps]	DIVCLK	F	D	total	bad [ppm]	min	avg	max	sdev
(0)						0.185	0.249	0.957	0.062
7928	5	372	1	1176360	—	0.310	2.610	13.811	1.467
9910	4	372	1	1176360	—	0.333	2.822	11.208	1.161
11520	5	512	2	1176360	—	0.333	2.600	9.648	1.008
14400	4	512	2	1176360	—	0.350	2.657	7.653	0.911
15855	5	372	2	1176360	—	0.332	2.554	6.672	0.799
19819	4	372	2	1176360	—	0.285	2.421	5.824	0.843
23040	5	512	4	1176360	—	0.384	2.488	5.506	0.819
28800	4	512	4	1176360	—	0.309	2.377	4.663	0.820
31711	5	372	4	11760432	—	0.324	2.362	5.029	0.938
39639	4	372	4	11784551	10.8	0.303	2.354	5.345	1.058
46080	5	512	8	1181305	116.8	0.293	2.043	5.195	1.211

Table 5.7: Communication benchmark for idle system showing error rates and responsiveness in terms of ICMP Ping round-trip-times at the various tested bit-rates

bit-rate				frames read		ping round-trip-times [ms]			
[bps]	DIVCLK	F	D	total	bad [ppm]	min	avg	max	sdev
(0)						0.307	0.420	1.322	0.048
7928	5	372	1	11760356	—	0.274	2.687	14.290	1.448
9910	4	372	1	1176360	—	0.307	2.939	11.012	1.147
11520	5	512	2	1176360	—	0.281	2.615	9.136	0.968
14400	4	512	2	1176360	—	0.384	2.679	7.421	0.868
15855	5	372	2	11760356	—	0.294	2.590	7.161	0.871
19819	4	372	2	1176360	—	0.303	2.592	5.866	0.945
23040	5	512	4	1176396	—	0.319	2.536	5.005	0.911
28800	4	512	4	1178079	14.4	0.389	2.531	4.441	0.961
31711	5	372	4	11800317	79.1	0.360	2.355	5.079	1.104
39639	4	372	4	1239648	447.7	0.269	1.662	3.983	1.315

Table 5.8: Communication benchmark for stressed system showing error rates and responsiveness in terms of ICMP Ping round-trip-times at the various tested bit-rates

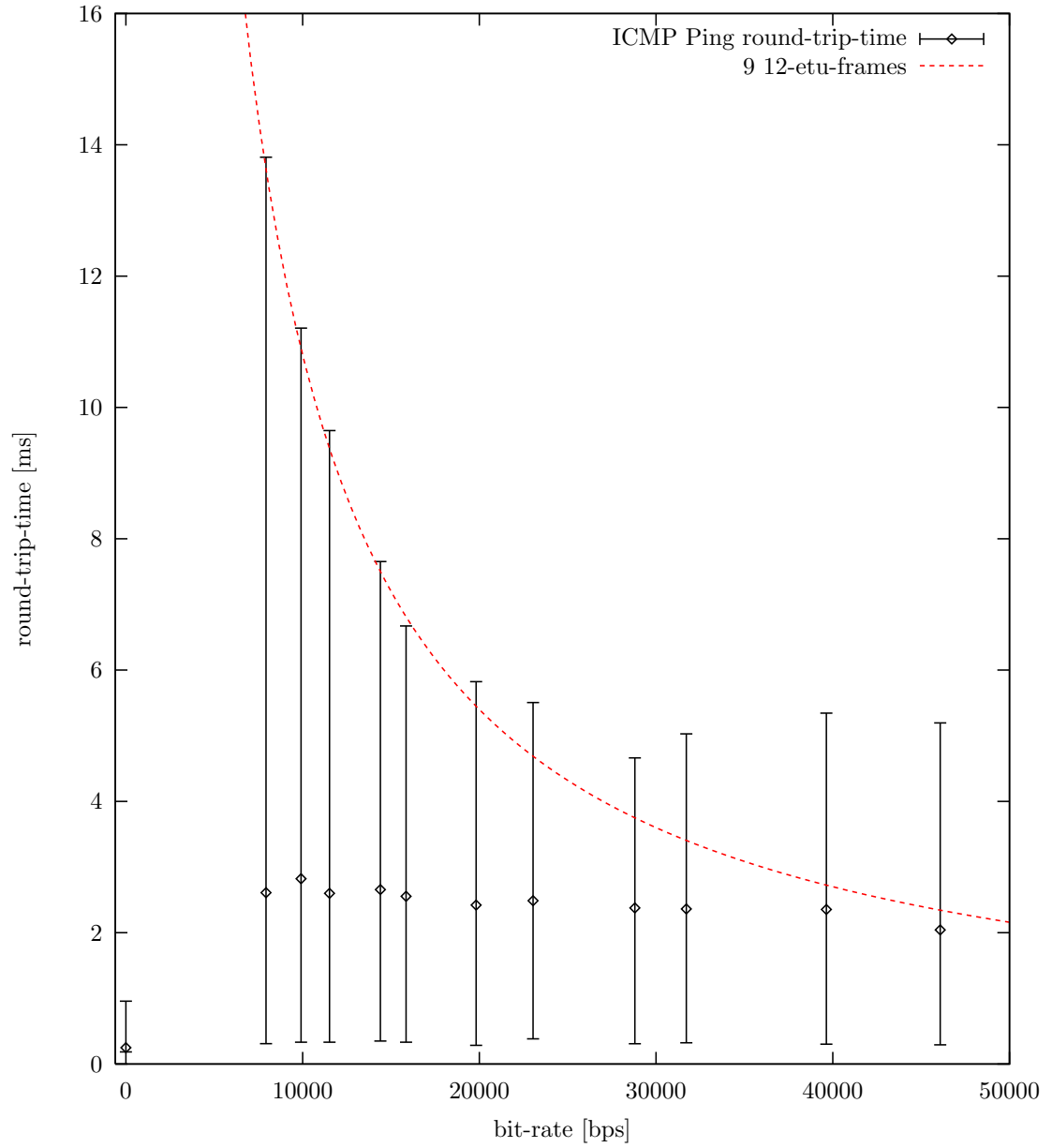


Figure 5.7: ICMP Ping round-trip-times for idle system plotted as error-bars for the various bit-rates during communication

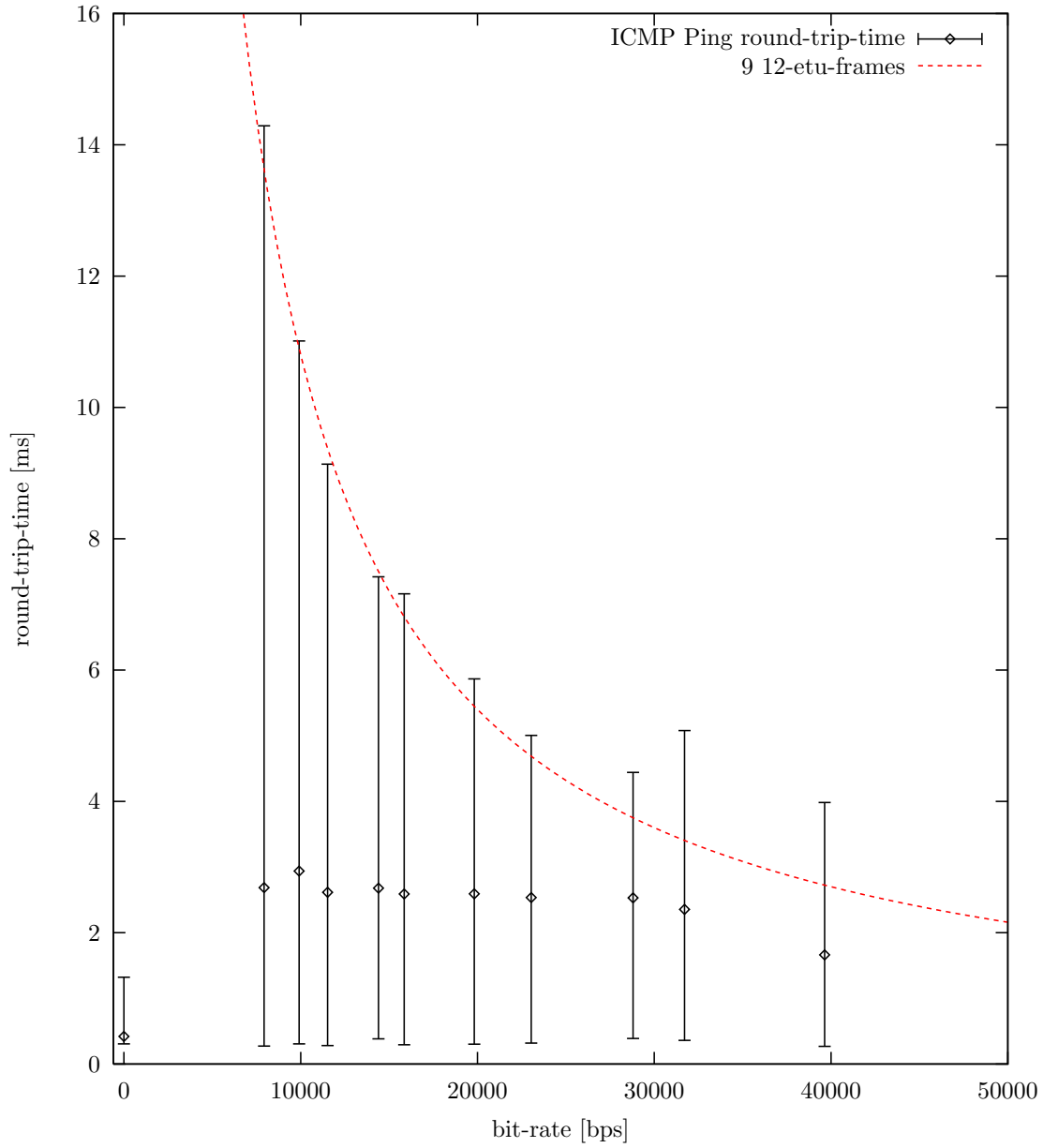


Figure 5.8: ICMP Ping round-trip-times for stressed system plotted as error-bars for the various bit-rates during communication

CHAPTER 6

Analysis of the Results

The aim of this thesis was to present a proof-of-concept implementation of a software UART, in the context of a non-real-time general purpose multi-tasking operating system such as Linux, capable of error-free operation with fair performance while not sacrificing too much responsiveness.

