Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMA THESIS

Design and Implementation of a VOIP Stack for High Packet Rates in an FPGA

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ. Prof. Dr. habil. Christoph Grimm
Dipl. Inf. Dr. Jan Haase
Dipl.-Ing. (FH) Peter Brunmayr

Institute number: 384
Institute of Computer Technology

by

Matthias Wenzl, Bsc.
Matr.Nr. 0425388
Elisabethallee 39, 1130 Vienna

Date _____

Declaration

Matthias Wenzl, BSc.
Elisabethallee 39
1130 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 25.3.2011,

Kurzfassung

Dank einer breiteren Verfügbarkeit von Internetanschlüssen mit entsprechender Bandbreite erfreut sich Voice over Internet Protocol (VOIP) Kommunikation im Privaten wie im Geschäftsbereich in industrialisierten Ländern immer größerer Beliebtheit. Die meisten VOIP-fähigen Endgeräte sind jedoch nicht auf die speziellen Bedürfnisse einiger Einrichtungen, wie zum Beispiel der Luftraumüberwachung, dem öffentlichen Verkehr, oder militärischen Einrichtungen, abgestimmt.
Um diesen speziellen Anforderungen besser Rechnung tragen zu können wurde im Rahmen eines Forschungsprojektes ein spezielles Hardwaremodule entwickelt, welches sämtliche Aufgaben den Sprachdatenaustausch einer VOIP Verbindung betreffend in Hardware realisiert. Dieser Intellectual Property Core (IP-Core), genannt Real-time Transport Protocol (RTP)-Engine, wird in einer standalone Konfiguration in einem Xilinx Virtex4 Field Programmable Gate Array (FPGA) auf einer Prototypingplattform betrieben. Die notwendigen Kommunikationsparameter werden über statische Konfigurationsvektoren gesetzt. Es ist die Aufgabe dieser Diplomarbeit die Konfigurationsschnittstelle der RTP-Engine so zu erweitern, dass es zur Laufzeit entsprechend den Anforderungen einer VOIP-Verbindung konfiguriert werden kann. Ziel dieser Arbeit ist die Implementierung eines kompletten VOIP-Stacks bestehend aus einem Signalisierungsprotokoll (Verbindungsaufbau - Abbau), einem Verbindungsqualitätssicherungsprotokoll, und der RTP-Engine inklusive der zu entwickelnden Konfigurationsschnittstelle. Als Randbedingungen treten hierbei Embedded System spezifische Beschränkungen bezüglich limitierter Rechenleistung und eingeschränkten FPGA Resourcen auf. Diese haben direkten Einfluss auf den VOIP-Stack Designprozess. Des weiteren müssen gewisse Auflagen betreffend der Hardware/Software Schnittstelle der RTP-Engine, welche durch die vorgegeben VOIP-Stack Funktionalität induziert sind, eingehalten werden.
Um die Aufgabenstellung entsprechend umsetzen zu können wird eine Evaluierung des grundlegenden Aufbaus eines VOIP-Stacks durchgeführt und die Anforderungen an das Design eines Basis-Systems und der Hardware/Software Schnittstelle der RTP-Engine erhoben. Die so erhaltenen Anforderungen sowie Randbedingungen fließen in die Auswahl entsprechender Software und Hardware Komponenten ein. Außerdem wird der Einsatz unterschiedlicher Software-Stacks, Schnittstellendesigns und User/Kernel -Space Separierungen diskutiert.
Schließlich wird ein funktionstüchtiger Prototyp implementiert, welcher in der Lage ist, einen VOIP-Sitzung direkt zwischen zwei Prototypingplattformen oder einem Personal Computer (PC) zu realisieren. Es wird gezeigt, dass eine den Anforderungen entsprechend entwickelte und umgesetzte Schnittstelle eines IP-Cores welcher den unmittelbaren Sprachdatenaustauch zwischen zwei Kommunikationspartnern durchführt, eine wesentliche Freisetzung von Rechenzeit eines Prozessors mit sich bringt und somit entsprechende Echtzeitanforderungen leichter erfüllt werden können. Deshalb wäre es möglich, eine Erweiterung des Prototypen durchzuführen, um Konferenzschaltungen zu unterstützen.