The implementation approach taken to accomplish these goals was to use edge sensitive interrupts in order to detect the beginning of a new character frame being received, without having to resort to actively search the data line for start-bits by polling and thus harming the responsiveness of the system.

In the previous chapter, after having measured the interrupt latency of the target system at various stress conditions, the performance of the presented implementation was bench-marked.

Interrupt Latency

It has been shown that the interrupt latency is the main determining factor for the maximum achievable error-free sampling bit-rate. Therefore, the interrupt latency was measured under various stress conditions to identify how it is influenced.

The interrupt latencies measurements in section 5.1 on page 43 have shown, that cache-misses have a major impact on the interrupt latency. By comparing the various interrupt latency measurements, the following observations can be made:

- For higher sample-taking frequencies f , the maximum of the latency increases, while at the same time, the mean latency decreases slightly. This is assumed to be caused by an increasing probability of the interrupt handling code path (or parts thereof) being still cached for higher frequencies.
- By disabling the caches, the latency distribution curves become less affected by the trigger rate of the latency samples, i.e. they look the same for different trigger

rates. This sustains the assumption, that the effect observed of decreasing mean latency values for increasing interrupt frequencies, is caused by cache-effects.

- The use of memory caches improves the interrupt latency by between one to two orders of magnitude, caused by the major difference of the memory access times of main memory compared with cache memory.
- If the memory data rates are known, the amount of instructions necessary for reaching the interrupt handler can be estimated by dividing the lowest time measured (with disabled instruction caches) to reach the handler (which has been measured to take $53\ \mu\text{s}$) by the value for the memory read transfer-rate.
- The instruction cache-misses seem to have an impact about 5 times greater than data cache-misses, since disabling the data cache added only $10\ \mu\text{s}$ to the minimum latency, which was at $53\ \mu\text{s}$ with only instruction cache disabled.
- The network stress applied had only little effect on the interrupt latency compared to the impact of cache-misses. That is, cache-misses do more harm to the latency than hardware interrupts (which occur frequent with network stress applied); On the contrary, it might have caused the common interrupt handling code path to be cached with a higher probability, as it is executed more often and therefore be less likely to have been replaced by a least-recently-used cache-line replacement-strategy.
- The measured interrupt latencies were used to calculate the theoretically achievable bit-rates. The limits calculated by assuming worst-case exploitation of the tolerances allowed by the specification (bit-rate columns min and max in the tables) and no use of sampling-offset-shifting (as explained in section 4.2.2 on page 39), predicts that bit-errors are possible, as no bit-rate exists since $\text{min} > \text{max}$; therefore, error-free operation can not be guaranteed and thus was not expected with the tested implementation. On the other hand, when using sampling-offset-shifting (bit-rate column max_{ofs}) error-free communication at the default communication rates defined by the ISO 7816 specification seemed possible.

Achieved Data Rates

So, while giving some insight into the interrupt latency behaviour, the measurements also predicted that the communication could not be guaranteed to be error-free. Nevertheless, it was attempted to benchmark the software UART implementation, which surprisingly—according to the prediction made based on the interrupt latency measurements done—would seem to allow for error free communication.

The reason for this is that actual smart-cards show significantly more accurate timings than the tolerance ε of up to 0.2 etu as mandated by the ISO 7816 specification[29]. Therefore, a column was added to the tables for the results of the performed interrupt latency tests, for estimating the theoretically maximum achievable bit-rate for negligible $\varepsilon \rightarrow 0$ values, in order to compare it with the maximum achieved error-free data-rates of the implementation.

The actual results of benchmarking the implementation in section 5.2 on page 52 lead to the following observations:

- The predicted maximum achievable bit-rate for negligible $\varepsilon \rightarrow 0$ values matches the actually achieved error free maximum for an idle system.
- For the stressed system, the actually achieved maximum error-free bit-rate was slightly higher than predicted. It is assumed that the interrupt latencies are not as bad as measured in the latency test, or alternatively, the maximum interrupt latency is reached with such a small probability that the measurements resulted error-free. Repeating the measurements with an order of magnitude more test samples might have shown results more in line with the predicted maximum bit-rate; due to limited time resources, the measurements could not be repeated with the necessary amount of test samples, as they would have taken several days to complete.
- It is important to point out that we cannot prove error-free operation as we cannot even prove that the interrupt latency is really bounded by the maximum value encountered during the measurements. We can only make statements about the predicted probabilities for future values, based on the samples of the quantity in question taken. If bounded interrupt latencies are required, a real-time operating system is needed to guarantee those.
- The maximum ping round-trip-times for low bit-rates are mainly influenced by the maximum time the interrupts are disabled, which is mainly caused by the sender-algorithm disabling the interrupts for the duration of the transmission of up to nine 12-etu-frames at once for the *ISO READ BINARY* command; curiously, the bit-rates for which the max round-trip-times diverge from the curve are also those for which communication errors were observed. It seems, that when the interrupt rate begins to cause congestion, the system becomes more and more non-deterministic. Moreover, the interrupt handling overhead increases relative to the time spent in the interrupt handler, which decreases for higher bit-rates—and thus shorter bit-lengths.

CHAPTER 7

Conclusion

This thesis has introduced universal asynchronous serial protocols, described the role of UARTs, defined the basic conditions for successful data transmission, and specifically addressed the application as the T=0/T=1 protocol in the ISO 7816 standard.

It has been made the case, that it is possible to implement a software ISO 7816 UART in such a way, that a general purpose non-real-time multi-tasking operating system such as Linux would still be responsive while the software UART is active.

The motivation for software UART implementation is the increased flexibility compared to hardware UART components, and potential cost-savings by eliminating a hardware component; additionally, real-time enhanced Linux is not always available for the required target platform, thus a Linux version lacking real-time enhancements was used as implementation target.

A few common software implementation patterns have been presented, which are unsuitable in the context of the target platform at hand. In particular, those would either require deterministic interrupt latencies—which is not guaranteed by standard Linux—, or disabling all processing except for the software UART routines; the latter would cause a decrease in responsiveness of the operating system, and harm other (soft or hard) real-time requiring services (such as network processing or even user interaction) being provided by the system. In the first place, the software UART is required to provide error-free operation.

To this end, a software UART implementation meeting the requirements has been proposed in this thesis, derived as a combination of the common implementation patterns. The proposed implementation uses edge-sensitive GPIO interrupts for start-bit detection, and disables interrupts for the subsequent bits within the frame. Furthermore, the algorithm detects the edge transition in order to synchronise the receiver clock with the sender clock, i.e. compensating for the interrupt latency jitter.

As the presented implementation relies primarily on the interrupt latency to be within certain boundaries, the interrupt latency was measured under various conditions in order

to determine the operational conditions for which error-free software UART operation is to be expected. Finally the actual software UART implementation was bench-marked with respect to error rates and network responsiveness at various communication rates.

The data analysis has shown, that basically the stated goals have been attained in principle, albeit under the favourable condition of accurate protocol timings of the smart-card used; if the smart-card had exploited the tolerances conceded by the ISO 7816 specification, the results would have been significantly worse, possibly resulting in having missed the stated goals with the current implementation.

Furthermore, the dominant effect of the memory system on the interrupt latency has been clearly demonstrated by the measurements, outweighing other effects simulated during the latency tests. On the one hand, memory caches allow for lower interrupt latencies, but on the other hand, when cache-line replacement is not controlled, caches introduce interrupt latency indeterminism.

While avoiding real-time operating systems for tasks having real-time requirements has been shown to be viable under favourable conditions, the motivation for doing so will decrease though; especially, given the long-term intent by the developers of operating systems such as Linux to provide low latency event handling and even real-time facilities in the standard Linux kernel tree. For real-time operating systems, the proposed hybrid algorithm would not provide any gain over the timer- or signal-edge-interrupt based algorithms, as a RTOS provides low bounded interrupt latencies, and would only suffer from algorithms which perform busy-waiting.

7.1 Future Work

As the implementation presented served only as a proof of concept, it suffers from various limitations, which were identified in section 4.2.3 on page 42. Eliminating those limitations is regarded as future work, and there is also room left for various other optimisations and improvements. Some of these candidates for future work are highlighted in this chapter.

7.1.1 Protocol Conformance

This thesis has only presented a proof of concept, without thoroughly adhering to timing requirements mandated by the ISO 7816 specification. In order to pass smart-card terminal certification test cases, this would have to be made up for.

7.1.2 Latency Improvements

As this thesis has focused on character reception rather than character transmission, for simplicity, the presented implementation disables interrupt processing and context switching for the complete duration of transmitting frames to the smart-card. By

re-enabling interrupt processing between the character frames—while taking necessary steps to ensure keeping the inter-frame timings within protocol constraints—the latency penalties incurred by the frame sending process can be reduced, at the cost of possibly lower effective data rates due to possible additional inter-frame delays.

7.1.3 Performance Improvements

The EP9301 has support for so-called fast interrupts, which are processed with higher priority than regular interrupts, and have a set of private registers at their disposition. These fast interrupts are intended for use in data channel input/output routines.

Moreover, since cache effects are one major cause of increased and non-deterministic timing latency, the ARM920T core allows for cache lock-down, i.e. to keep certain cache lines from being replaced, thus providing lower memory access times in a more deterministic manner.

By making use of these two latency decreasing—albeit platform dependent—facilities, it should be possible to improve the software UART's performance.

7.1.4 Reliability Improvements

The provided implementation does not make use of minimum latency offset compensation in order to shift the sampling point into the sampling zone where the data signal is guaranteed by the specification to be reliable. Implementing the offset compensation would improve the ISO 7816 conformance with respect to worst-case timing scenarios.

7.1.5 Full-duplex support

Although not required for the ISO 7816 protocol, especially given that only one wire is used for both data transmission directions, extending the algorithm presented in this thesis to the generic case of full-duplex operation might be desirable for other protocols.

APPENDIX A

ISO 7816 Software UART Driver Source Code

```
/*
 * Software ISO7816 UART implementation for EP9301 CPU
 *
 * written by
5  * Herbert Valerio Riedel <hvr@inso.tuwien.ac.at>
 *
 */

/*
10 * TODOS/ISSUES (incomplete list):
 * - T=0 RX retransmissions not implemented
 * - proper RX timeout implementation (e.g. use hw timer instead of kernel timer)
 * - worst case for autosync are null-bytes!
 * - only class A supported
15 * - only CLKDIV4 and CLKDIV5 supported, by compile time selection
 * - poweron/off not reentrancy safe
 *
 */

20 #include <linux/config.h>

#if !defined(CONFIG_ARCH_EP9301)
# error This driver works only with the cirrus ep9301 (arm920t) arch
#endif

25 #include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/delay.h>
30 #include <linux/interrupt.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/poll.h>
#include <linux/tqueue.h>
35 #include <linux/timer.h>
```

```

#include <linux/i2c-id.h>
#include <linux/i2c.h>

40 #include <asm/irq.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm-arm/arch-ep93xx/regmap.h> /* EP93XX register defines */

45 #include "ifd.h"
#include "ep93-wd.h"

#warning TODO: reserve 0x21 as well, i.e. create a 2nd i2c client object
static unsigned short normal_i2c[] = { 0x20, I2C_CLIENT_END };
50 static unsigned short normal_i2c_range[2] = { I2C_CLIENT_END, I2C_CLIENT_END };

I2C_CLIENT_INSMOD;

#define TICKS_SHIFT 1
55 #define TICKS_PER_ETU (1 << TICKS_SHIFT)
#define ETU(bit, samp) (((bit) << TICKS_SHIFT) + (samp))

// #define USE_TICKS_TIMEBASE
#define DEFAULT_CLKDIV 5
60 // #define DEFAULT_CLKDIV 4
#define I2C_DRIVERID_TDA8023 0xf021

static int tda8023tt_debug = 1;
const static int tda8023tt_io_irq = IRQ_GPIO1; // FPGIO[1]
65 const static int tda8023tt_ctl_irq = IRQ_GPIO2; // FPGIO[2]

static unsigned tda8023tt_instances = 0;
static struct tda8023tt_data *tda8023tt_instance_data[1] = { 0 };

70 #define IFD_OUTBUF_MAXSIZE (3+255+2) // for T=1: 3 byte prolog,
// 255 bytes payload, 1-2 epilogs;
// for T=0: 5 command header, 255 payload

static inline void
75 __distract_watchdog (void)
{
#ifdef CONFIG_EP93_WD || defined(CONFIG_EP93_WD_MODULE)
ep93_wd_feed (); // feed watchdog
#endif
80 }

struct tda8023tt_data {
struct ifd_device _ifd;

85 struct i2c_client *client;
struct tasklet_struct tasklet_bh;
struct tasklet_struct tasklet_io_bh;

struct tq_struct _task_start;
90 struct tq_struct _task_tx;
struct tq_struct _task_rx_to;
struct timer_list _timer_rx_to;

ifd_device_class_t __start_class;

```

```

95  char __start_warm:1;
    char __card_inserted:1;
    char tx_commit:1;          // see _ifd_frames_tx

    u8 divisor; /* n = {1,2,4,5} */
100  u16 outbuf[IFD_OUTBUF_MAXSIZE];
    u16 outbuf_len; // producer pos

    // stats
105  u16 wretrns;
    u16 rretrns;

    u16 frames_perrors;
    u16 frames_badframes;
110  u16 frames_rx;
    u16 frames_tx;
};

// reg0/reg1 accessor methods - hack to workaround limitation in i2c framework
115  static inline s32
tda8023tt_write_reg (struct tda8023tt_data *data, unsigned reg, u8 value)
{
    struct i2c_client *const client = data->client;
    BUG_ON(reg > 1);
120  return i2c_smbus_xfer(client->adapter, client->addr + reg, client->flags,
                          I2C_SMBUS_WRITE, value, I2C_SMBUS_BYTE, NULL);
}

static inline s32
125  tda8023tt_read_reg (struct tda8023tt_data *data, unsigned reg)
{
    struct i2c_client *const client = data->client;
    union i2c_smbus_data i2cdata;
    BUG_ON(reg > 1);
130  if (i2c_smbus_xfer(client->adapter, client->addr + reg, client->flags,
                      I2C_SMBUS_READ, 0, I2C_SMBUS_BYTE, &i2cdata))
        return -1;
    else
        return 0x0FF & i2cdata.byte;
135  }

#define MAX_RETRANS 100
// implementation

140  // I/OuC related
static volatile u32 *const __gpio_pfdr = (u32*)IO_ADDRESS(GPIO_PFDR);
static volatile u32 *const __gpio_pfddr = (u32*)IO_ADDRESS(GPIO_PFDDR);
static volatile u32 *const __gpio_finten = (u32*)IO_ADDRESS(GPIO_INTEN);
static volatile u32 *const __gpio_finttype1 = (u32*)IO_ADDRESS(GPIO_INTTYPE1);
145  static volatile u32 *const __gpio_finttype2 = (u32*)IO_ADDRESS(GPIO_INTTYPE2);
static volatile u32 *const __gpio_fdb = (u32*)IO_ADDRESS(GPIO_FDB);
static volatile u32 *const __gpio_feoi = (u32*)IO_ADDRESS(GPIO_FEOI);
static volatile u32 *const __gpio_fintstatus = (u32*)IO_ADDRESS(GPIO_INTSTATUS);

150  static const u32 __gpio_pf_iouc_mask = (1<<1); // FGPIO[1]
static const u32 __gpio_pf_ctl_mask = (1<<2); // FGPIO[2]

```

70 Appendix A. ISO 7816 Software UART Driver Source Code

```
static inline int
_iouc_get (void)
155 {
    *__gpio_pfddr &= ~__gpio_pf_iouc_mask;
    return (*__gpio_pfdr & __gpio_pf_iouc_mask) != 0;
}

160 static inline void
_iouc_set (int state)
{
    *__gpio_pfddr |= __gpio_pf_iouc_mask;

165     if (state)
        *__gpio_pfdr |= __gpio_pf_iouc_mask;
    else
        *__gpio_pfdr &= ~__gpio_pf_iouc_mask;
}

170 static inline void
_iouc_init (void)
{
    *__gpio_finten &= ~__gpio_pf_iouc_mask; // disable IRQ capabilities
175     _iouc_set (1); // set I/O signal high
}

static int tda8023tt_probe_client (struct i2c_adapter* adapter, int address,
                                unsigned short flags, int kind);

180 static int
tda8023tt_attach_adapter (struct i2c_adapter *adapter)
{
    return i2c_probe (adapter, &addr_data, tda8023tt_probe_client);
185 }

//
// Hardware timer abstraction
//
190 #define EP93XX_TIMER_FREQ      983040 // 14.7456/15 Mhz = 983.04 KHz

/* enables the timer if set, disables otherwise */
#define EP93XX_TIMER_ENABLE    0x100

195 /* Sets up a fast hardware timer with 983.04 KHz frequency
   * in free running mode
   */
static void
ep93xx_timer_init(void)
200 {
    /* reset the timer */
    outl(0x0, TIMER4VALUEHIGH);

    /* enable the timer */
205     outl(EP93XX_TIMER_ENABLE, TIMER4VALUEHIGH);
}

/* Returns the current value of the timer which
   * must have been set up previously; otherwise, the result
210   * is undefined
```

```

*/
static inline u32
ep93xx_timer_get(void)
{
215 // overflow in the low 32bits occurs every ~4369s (~73m)
    return inl(TIMER4VALUELOW);
}

/* Resets the timer to the initial value */
220 static inline void
ep93xx_timer_reset(void)
{
    ep93xx_timer_init();
}
225

//
// End of hardware timer abstraction
//

230 static inline void tda8023tt_reset_counter (void);

static inline int
tda8023tt_timer_setup_try (const unsigned clkdiv, const unsigned F,
                           const unsigned D)
235 {
    if (clkdiv != DEFAULT_CLKDIV)
        return -1;

    switch (F) {
240 case 512:
        if (D == 1) return 0;
        if (D == 2) return 0;
        if (D == 4) return 0;
        if (D == 8) return 0;
245 break;
    case 372:
        if (D == 1) return 0;
        if (D == 2) return 0;
        if (D == 4) return 0;
250 if (D == 8) return 0;
        break;
    }

    return -1;
255 }

static int __counter_shift = 0;
static int __counter_372 = 1;

260 static inline int
tda8023tt_timer_setup (const unsigned clkdiv, const unsigned F, const unsigned D)
{
    if (!clkdiv)
        return 0; // just disable timer

265 if (clkdiv != DEFAULT_CLKDIV) {
    printk (KERN_ERR "%s: _unsupported/invalid _CLKDIV_(%d) _passed\n",
            __func__, clkdiv);
}