Abstract

In recent years, Voice over Internet Protocol (VOIP) communication has become an interesting alternative to the Public Switched Telephone Network (PSTN) for casual and business phone-calls in industrialized countries. Most of the available VOIP capable end-devices are considered for general purpose use, thus neglecting firmer real-time requirements of certain operational environments, like air traffic management, public transport, or defense.
In order to accommodate these requirements, a dedicated hardware module performing all tasks necessary to exchange voice packets between two communication partners, called Real-time Transport Protocol (RTP)-Engine, has been developed in a research project. The Intellectual Property Core (IP-Core) is operated standalone on a Xilinx Virtex4 Field Programmable Gate Array (FPGA) sited on a prototyping board. The necessary configuration parameters are supplied through static configuration vectors. It is the objective of this diploma thesis to extend the module's interface in such a way that it can be configured in a VOIP-session appropriate way at run-time. The aim of this work is to design and implement a complete VOIP-stack consisting of a signaling protocol (call setup - tear-down), a call quality feedback protocol, and the RTP-Engine including the interface to be designed. However, the a priori chosen prototyping platform induces several embedded system specific constraints, like limited computing speed and limited FPGA resources, directly affecting the VOIP-stack design. Moreover, special care in designing the hardware/software interface of the IP-Core must be taken to achieve certain constraints imposed by the VOIP-stack.
To achieve the given objective, the general design of a VOIP-stack has been evaluated to derive appropriate constraints when designing a Base-System and the RTP-Engine's hardware/software interface. The gained constraints are used to evaluate and decide upon interface design questions as well as choosing suitable components for the system. Furthermore, several implementation options considering software-stacks, interface design, and user/kernel -space realization of certain software related interface parts are discussed.
Finally, an operational prototype able to perform a VOIP-call directly between two prototyping platforms or a Personal Computer (PC) has been realized. It can be seen that a properly designed and implemented interface and VOIP-stack for an IP-Core transposing the necessary tasks to exchange voice packets between two communication partners results in less pressure on the general purpose processor on meeting the required real-time constraints. Therefore, it would be possible to extend the prototype in such a way to support teleconference calls as well, which can be the prospect of future work.

Acknowledgements

First, I would like to thank Univ. Prof. Dr. habil. Christoph Grimm and Dipl. Inf. Dr. Jan Haase for their valuable feedback and remarks. Additionally, I would like to thank Dipl. Ing. (FH) Peter Brunmayr for his short response times and thorough supervision of my thesis. Last, but most important I would like to thank my parents and Sigrid for their support and understanding.

Table of Contents

Abbreviations

AC97	Audio Codec 97
ARP	Address Resolution Protocol
BRAM	Block Random Access Memory
CPU	Central Processing Unit
CSRC	Contributing Source
EDK	Embedded Development Kit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPNIF	General Purpose Network Interface
IOCTL	Input/Output Control
IP-Core	Intellectual Property Core
IP	Internet Protocol
IPIF	Intellectual Property Interface
JTAG	Joint Test Action Group
LLI	Local Link Interface
LUT	Look-Up-Table
MAC	Media Access Control
MMU	Memory Management Unit
PCMA	Pulse Code Modulation A-Law
PCMU	Pulse Code Modulation μ-Law
PLB	Processor Local Bus
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request For Comment
RTCP	Real-time Control Protocol
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
SDES	Source Description
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SOC	System on Chip
SSRC	Synchronisation Source
TCP	Transmission Control Protocol
UDP	Universal Datagram Protocol
VOIP	Voice over Internet Protocol

1 Introduction

In recent years, Voice over Internet Protocol (VOIP) [FCP09] communication has become an interesting alternative to the Public Switched Telephone Network (PSTN) for casual and business phone calls in industrialised countries. This is mainly the fact, because of a wider availability of internet connections offering the needed bandwidth. A convenient example is the wide acceptance of the VOIP application Skype [73]. Moreover, there exist a wide range of solutions offering ways to pool telephone and data networks in companies and communities.

However, most of the available VOIP capable end-devices are considered for general purpose use, thus neglecting special needs of certain operational environments, like air traffic management, public transport, or defense. In order to facilitate VOIP services in especially isochrone-sensitive environments, special care must be taken upon regarding the guarantee of isochronous behavior in the network domain, various ways to achieve this requirement exist. However, considering the violation of isochronous behavior at the end-devices no wide spread solutions are available.

Therefore, a hardware module performing all isochrone sensitive computations in a deterministic way has been developed. The module is suited to run on an Field Programmable Gate Array (FPGA) placed on a prototyping board. It performs the following tasks:

- Capturing and encoding of voice samples,
- Packetizing and scheduling of media packets,
- Transmission and reception of media packets,
- Filling and mangling the jitter buffer,
- Decoding and playback of received voice packets

These tasks are necessary to let an already set-up VOIP-session exchange packets which include encoded voice samples. This will lead to a reduced end-to-end delay on the terminal device side. Moreover, the provided module reduces possible non-determinism delays to an insignificant value. Non-deterministic delays directly affect isochronous requirements in a negative way.

Non-determinism may be introduced by general purpose operating systems like Windows or Linux because it is possible that, although a media-transport packet is ready to send, the operating system might postpone its transmission by issuing a context- or mode-switch. The same is true for corresponding media packet reception.