```

```

    return -1;
270 }

    if (tda8023tt_timer_setup_try (clkdiv, F, D) < 0) {
        printk (KERN_ERR "%s: unsupported F/D combination (%d/%d) passed\n",
                __func__, F, D);
275     return -1;
    }

    __counter_372 = (F == 372);

280     if (F == 372)
        switch (D) {
            case 1: __counter_shift = 4; break;
            case 2: __counter_shift = 3; break;
            case 4: __counter_shift = 2; break;
285     case 8: __counter_shift = 1; break;
            default: BUG(); return -1;
        }
    else if (F == 512)
        switch (D) {
290     case 1: __counter_shift = 10; break;
            case 2: __counter_shift = 9; break;
            case 4: __counter_shift = 8; break;
            case 8: __counter_shift = 7; break;
            default: BUG(); return -1;
295     }
        else {
            BUG ();
            return -1;
        }
300     tda8023tt_reset_counter ();

    return 0;
}

305 // high precision mdelay()
// wait for x msec (for x < 1000)

    static inline u32
310 ep93xx_msec2timer (unsigned msec)
    {
        return (msec * EP93XX_TIMER_FREQ) / 1000;
    }

315 // funny integer arithmetic to get from CLK/15 to D*CLK/4/F
//     for D={1,2,4,8} (causes wrapover at ~7.3m, due to losing high bits)
// getting to CLK/512/D for D={1,2,4,8,...,512} (wrapover @ ~4.9m)
    static inline u32
320 ep93xx_ticks2timer (unsigned ticks)
    {
        #if DEFAULT_CLKDIV==4
            return (__counter_372
                    ? ((ticks << __counter_shift) * 31 + 5) / 10
                    : ((ticks << __counter_shift) + 7) / 15);
325 #elif DEFAULT_CLKDIV==5
            return (__counter_372

```

```

        ? ((ticks << __counter_shift) * 31 + 4)/8
        : ((ticks << __counter_shift) + 6)/ 12);
#else
330 # error CLKDIV not implemented
#endif
}

static inline unsigned
335 ep93xx_timer2ticks (u32 t)
{
#if DEFAULT_CLKDIV==4
    return (__counter_372
        ? ((t * 10 + 15) / 31) >> __counter_shift
        : (t * 15) >> __counter_shift);
340 #elif DEFAULT_CLKDIV==5
    return (__counter_372
        ? ((t * 8 + 15) / 31) >> __counter_shift
        : (t * 12) >> __counter_shift);
345 #else
# error CLKDIV not implemented
#endif
}

350 ///////////////////////////////////////////////////////////////////
static inline void
tda8023tt_mdelay (unsigned msec, int now)
{
    u32 _deadline = now ? ep93xx_timer_get () : 0;
355    _deadline += ep93xx_msec2timer (msec);

    while (ep93xx_timer_get () < _deadline) {};
}

360 static inline u32
ep93xx_timer_get_ticks (void)
{
    return ep93xx_timer2ticks (ep93xx_timer_get ());
}
365 ///////////////////////////////////////////////////////////////////
// some timing related primitives

//wait until 'ticks' ticks have passed
370 #define ZERO_OFS 0
static u32 __counter_deadline = ZERO_OFS;

#if !defined(USE_TICKS_TIMEBASE)
375 # define ZERO_OFS_TICKS 0
static u32 __counter_deadline_ticks = ZERO_OFS_TICKS;
#endif

static inline void
380 tda8023tt_consume_ticks_nowait (const u32 ticks)
{
    BUG_ON(!ticks);
    #if defined(USE_TICKS_TIMEBASE)
        __counter_deadline += ticks;
    #endif
}

```

```
385 #else
    __counter_deadline_ticks += ticks;
    __counter_deadline = ep93xx_ticks2timer(__counter_deadline_ticks);
#endif
}
390
static inline int
tda8023tt_counter_synced (void)
{
    #if defined(USE_TICKS_TIMEBASE)
395     return (ZERO_OFS + ep93xx_timer_get_ticks ()) >= __counter_deadline;
    #else
        return (ZERO_OFS + ep93xx_timer_get ()) >= __counter_deadline;
    #endif
}
400
static inline void
tda8023tt_sync_deadline (void)
{
    while (!tda8023tt_counter_synced ()) {/*noop*/}
405 }

static inline void
tda8023tt_consume_ticks (const u32 ticks)
{
410     tda8023tt_consume_ticks_nowait (ticks);
    tda8023tt_sync_deadline ();
}

static inline void
415 tda8023tt_reset_counter (void)
{
    __counter_deadline = ZERO_OFS;
    #if !defined(USE_TICKS_TIMEBASE)
        __counter_deadline_ticks = ZERO_OFS_TICKS;
420 #endif
    ep93xx_timer_reset ();
}

////////////////////////////////////

425
static inline int
tda8023tt_ctl_irq_pending (void)
{
    return (*__gpio_fintstatus & __gpio_pf_ctl_mask) != 0;
430 }

static inline void
tda8023tt_ctl_irq_on (void)
435 {
    *__gpio_pfddr      &= ~__gpio_pf_ctl_mask;
    *__gpio_finten     &= ~__gpio_pf_ctl_mask;
    *__gpio_finttype1 |= __gpio_pf_ctl_mask; // edge
    *__gpio_finttype2 &= ~__gpio_pf_ctl_mask; // falling
440    *__gpio_feoi      = __gpio_pf_ctl_mask; // clear irq
    *__gpio_fdb       &= ~__gpio_pf_ctl_mask;
    *__gpio_finten    |= __gpio_pf_ctl_mask;
}
```

```

}

445 static inline void
tda8023tt_ctl_irq_clear (void)
{
    *__gpio_feoi      = __gpio_pf_ctl_mask; // clear irq
}

450 static inline void
tda8023tt_ctl_irq_off (void)
{
455  *__gpio_finten    &= ~__gpio_pf_ctl_mask;

static inline void
tda8023tt_io_irq_on (void)
{
460  *__gpio_pfddr     &= ~__gpio_pf_iouc_mask;
    *__gpio_finten   &= ~__gpio_pf_iouc_mask;
    *__gpio_finttype1 |= __gpio_pf_iouc_mask; // edge
    *__gpio_finttype2 &= ~__gpio_pf_iouc_mask; // falling
465  *__gpio_feoi      = __gpio_pf_iouc_mask; // clear irq
    *__gpio_fdb      &= ~__gpio_pf_iouc_mask;
    *__gpio_finten   |= __gpio_pf_iouc_mask;
}

static inline void
470 tda8023tt_io_irq_clear (void)
{
    *__gpio_feoi      = __gpio_pf_iouc_mask; // clear irq
}

475 static inline void
tda8023tt_io_irq_off (void)
{
    *__gpio_finten    &= ~__gpio_pf_iouc_mask;
}

480 static int
tda8023tt_detach_client (struct i2c_client *client)
{
485  struct tda8023tt_data *data = client->data;
    int err;

    tda8023tt_instances--;

    //BUG_ON (tda8023tt_detach_client != 0);

490  free_irq (tda8023tt_io_irq, client->data);
    free_irq (tda8023tt_ctl_irq, client->data);

    if ((err = i2c_detach_client (client)))
495  {
        printk (KERN_WARNING "%s: i2c_detach_client() failed\n", __func__);
        return err;
    }

500  ifd_unregister_dev (&data->_ifd);

```

```

    kfree (client->data);
    kfree (client);

505     return 0;
}

static struct i2c_driver tda8023tt_driver = {
    .name      = "tda8023tt",
510     .id       = I2C_DRIVERID_TDA8023,
    .flags     = I2C_DF_NOTIFY,
    .attach_adapter = tda8023tt_attach_adapter,
    .detach_client = tda8023tt_detach_client,
};

515 #define TDA8023TT_REG0_R_PRES    (1<<0)
#define TDA8023TT_REG0_R_PRESL   (1<<1)
#define TDA8023TT_REG0_R_CLKSWE  (1<<2)
#define TDA8023TT_REG0_R_SUPL    (1<<3)
520 #define TDA8023TT_REG0_R_PROT   (1<<4)
#define TDA8023TT_REG0_R_MUTE    (1<<5)
#define TDA8023TT_REG0_R_EARLY  (1<<6)
#define TDA8023TT_REG0_R_ACTIVE (1<<7)

525 #define TDA8023TT_REG0_W_START  (1<<0)
#define TDA8023TT_REG0_W_WARM    (1<<1)
#define TDA8023TT_REG0_W_5V3V   (1<<2)
#define TDA8023TT_REG0_W_PDOWN  (1<<3)
#define TDA8023TT_REG0_W_REG0   (1<<4)
530 #define TDA8023TT_REG0_W_REG1  (1<<5)
#define TDA8023TT_REG0_W_IOEN   (1<<6)
#define TDA8023TT_REG0_W_18V    (1<<7)

#define TDA8023TT_REG1_CLKDIV1   (1<<0)
535 #define TDA8023TT_REG1_CLKDIV2 (1<<1)
#define TDA8023TT_REG1_CLKPD1   (1<<2)
#define TDA8023TT_REG1_CLKPD2   (1<<3)
#define TDA8023TT_REG1_C4       (1<<4)
#define TDA8023TT_REG1_C8       (1<<5)
540 #define TDA8023TT_REG1_RSTIN   (1<<6)
#define TDA8023TT_REG1_TEST     (1<<7)

#define TDA8023TT_REG1_CLKIN_1  0
#define TDA8023TT_REG1_CLKIN_2  TDA8023TT_REG1_CLKDIV1
545 #define TDA8023TT_REG1_CLKIN_4  TDA8023TT_REG1_CLKDIV2
#define TDA8023TT_REG1_CLKIN_5  (TDA8023TT_REG1_CLKDIV1 | TDA8023TT_REG1_CLKDIV2)

#define TDA8023TT_REG1_INIT     \
    (TDA8023TT_REG1_CLKPD1 | TDA8023TT_REG1_CLKPD2 | TDA8023TT_REG1_RSTIN)

550 /*
 * may be replaced by table lookup, if we _really_ need to save some
 * cycles (with the tradeoff to require ~200 byte more memory)
 */
555 static inline u8
__reverse8 (u8 byte)
{
    byte = ((byte & 0x55) << 1) | ((byte & 0xaa) >> 1);