However, what is still missing is an interface connecting the IP-Core with a Base-System providing a general purpose CPU, memory, and several peripherals. This leads to a VOIP-stack implementation in an FPGA, which is a well mastered task by industry and academia in general. To comply with the requirement to reduce the end-to-end delay on the terminal device further, care must be taken when designing the interface between the hardware-module and a Base-System. This is especially true regarding the interface between media-transport and call-quality feedback protocols due to their strong entanglement.

The aim of this thesis is to perform a hardware/software partitioning of such an interface as well as providing the following components to let VOIP calls take place between two communication partners using this dedicated hardware-module:

- A Base-System consisting of a Central Processing Unit (CPU), several peripherals and an appropriate operating-system, or runtime-environment.
- A hardware/software interface providing access to the hardware-module performing the VOIP related computations.
- A software stack completing the necessary VOIP-stack functions, like signaling and call-quality feedback protocol not provided by the hardware-module must be implemented to verify the interface approach.

To achieve the given objective, the general design of a VOIP-stack has been evaluated to derive appropriate constraints when designing a Base-System and the RTP-Engine's hardware/software interface. The gained constraints are used to evaluate and decide upon interface design questions as well as choosing suitable components for the system. Furthermore, several implementation options considering software-stacks, interface design, and user/kernel -space realization of certain software related interface parts are discussed.

Finally, an operational prototype able to perform a VOIP-call directly between two prototyping platforms or a Personal Computer (PC) has been realized. It can be seen that a properly designed and implemented interface and VOIP-stack for an IP-Core transposing the necessary tasks to exchange voice packets between two communication partners results in less pressure on the general purpose processor on meeting the required real-time constraints. Therefore, it would be possible to extend the prototype in such a way to support teleconference calls as well, which can be the prospect of future work.

The remainder of this theses is structured as follows: In Chapter 2, a general overview on VOIP communication examining several ways to setup, close, and perform a VOIP call is given. Furthermore, three different signaling protocols are examined from a functional and structural point of view. In addition, an overview on media transport and their underlying Open System Initiative (OSI) transport protocols is presented. Finally, an introduction on several Quality of Service (QoS) enforcing aspects in general purpose networks is given.

Subsequently, Chapter 3 introduces the design of a VOIP-stack on the target platform basing on the information gathered by reviewing similar approaches. Therefore, several hardware and software components and implementation options are evaluated. Furthermore, the IP-Core's interface design is examined in detail. Eventually, the hardware/software partitioning of the interface as well as Base-System to be implemented is presented.

Chapter 4 illustrates the implementation including an evaluation of several implementation options of diverse components of the design decisions made in the previous Chapter. This Chapter

focusses on the implementation details and decisions made at the implementation level which have not been considered in the design phase presented in Chapter 3. Moreover, the implementation of the supplementary VOIP components like signaling and call-quality feedback protocol stacks are examined.

Chapter 5 presents synthesis results and a VOIP-session trace showing the validity of the implemented solutions. Moreover, requirements for further extensions are derived from these results. Finally, Chapter 6 summarizes the insights gained by realizing the designed interface and the Base-System. Furthermore, possibilities for further extensions gained from the requirements in Chapter 5 are discussed.

2 VOIP - An Overview

The first efforts on transmitting voice over packet switched networks were taken in 1974, at the time of the Advanced Research Projects Agency Network (ARPANET). At that time, Danny Cohen and his colleagues developed the Network Voice Protocol (NVP) [82] as a counterpart to Transmission Control Protocol (TCP) [84], in order transmit voice in real-time over the ARPANET. Cohen saw the need for a dedicated voice protocol because he realized, amongst others, that the reliability mechanisms built into the current TCP implementation were not suitable to transport voice in real-time. [Gra05].

In the mid 1990's, voice over packet switched networks had a short revival, but due to its poor quality, its limitation to Internet Protocol (IP) network communication, a not very widely spread availability of IP networks, and the still limited bandwidth, Voice over Internet Protocol (VOIP) could not succeed commercially. This was also emphasized by the fact that circuit switched voice networks provided high quality and high reliability voice calls, so that there was no urge to switch to VOIP. [VSMH02].

When IP networks became more common, due to the struggle in expanding the infrastructure for the growing Internet in the 1990s, the advantages of VOIP over Public Switched Telephone Network (PSTN) became clearer [VSMH02]. Where PSTN had to provide 32 to 64 kbit/s for the whole communication session, a VOIP based call is able to succeed at a guaranteed bit-rate of 14 kbit/s plus administrative overhead, for example [FCP09]. Another VOIP supporting argument is that a VOIP call's bandwidth is only needed if actual voice packets are transmitted, therefore the capabilities of a packet switched network can be exploited. However, it should have lasted until the early 2000's, when VOIP was considered to replace the PSTN, that VOIP communication gained a broader economic acceptance, according to Bur Goode. [FCP09].