```

```

byte = ((byte & 0x33) << 2) | ((byte & 0xcc) >> 2);
560 byte = ((byte & 0x0f) << 4) | ((byte & 0xf0) >> 4);

return byte;
}

565 static int
tda8023tt_powerup (struct tda8023tt_data *const data)
{
    u8 _cmd0 = TDA8023TT_REG0_W_START | TDA8023TT_REG0_W_IOEN;
    u8 _cmd1 = TDA8023TT_REG1_INIT;

570     switch (data->__start_class) {
    case IFD_DEVICE_CLASS_A:
        _cmd0 |= TDA8023TT_REG0_W_5V3V;
        break;
575     case IFD_DEVICE_CLASS_B:
        // noop
        break;
    case IFD_DEVICE_CLASS_C:
        _cmd0 |= TDA8023TT_REG0_W_18V;
580         break;
    default:
        BUG();
    }

585     if (data->__start_warm)
        _cmd0 |= TDA8023TT_REG0_W_WARM;

    switch (data->divisor) {
    case 1: _cmd1 |= TDA8023TT_REG1_CLKIN_1; break;
590     case 2: _cmd1 |= TDA8023TT_REG1_CLKIN_2; break;
    case 4: _cmd1 |= TDA8023TT_REG1_CLKIN_4; break;
    case 5: _cmd1 |= TDA8023TT_REG1_CLKIN_5; break;
    default:
        printk (KERN_ERR "%s: _invalid_divisor_(%d)_detected\n",
595             __func__, data->divisor);
        return 1; // break;
    }

    _iouc_set (1);

600     printk (KERN_DEBUG "%s: _initiating_powerup_(sending_cmd_%.2x_%.2x)\n",
             __func__, _cmd0, _cmd1);

    if (tda8023tt_write_reg (data, 1, _cmd1)) {
605         printk (KERN_ERR "%s: _write_reg1_failed ,_aborting\n", __func__);
        return 1;
    }
    if (tda8023tt_write_reg (data, 0, _cmd0)) {
        printk (KERN_ERR "%s: _write_reg0_failed ,_aborting\n", __func__);
610         return 1;
    }

    return 0;
}

615 static void

```

78 Appendix A. ISO 7816 Software UART Driver Source Code

```
tda8023tt_powerdown (struct tda8023tt_data *const data)
{
    u8 _cmd = 0;
620     printk (KERN_DEBUG "%s:_performing_powerdown_(sending_cmd_%2x)\n",
             __func__, _cmd);

    if (tda8023tt_write_reg (data, 0, _cmd))
625     printk (KERN_ERR "%s:_write_reg0_failed\n", __func__);
}

/* much of state-handling gets in here eventually... */
static void
630 tda8023tt_tasklet_bh (unsigned long devl)
{
    struct tda8023tt_data *const data = (void*)devl;
    const s32 status = tda8023tt_read_reg (data, 0);

635     if (status < 0)
        {
            printk (KERN_ERR "%s:_failed_to_read_status_(errno=%d)\n",
                    __func__, status);
            return;
640         }

    printk (KERN_DEBUG "%s:_status_=%2x\n", __func__, status);

    if (status & TDA8023TT_REG0_R_PRESL)
645     {
        if (status & TDA8023TT_REG0_R_PRES)
            {
                if (data->__card_inserted)
                    ifd_notify (&data->_ifd, IFD_EVENT_CARD_OUT);
650                else
                    data->__card_inserted = 1;

                ifd_notify (&data->_ifd, IFD_EVENT_CARD_IN);
            }
655        else
            {
                if (!data->__card_inserted)
                    ifd_notify (&data->_ifd, IFD_EVENT_CARD_IN);
660                else
                    data->__card_inserted = 0;

                ifd_notify (&data->_ifd, IFD_EVENT_CARD_OUT);
            }
        }

665     if ((status & (TDA8023TT_REG0_R_MUTE | TDA8023TT_REG0_R_EARLY))
        == (TDA8023TT_REG0_R_MUTE | TDA8023TT_REG0_R_EARLY))
        {
            printk (KERN_WARNING "tda8023:_ATR_failed,_I/O_was_low_during_reset\n");
670        }
        else if (status & TDA8023TT_REG0_R_MUTE)
            {
                printk (KERN_WARNING "tda8023:_ATR_not_started_within_timelimit\n");
                BUG_ON (status & TDA8023TT_REG0_R_EARLY);
            }
    }
```

```

675     }
        else if (status & TDA8023TT_REG0_R_EARLY)
        {
            printk (KERN_WARNING "tda8023:_ATR_started_to_early?!?\n");
            BUG_ON (status & TDA8023TT_REG0_R_MUTE);
680     }

        if (status & TDA8023TT_REG0_R_PROT)
        {
#warning TODO
685     printk (KERN_WARNING "tda8023:_overload/overheat_detected\n");
        }
    }

    static void
690 tda8023tt_interrupt (int irq, void *dev, struct pt_regs *regs)
    {
        struct tda8023tt_data *data = dev;

        tasklet_schedule (&data->tasklet_bh);
695     tda8023tt_ctl_irq_clear();
    }

    static inline int
700 __parity8 (u8 byte)
    {
        byte ^= byte >> 1;
        byte ^= byte >> 2;
        byte ^= byte >> 4;
705     return byte & 1;
    }

    static inline u16
710 __raw_cframe (const u8 c, const int inverse)
    {
        return inverse
            ? (0xfc00
              | ((1 ^ __parity8 (c)) << 9)
              | ((0xff ^ __reverse8 (c)) << 1))
715     : (0xfc00
        | (__parity8 (c) << 9)
        | (c << 1));
    }

720     static int
    _ifd_start (struct ifd_device *dev, int warm, ifd_device_class_t class)
    {
        struct tda8023tt_data *const data = dev->private_data;

725     if (!data->__card_inserted)
        {
            printk (KERN_ERR "%s:_called ,_but_no_card_inserted\n", __func__);
            return -1;
        }

730     switch (class) {
        case IFD_DEVICE_CLASS_A:

```

```
    case IFD_DEVICE_CLASS_B:
    case IFD_DEVICE_CLASS_C:
735     data->__start_class = class;
        break;
    default:
        return -1;
}
740
data->__start_warm = warm ? 1 : 0;

schedule_task (&data->_task_start);

745     return 0;
}

////////////////////////////////////
// irq handling fun
750
#define FIFO_SIZE (1<<9) // must be 2^n
static u16 io_buf[FIFO_SIZE] = { 0, };
static unsigned io_widx = 0, io_ridx = 0;

755 static void
_bh_fifo_flush (void)
{
    memset (io_buf, 0x00, sizeof(io_buf));
    io_widx = io_ridx = 0;
760 }

static void
_bh_fifo_put (u16 c)
{
765     io_buf[io_widx & (FIFO_SIZE-1)] = c;
    wmb();
    ++io_widx;
    mb();
}

770 static inline int
_bh_fifo_get (u16 *c)
{
    if (io_ridx < io_widx) {
775         *c = io_buf[io_ridx++ & (FIFO_SIZE-1)];
        return 0;
    }
    return 1;
}

780 static void task_rx_to (void *_dev);

// gets called from timer
static void
785 timer_rx_to (unsigned long devl)
{
    struct tda8023tt_data *const data = (void*)devl;

    tda8023tt_io_irq_off();
790     schedule_task (&data->_task_rx_to);
}
```

```

// ^^^^ deferred ifd_notify (&data->_ifd , IFD_EVENT_RX_TIMEOUT);
}

static void
795 tda8023tt_tasklet_io_bh (unsigned long devl)
{
    struct tda8023tt_data *const data = (void*)devl;

    u16 c;

800     while (!_bh_fifo_get (&c)) {
        data->frames_rx++;

        if (unlikely((c & 0x601) != 0x400)) { // parity/framing error
805         //printk ("bad frame %.2x\n", c << 3);
            if (c & 0x200)
                data->frames_perrors++;
            else
                data->frames_badframes++;
810         }

        c >>= 1;
        c &= 0x1ff;

815         if (!ifd_frame_rx (&data->_ifd , c))
            {
                tda8023tt_io_irq_off();
                del_timer_sync(&data->_timer_rx_to);
                schedule_task (&data->_task_rx_to);
820                 // ^^^^ deferred ifd_notify (&data->_ifd , IFD_EVENT_RX_TIMEOUT);
                break;
            }
        }
    }
825 }

////////////////////////////////////

static void
830 tda8023tt_io_interrupt (int irq, void *dev, struct pt_regs *regs)
{
    // all irqs are disabled, since we are SA_INTERRUPT!
    struct tda8023tt_data *data = dev;

    unsigned curr_frame = 0;
835     unsigned idx;

    //tda8023tt_io_irq_clear();
    tda8023tt_io_irq_off();
    tda8023tt_reset_counter ();

840     // udelay()

    //BUG_ON (!_iouc_get ());

845     // scan for first edge...

    for (idx = 0; idx < 1+8+1+1; ++idx)
        {

```

82 Appendix A. ISO 7816 Software UART Driver Source Code

```

    tda8023tt_consume_ticks_nowait (ETU(1,0));
850
    while (!tda8023tt_counter_synced ())
        if (_iouc_get ())
            goto edge_detected;
    }
855
edge_detected:
    tda8023tt_reset_counter ();

    // move into bit-cell middle
860    tda8023tt_consume_ticks_nowait (ETU(0,TICKS_PER_ETU/2));

    for (++idx; idx < 1+8+1+1; ++idx)
    {
        int curr_bit = 0;
865
        tda8023tt_sync_deadline ();

        curr_bit = _iouc_get ();

870        curr_frame |= curr_bit << idx;

        curr_frame ^= curr_bit << 9; // xor parity on the fly

        tda8023tt_consume_ticks_nowait (ETU(1,0));
875    }

    curr_frame ^= (curr_frame >> 1) & (1<<9); // un-parity stop-bit

    //if (curr_frame & 0x401 != 0x400) { ... } // framing error
880
    if (unlikely(curr_frame & (1<<9))
        && data->ifd.parm.flag_parity_signaling)
    {
        // perform error signalling, and re-sample frame
885        // data->frames_tx++;
    }

    _bh_fifo_put (curr_frame);
    tda8023tt_io_irq_on ();
890    tasklet_schedule (&data->tasklet_io_bh);
}

static void
task_start (void *_dev)
895 {
    struct ifd_device *dev = _dev;
    struct tda8023tt_data *const data = dev->private_data;
    unsigned long flags;

900    BUG_ON(in_interrupt ());

    // try to avoid doing resets before 5ms have passed since last operation
    tda8023tt_mdelay (5, 0);

905    if (!data->__card_inserted)
    {
```

```

        printk (KERN_ERR "%s: _card_removed_unexpectedly, _aborting_call...\n",
                __func__);
        return;
910     }

    local_irq_save (flags);

    // reset state
915     _bh_fifo_flush ();
    data->wretrans = data->rretrans = 0;
    data->frames_tx = data->frames_rx = 0;
    data->frames_badframes = data->frames_perrors = 0;

920     data->tx_commit = 1;
    data->outbuf_len = 0;

    tda8023tt_timer_setup (data->divisor, 372, 1); // Dd = 1, Fd = 372
    if (!_iouc_get ())
925     printk (KERN_WARNING "I/O_line_is_down\n");

    if (tda8023tt_powerup (data)) {
        printk (KERN_ERR "%s: _tda8023tt_powerup_failed\n", __func__);
        restore_flags (flags);
930     ifd_notify (&data->_ifd, IFD_EVENT_RX_TIMEOUT);
        return;
    }

    tda8023tt_reset_counter ();
935     tda8023tt_io_irq_on(); // --> io irq handler...
    restore_flags (flags);
}

static int
940 _ifd_stop (struct ifd_device *dev)
{
    struct tda8023tt_data *const data = dev->private_data;

    tda8023tt_powerdown (data);
945     return 0;
}

static int
950 _ifd_eject (struct ifd_device *dev)
{
    printk (KERN_WARNING "tda8023: _warning: _card_ejection_emulated!\n");

    return 0;
955 }

static int
_ifd_frames_tx (struct ifd_device *dev, const u8 frames[], unsigned len,
                int more)
960 {
    struct tda8023tt_data *const data = dev->private_data;
    unsigned idx;

    if (data->tx_commit) {

```

84 Appendix A. ISO 7816 Software UART Driver Source Code

```
965     data->outbuf_len = 0;
      data->tx_commit = 0;
    }

    if (data->outbuf_len + len > IFD_OUTBUF_MAXSIZE)
970     {
        printk (KERN_ERR "%s: output buffer space exhausted (%d > %d)\n",
                __func__, data->outbuf_len + len, IFD_OUTBUF_MAXSIZE);
        data->tx_commit = 1;
        return -1;
975     }

    for (idx = 0; idx < len; ++idx)
        data->outbuf[data->outbuf_len++] =
            _raw_cframe (frames[idx], dev->parm.flag_inverse);
980

    if (!more) {
        data->tx_commit = 1;
        schedule_task (&data->_task_tx);
    }
985

    return 0;
}

static int
990 _ifd_set_rate (struct ifd_device *dev, unsigned F, unsigned D, int try)
{
    struct tda8023tt_data *const data = dev->private_data;
    return (try ? tda8023tt_timer_setup (
995                                     (data->divisor, F, D);
    )
    )

static void
task_tx (void *_dev)
{
1000     struct ifd_device *dev = _dev;
    struct tda8023tt_data *const data = dev->private_data;
    unsigned long flags;

    BUG_ON(in_interrupt());
1005

    if (!data->_card_inserted)
    {
        printk ("%s: card removed unexpectedly, aborting call...\n", __func__);
        return;
1010    }

    local_irq_save (flags);

    // reset state
1015     _bh_fifo_flush ();
    data->wretrans = data->rretrans = 0;
    data->frames_tx = data->frames_rx = 0;
    data->frames_badframes = data->frames_perrors = 0;

1020     if (!_iouc_get ())
        printk ("I/O line is down!\n");
}
```

```

// write routine
{
1025   unsigned outbuf_idx;

   const u16 _guard_ticks = ETU(dev->parm.guard_bits, 0);

   if (dev->parm.delay_bits > 0)
1030     tda8023tt_consume_ticks (ETU(dev->parm.delay_bits, 0));

   tda8023tt_reset_counter ();

   for (outbuf_idx = 0; outbuf_idx < data->outbuf_len; ++outbuf_idx)
1035     {
       const u16 _cframe = data->outbuf[outbuf_idx];
       int idx;

       __distract_watchdog ();

1040     resend:
       tda8023tt_sync_deadline ();

       for (idx = 0; idx != 10; ++idx) {
1045         _iouc_set ((_cframe >> idx) & 1);
         tda8023tt_consume_ticks (ETU(1,0));
       }

       _iouc_set (1); // stop bit(s)

1050     data->frames_tx++;

     if (dev->parm.flag_parity_signaling)
       {
1055         tda8023tt_consume_ticks (ETU(1,0)); // check state at 11+/-0.2 etu
         if (likely(_iouc_get ()))
           { // no error signal -> consume remaining _guard_ticks
             tda8023tt_consume_ticks_nowait (_guard_ticks - ETU(1,0));
           }
1060         else
           { // error signalling :-(
             if (data->wretrans > MAX_RETRANS)
               break; // too many retransmissions
             data->wretrans++;

1065             // wait at least 2etu's after detection of error signal
             tda8023tt_consume_ticks_nowait (ETU(2,0));

             goto resend;
           }
1070       }
     else // no parity error signalling necessary
       tda8023tt_consume_ticks_nowait (_guard_ticks);
1075   }

   // no tda8023tt_sync_deadline () in order to catch IRQ and improve latency

   __distract_watchdog ();
} // _do_write ()
1080

```

```
    mod_timer (&data->_timer_rx_to , jiffies + HZ); // 1sec timeout (incorrect)

    tda8023tt_reset_counter ();

1085    tda8023tt_io_irq_on ();
    local_irq_restore (flags);

    // irq based sampling takes over here...

1090    // at end of rx transmission task_rx_to() gets called
}

static void
task_rx_to (void *_dev)
1095 {
    struct ifd_device *dev = _dev;
    struct tda8023tt_data *const data = dev->private_data;

    // hopefully there should be only one task_rx_to() call
1100    // scheduled... but this driver is a hack-of-concept anyway

    // FIXME - check this is really true...

    data->_ifd.stats.frames_tx += data->frames_tx;
1105    data->_ifd.stats.frames_tx_errors += data->wretrans;

    data->_ifd.stats.frames_rx += data->frames_rx;
    data->_ifd.stats.frames_rx_errors += data->frames_badframes
        + data->frames_perrors;

1110    ifd_notify (&data->_ifd , IFD_EVENT_RX_TIMEOUT);
}

static int
1115 tda8023tt_probe_client (struct i2c_adapter* adapter , int address ,
                        unsigned short flags , int kind)
{
    int err = -1;
    struct i2c_client *new_client = 0;
1120    struct tda8023tt_data *new_data = 0;

    printk (KERN_DEBUG "tda8023tt_probe_client(,%d,%d,%d)\n" ,
            address , flags , kind);

1125    if (tda8023tt_instances > 0)
    {
        printk (KERN_ERR "only_one_tda8023_driver_instance_supported\n");
        goto failout;
    }

1130    new_client = kmalloc (sizeof *new_client , GFP_KERNEL);

    if (!new_client) {
        printk ("kmalloc_failed\n");
1135        goto failout;
    }

    new_client->addr = address;
```

```

new_client->adapter = adapter;
1140 new_client->driver = &tda8023tt_driver;
new_client->flags = 0;
strcpy (new_client->name, "tda8023_client");

new_client->data = new_data = kmalloc (sizeof *new_data, GFP_KERNEL);
1145
if (!new_data) {
    printk ("kmalloc_failed\n");
    goto failout;
}
1150
memset (new_data, 0, sizeof *new_data);
new_data->client = new_client;

{
1155     struct ifd_device *const _ifd = &new_data->_ifd;
    _ifd->owner = THIS_MODULE;
    strcpy (_ifd->name, "tda8023tt");

    _ifd->private_data = new_data;
1160     _ifd->start = _ifd_start;
    _ifd->stop = _ifd_stop;
    _ifd->eject = _ifd_eject;
    _ifd->frames_tx = _ifd_frames_tx;
    _ifd->set_rate = _ifd_set_rate;
1165     _ifd->supported_classes = IFD_DEVICE_CLASS_A;
}

PREPARE_TQUEUE (&new_data->_task_start, task_start, new_data);
PREPARE_TQUEUE (&new_data->_task_tx, task_tx, new_data);
1170 PREPARE_TQUEUE (&new_data->_task_rx_to, task_rx_to, new_data);
init_timer(&new_data->_timer_rx_to);
new_data->_timer_rx_to.data = (unsigned long)new_data;
new_data->_timer_rx_to.function = &timer_rx_to;
tasklet_init (&new_data->tasklet_bh, tda8023tt_tasklet_bh,
1175             (unsigned long)new_data);
tasklet_init (&new_data->tasklet_io_bh, tda8023tt_tasklet_io_bh,
             (unsigned long)new_data);

new_data->divisor = DEFAULT_CLKDIV; /* safe choice */
1180
if ((err = i2c_attach_client (new_client)))
    goto failout;

if ((err = request_irq (tda8023tt_ctl_irq, tda8023tt_interrupt,
1185                     SA_INTERRUPT,
                     "TDA8023TT_status", new_data)))
    {
        printk ("failed_to_request_status_irq_%d\n", tda8023tt_ctl_irq);
        goto failout;
1190    }

if ((err = request_irq (tda8023tt_io_irq, tda8023tt_io_interrupt,
1195                     SA_INTERRUPT,
                     "TDA8023TT_I/OuC", new_data)))
    {
        printk ("failed_to_request_I/OuC_irq_%d\n", tda8023tt_io_irq);
    }

```

```
        goto failout;
    }
1200   if ((err = ifd_register_dev (&new_data->_ifd)) {
        printk (KERN_ERR "ifd_register_dev() failed (rc=%d)\n", err);
        goto failout;
    }

1205   /* setup I/OuC pin */
        _iouc_init ();

        /* reset tda8023 */
1210   if (tda8023tt_write_reg (new_data, 0, 0x00))
        printk ("write_reg0 failed\n");

        if (tda8023tt_write_reg (new_data, 1, TDA8023TT_REG1_INIT))
            printk ("write_reg1 failed\n");

1215   if (!_iouc_get ())
        {
            printk (KERN_ERR "tda8023: I/OuC not high!\n");
            goto failout;
        }

1220   if (tda8023tt_write_reg (new_data, 0, TDA8023TT_REG0_W_IOEN))
        printk ("write_reg0 failed\n");

        /* enable irq */
1225   tda8023tt_ctl_irq_on ();

        tda8023tt_instance_data [tda8023tt_instances] = new_data;
        tda8023tt_instances++;

1230   {
        const s32 status = tda8023tt_read_reg (new_data, 0);

        printk (KERN_DEBUG "tda8023: status = 0x%.2x/0x%.2x\n", status,
            tda8023tt_read_reg (new_data, 1));

1235         if (status & TDA8023TT_REG0_R_PRES) {
            printk (KERN_INFO "card detected already inserted\n");
            new_data->_card_inserted = 1;
            ifd_notify (&new_data->_ifd, IFD_EVENT_CARD_IN);
1240         } else
            new_data->_card_inserted = 0;
        }

        return 0;

1245   failout:
        /* FIXME */
        free_irq (tda8023tt_ctl_irq, new_data);
        free_irq (tda8023tt_io_irq, new_data);
1250   kfree (new_data);
        kfree (new_client);

        return err;
    }
}
```

```

1255 static int __init
tda8023tt_init(void)
{
1260     if (tda8023tt_debug)
        printk (KERN_INFO "%s: Rev: 737, irq %d, CLKDIV %d, compiled %s %s\n",
                __func__, tda8023tt_ctl_irq, DEFAULT_CLKDIV, __TIME__, __DATE__);

    return i2c_add_driver (&tda8023tt_driver);
}
1265 static void __exit
tda8023tt_exit (void)
{
1270     tda8023tt_powerdown (tda8023tt_instance_data [0]);

    /* disable irq */
    tda8023tt_ctl_irq_off ();

1275     i2c_del_driver (&tda8023tt_driver);

MODULE_AUTHOR(" Herbert_Valerio_Riedel <hvr@inso.tuwien.ac.at>");
MODULE_DESCRIPTION("TDA8023TT/ep930x driver");
MODULE_PARM(tda8023tt_debug, "i");
1280 MODULE_PARM_DESC(tda8023tt_debug, "debug level");
MODULE_LICENSE("GPL");

module_init(tda8023tt_init);
module_exit(tda8023tt_exit);

```

Listing A.1: ISO 7816 software UART driver source code

APPENDIX B

Latency Measurement Kernel Module Source Code

```
1 // simple linux 2.4 kernel module for interrupt latency measurements
  // dumps latency time histogram to console on module unload

#include <linux/config.h>
#include <linux/kernel.h>
6 #include <linux/spinlock.h>
#include <linux/interrupt.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/sched.h>
11 #include <linux/version.h>
#include <linux/delay.h>
#include <linux/module.h>
#include <asm/io.h>
#include <asm/arch/regmap.h>
16 #include <asm/arch/irqs.h>
#include <asm/arch/system.h>

static int DIV = 1;
static u32 calibrated_tc3_tc4 = 0; // about 493

21 #define EP93XX_TIMER3_FREQ    1994UL
#define TRIGGER_INTERVAL      (EP93XX_TIMER3_FREQ/DIV) // DIV per sec

#define EP93XX_TIMER4_FREQ    983040UL // 14.7456/15 Mhz = 983.04 KHz
26 #define TRIGGER_INTERVAL_TC4 (EP93XX_TIMER4_FREQ/DIV) // DIV per sec

#define EP93XX_TIMER4_ENABLE  0x100

/* Sets up a fast hardware timer with 983.04 KHz frequency
31 * in free running mode
 */
static void
ep93xx_timer4_init(void)
{
36 /* reset the timer */
```

92 Appendix B. Latency Measurement Kernel Module Source Code

```
    outl(0x0, TIMER4VALUEHIGH);

    /* enable the timer */
    outl(EP93XX_TIMER4_ENABLE, TIMER4VALUEHIGH);
41 }

static void
ep93xx_timer3_init(u32 interval) // interval=0 ==> disabled
{
46     outl(0x0, TIMER3CONTROL); // disable tc3...
    outl(interval, TIMER3LOAD); // pre-load with interval...

    if (interval)
51     outl(0xc0, TIMER3CONTROL);
}

static inline u32
ep93xx_timer4_get(void)
{
56     // return inl(TIMER4VALUEHIGH) << 32 | inl(TIMER4VALUELOW);
    //
    // alas 64bit arithmetic seems to take too long to be useful, so we
    // simply ignore the upper 8 bits overflow in the lower 32bits
    // occurs every ~4369s (~73m), long enough for our purposes
61     return inl(TIMER4VALUELOW);
}

static inline u32
ep93xx_timer3_get(void)
66 {
    return inl(TIMER3VALUE);
}

static inline void
71 ep93xx_timer3_clear_irq(void)
{
    outl(0x0, TIMER3CLEAR);
}

76 #define MAX_DELAY 500
volatile static unsigned buckets[MAX_DELAY] = { 0, };
volatile static unsigned samples = 0;

static irqreturn_t
81 lat_timer3_isr(int irq, void *data, struct pt_regs *regs)
{
    const u32 _timer4 = ep93xx_timer4_get ();
    const u32 _timer3 = ep93xx_timer3_get ();

86     ep93xx_timer3_clear_irq ();

    while (_timer3 == ep93xx_timer3_get ()) {}; // wait till next count...

91     const u32 _ntimer4 = ep93xx_timer4_get ();
    const unsigned timer4delta = calibrated_tc3_tc4 - (_ntimer4 - _timer4);

    ++samples;
}
```

```

96     if (_timer3 == 0 && timer4delta < MAX_DELAY)
        ++buckets[timer4delta];

        return IRQ_HANDLED;
    }
101
static int __init
ep93_lat_init (void)
{
106     printk ("%s_[compiled_%s_%s]\n", __func__, __TIME__, __DATE__);

    if (request_irq (IRQ_TIMER3, lat_timer3_isr, SA_INTERRUPT,
                    "ep93_lat_timer_irq", 0))
    {
111         printk ("failed_to_request_status_irq_%d\n", IRQ_TIMER3);
        return -1;
    }

    // reset data
116     int idx;
    for (idx = 0; idx < MAX_DELAY; ++idx)
        buckets[idx] = 0;
        samples = 0;

121     printk ("calibrating...\n");

    local_irq_disable();

    ep93xx_timer3_init(TRIGGER.INTERVAL-1); // try with DIV irq/sec rate...
126     ep93xx_timer4_init();

    const u32 _timer3 = ep93xx_timer3_get ();
    while (_timer3 == ep93xx_timer3_get ()) {};

131     const u32 _ntimer4 = ep93xx_timer4_get ();

    const u32 _ntimer3 = ep93xx_timer3_get ();
    while (_ntimer3 == ep93xx_timer3_get ()) {};

136     const u32 _mtimer4 = ep93xx_timer4_get ();

    // calibrated_tc3_tc4 represents the number of TC4 ticks for one TC3 tick
    calibrated_tc3_tc4 = _mtimer4 - _ntimer4;

141     local_irq_enable();

    printk ("calibrated_tc3_tc4_=%u\n", calibrated_tc3_tc4);

    return 0;
146 }

static void __exit
ep93_lat_exit (void)
{
151     printk ("%s\n", __func__);

```

```
ep93xx_timer3_init(0); // disable...
udelay(5); // try to avoid race condition

156  unsigned overflowed = samples;

      printk ("latency_histogram_(%u_samples@_interval_%d/1993.9Hz):\n",
              samples, (unsigned)TRIGGER_INTERVAL);

161  unsigned idx;
      for (idx = 0; idx < MAX_DELAY; ++idx) {
          if (buckets[idx])
              printk ("%3d_%8u\n", idx, buckets[idx]);
              overflowed -= buckets[idx];
166  }

      printk (>%d_%8u\n", MAX_DELAY-1, overflowed);
      printk ("EOF\n");

171  free_irq (IRQ_TIMER3, 0);
      }

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("latency_measurements_for_ep93xx\n");
176  MODULE_AUTHOR("Herbert_Valerio_Riedel_<hvr@inso.tuwien.ac.at>");

      module_init(ep93_lat_init);
      module_exit(ep93_lat_exit);

181  MODULE_PARM_DESC(DIV, "DIV");
      MODULE_PARM(DIV, "i");
```

Listing B.1: Latency measurement kernel module source code

Glossary

- Application Protocol Data Unit (APDU)** Message structure at application level (can be either command APDU or response APDU)[14], 19
- Application Programming Interface (API)** Set of defined functions representing the programming interface to a library or operating system services, 1, 37
- Asynchronous Software Thread Integration (ASTI)** Software Transformation Technique for inlining multiple threads of control into a single routine, 26
- Answer-To-Reset (ATR)** The sequence of bytes sent by a smart-card on reset, 18
- Complementary Metal-Oxide-Semiconductor (CMOS)** Major class of integrated circuits, see also TTL, 4
- Complex Programmable Logic Device (CPLD)** A programmable logic device, see also FPGA, 9
- Central Processing Unit (CPU)** Component of a microprocessor responsible for software execution, 7
- Cyclic Redundancy Check (CRC)** A type of hash function used for generating a check-sum over a block of data, 16, 18
- Carrier Sense Multiple Access (CSMA)** non-deterministic media access protocol based on detecting absence of other carriers before initiating transmission on a shared medium, 3
- Direct Memory Access (DMA)** The use of DMA allows subsystems direct access to memory independently from the CPU, 6
- Elementary Time Unit (etu)** The basic time unit denoting the bit-duration in the ISO 7816-3 standard[15], 17
- Frequency Division Duplex (FDD)** FDM used for direction duplexing, 3
- First In First Out (FIFO)** The way data is handled in data structures or hardware components resembling a queue which store data items of which the items added first are also the ones to be removed first, 4

Field-Programmable Gate Array (FPGA) An integrated circuit which contains programmable logic components and interconnects, see also CPLD, 9

General Purpose Input Output (GPIO) Flexible hardware component whose pins can be configured to either input or output signals, 8, 35

I²C Serial bus standard[27] for connecting on-board components, 2

Integrated Circuit (IC) miniaturised electronic (semiconductor) circuit, also called microchip, 4

Integrated Circuit(s) Card (ICC) The technical term used in the ISO 7816 specification to refer to small plastic cards with embedded integrated circuits, sometimes also called *Smart-cards*, 6, 19

ISO/IEC 7816 International standard[15, 14] defining contact-type ICCs, 6

Jitter Deviation from the ideal timing of an event, 22

Memory Management Unit (MMU) Computer component handling memory accesses, 35

Printed Circuit Board (PCB) Board on which electronic components are attached and interconnected, 1, 11

Portable Operating System Interface (POSIX) IEEE Standard which defines the user-space API for Unix-like operating systems, 36

Protocol-and-Parameters-Selection (PPS) The handshake protocol for negotiating communication protocol parameters with smart-cards, 18

Real-Time Operating System (RTOS) Operating system providing facilities to facilitate the implementation of *real-time* applications, 1

Serial Communications Interface (SCI) Synonymous with UART, 4

Space Division Duplex (SDD) SDM used for direction duplexing, 3

System Management Bus (SMBus) Serial bus standard[31] based on the I²C protocol used for low-speed system management communications, 2

Time Division Duplex (TDD) TDM used for direction duplexing, 3

Translation Look-aside Buffer (TLB) Cache component of the MMU for accelerating memory address translation, 35

Transmission Protocol Data Unit (TPDU) APDU transformed for transmission[15, 14], 19

Transistor-Transistor Logic (TTL) Common type of digital circuit which is composed of transistors, 4

Universal Asynchronous Receiver/Transmitter (UART) Serial asynchronous communication protocol controller component, i, ii, 4, 7–10, 13, 25–28, 35, 36

Universal Synchronous Asynchronous Receiver/Transmitter (USART) An UART with additional support for synchronous operation, 4

Universal Serial Bus (USB) Serial bus standard[36] for connecting devices to a host-system in a tree-like topology, 2

VHSIC Hardware Description Language (VHDL) Commonly used design-entry language for FPGAs and ASICs, 28

Bibliography

- [1] Vasanth Asokan and Alexander G. Dean. Providing time- and space- efficient procedure calls for asynchronous software thread integration. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 167–178, New York, NY, USA, 2004. ACM Press.
- [2] Atmel. *AVR304: Half Duplex Interrupt Driven Software UART*, August 1997.
- [3] Atmel. *AVR310: RC5 IR Remote Control Receiver*, May 2002.
- [4] Cirrus Logic, Inc. *EP9301 User's Guide*, February 2004.
- [5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition, 2005.
- [6] CORPORATE IEEE, Inc. Staff, New York, NY, USA. *IEEE Std 802.16-2004 IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems*, 2004.
- [7] C. Ebner. Description and comparison of selected UARTs. Technical Report 22/1994, TU Vienna, Institut fuer Technische Informatik, 1994.
- [8] Wilfried Elmenreich and Martin Delvai. Time-triggered communication with UARTs. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems*, pages 97–104, Aug. 2002.
- [9] Alexey Kuznetsov et al. Iputils package. <ftp://ftp.inr.ac.ru/ip-routing/iputils-ss020927.tar.gz>.
- [10] Linus Torvalds et al. Linux kernel. <http://www.kernel.org/>.
- [11] Scott George. *HC05 MCU Software-Driven Asynchronous Serial Communication Techniques using the MC68HC705J1A*. Freescale Semiconductor, 1995.
- [12] Greg Goodhue. *A software duplex UART for the 751/752*. Philips Semiconductors, June 1993.

- [13] Steve Heath. *Embedded Systems Design*. Butterworth-Heinemann, Newton, MA, USA, 2002.
- [14] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 7816-4:1995: Identification cards — Integrated circuit(s) cards with contacts — Part 4: Interindustry commands for interchange*, first edition, September 1995.
- [15] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 7816-3:1997: Identification cards — Integrated circuit(s) cards with contacts — Part 3: Electronic signals and transmission protocols*, second edition, December 1997.
- [16] International Telecommunication Union, Geneva, Switzerland. *ITU-R Recommendation M.1677: International Morse Code*, May 2004.
- [17] Adrienne Prahler Jaffe. *Implementation of a Software UART on TMS320C54x Using General-Purpose I/O Pins*. Texas Instruments, July 1999.
- [18] A.J. Jerri. The Shannon Sampling Theorem - Its Various Extensions and Applications: A Tutorial Review. In *Proceedings of the IEEE*, volume 65, pages 1565–1596, November 1977.
- [19] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 37–49, New York, NY, USA, 2001. ACM Press.
- [20] H Leon W. Couch. *Digital and Analog Communication Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, sixth edition, 2001.
- [21] D. Lioupis, A. Papagiannis, and D. Psihogiou. A systematic approach to software peripherals for embedded systems. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 140–145, New York, NY, USA, 2001. ACM Press.
- [22] D. Lioupis, A. Papagiannis, D. Psihogiou, and M. Stefanidakis. Software peripherals requirements and constraints for real-time embedded systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.
- [23] National Communications System. *Federal Standard 1037C: Telecommunications: Glossary of Telecommunication Terms*, August 1996.
- [24] National Semiconductor. *PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs*, June 1995.

-
- [25] H. Nyquist. Certain topics in telegraph transmission theory. In *Proceedings of the IEEE*, volume 90, pages 280–305, 2002.
- [26] Philips Semiconductors. *SAA3010 Infrared remote control transmitter RC-5*, June 1989.
- [27] Philips Semiconductors. *The I²C-Bus Specification Version 2.1*, January 2000.
- [28] Philips Semiconductors. *TDA8023 – Low power IC card interface*, April 2004.
- [29] Wolfgang Rankl and Wolfgang Effing. *Handbuch der Chipkarten – Aufbau – Funktionsweise – Einsatz von Smart-Cards*. Carl Hanser Verlag Munchen Wien, fourth edition, 2002.
- [30] Vincent Sanders. Linux 2.4.26-vrs1 kernel patch. <ftp://ftp.arm.linux.org.uk/pub/armlinux/kernel/v2.4/patch-2.4.26-vrs1.bz2>.
- [31] SBS Implementers Forum. *System Management Bus (SMBus) Specification, Version 2.0*, August 2000.
- [32] C.E. Shannon. Communication in the presence of noise. In *Proceedings of the IEEE*, volume 86, pages 447–457, 1998.
- [33] Inc. Software in the Public Interest. Debian GNU/Linux 3.1. <http://www.debian.org/releases/sarge/>.
- [34] Texas Instruments, Inc. *MAX232, MAX232I – Dual EIA-232 Drivers/Receivers*, February 1989.
- [35] Philipp Tomsich. UX/SC Proposal. 2004.
- [36] USB Implementers Forum, Inc. *Universal Serial Bus Revision 2.0 Specification*, 2000.
- [37] Dave Walsh. Reducing system cost with software modems. *IEEE Micro*, 17(4):37–43, 1997.
- [38] Hubert Zimmermann. The ISO reference model for open systems interconnection. In *Proceedings on Kommunikation in Verteilten Systemen*, pages 39–57, London, UK, 1981. Springer-Verlag.

Colophon

This thesis was prepared and typeset using L^AT_EX 2_ε. No proprietary software has been used, nor have any animals been harmed—except for those few that have been sacrificed for nourishment of the author. For the purpose of this thesis a document-class of its own has been created, based on the versatile KOMA-Script *Scrbook* L^AT_EX class. The following additional L^AT_EX packages have been used:

afterpage With `afterpage`, one can specify a command that will be executed after the current page is finished.

caption The `caption` package provides many ways to customise the captions in floating environments.

glossary The glossary in this thesis was automatically generated with the help of this package, which has support for managing acronyms as well.

hyperref A package for adding PDF related meta data and navigational information.

listings Package for typesetting source code listings with syntax highlighting.

titlesec Allows for customisation of the L^AT_EX chapter and section titles. This was used for creating the fancy headings at the beginning of each chapter.

titletoc With this, one is enabled to precisely control the table-of-contents entries.

varioref Standard package that provides “decorative intelligent” cross-references.

All of the figures, except the PC16550D block-diagram which was extracted from the corresponding data-sheet, were created with the help of METAPOST, a tool for describing technical diagrams geometrically—which admittedly needs getting used to, just as T_EX does.

The data used for the diagrams was extracted from log-files and processed with the help of python scripts, which transformed it into the appropriate input-files for Gnuplot, a tool for plotting graphs from functions and data series. Gnuplot was instructed to output the diagrams in METAPOST format, in order to have a uniform work-flow with

respect to figure inclusion, furthermore this also facilitated the use of \LaTeX typesetting commands in the diagrams.

The state machine diagram (figure 3.1 on page 31) was created with the help of Graphviz, a tool for automatic graph visualisation, which was instructed to output in METAPOST format as well.

All of the tools above were held together by the use of a sophisticated Make-script, which triggered the execution of each of the above mentioned tools in the build process as required by the dependencies specified in the makefile.

Combined with the AUCTEX package (which includes the Preview- \LaTeX mode) and the Flyspell mode, GNU Emacs provided an invaluable editing environment. The document source files were kept under revision control with the help of Subversion.

This thesis is typeset using the Computer Modern fonts designed by Donald E. Knuth, which is also the creator of the TEX system upon which \LaTeX is based.