In order to successfully perform a VOIP call between two participants, A and B, several operational supplements are necessary. First of all the communication partners have to establish a link between each other. The two parties have to agree on a voice-codec, and further informations to establish a peer-to-peer voice communication. Furthermore, B may want to send A Dual-tone Multi-frequency (DTMF) signals, and maybe if B called A, A's phone may display the Identifier (ID) of B on A's phone. This is done by so called signaling protocols addressed in Section 2.1.

After that, A and B may exchange voice packets over so called transport protocols, which are introduced in Section 2.2. In order to maintain a certain call quality, if B's bandwidth is smaller than A one's, it might be possible that A's voice packets may congest B's line. However, in order

to avoid such issues Quality of Service (QoS) techniques are deployed. These principles can be enforced by the network infrastructure or by special control protocols. Section 2.3.1 provides an overview of the most common QoS measurements and their usage in VOIP communication.

In order to get a clearer point of view on the basics of Voice Over IP communication, a schematic VOIP session, consisting of the call establishment, the call accomplishment and the VOIP-session tear-down is shown in Figure 2.1. Therefore, the purpose of this Chapter is to introduce the fundamentals of VOIP communication.

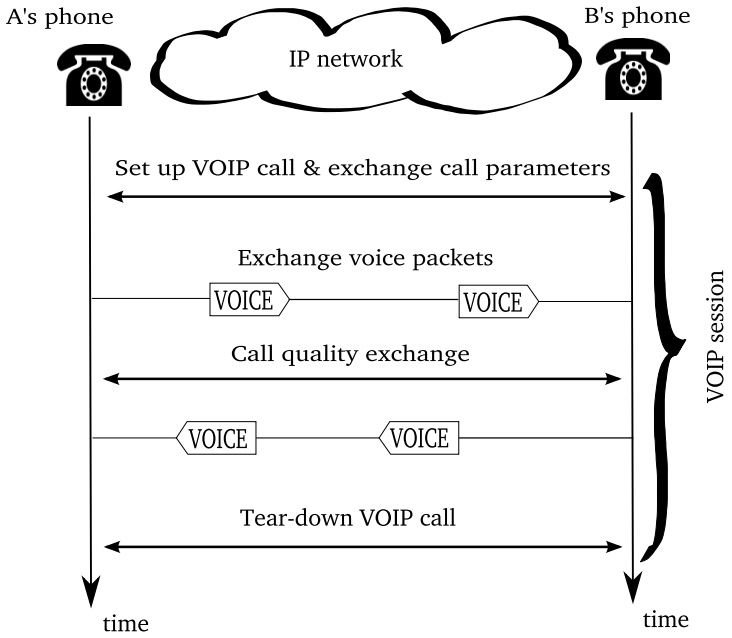


Figure 2.1: VOIP session scheme

2.1 Signaling

As mentioned in the introduction of this Chapter, signaling protocols are responsible for exchanging the necessary information to set up a communication channel between two hosts. There are two types of signaling protocols according to Goode [FCP09]. On the one hand, there are peer-to-peer protocols, like the Session Initiation Protocol (SIP) [93], H.323 [LM00], Inter Asterisk exchange (IAX)[1][98], and Jingle[SA07]. On the other hand, there are master-slave protocols like Media Gateway Control Protocol (MGCP) and Megaco/H.248. However, master-slave protocols are not very flexible and scalable, due to their nature in processing all calls over a central node [FCP09]. This section gives an overview on peer-to-peer protocols only.

2.1.1 H.323

The first protocol introduced is H.323, which has been released in its first version by the International Telecommunication Union Telecommunications (ITU-T) Study Group 16 in 1996 [LM00]. H.323 does not specify the physical network, network interface, or the used transport protocol.

[1]Signaling Protocol for Asterisk based VOIP to PSTN gateways. [98]

However, it defines functions on the application layer by recommending a set of network protocols. Yet, H.323 is not a protocol itself. It subsumes a set of protocols which accomplish the needed tasks, where the channel signaling protocol H.225.0 is described in [VB01]. In order to understand the interaction of the used protocols an overview of the structure of a H.323 is given. Its network may consist of the following components, according to [VB01]:

Terminal A terminal is used as a communication end-point and may be a H.323 capable phone or a computer with a corresponding softphone.

Gatekeeper A node in a H.323 network which houses a database where address mappings between alias and transport addresses are kept. Furthermore, it performs address translations.

Gateway This device provides an interconnection between a H.323 network and a PSTN network, for example. It is responsible for the whole communication translation.

Multi-point Control Unit The MCU accommodates services for conference calls. A conference call is a communication session where more than two communication partners are interacting with each other simultaneously.

A call setup based on H.323 can be seen in Figure 2.2.

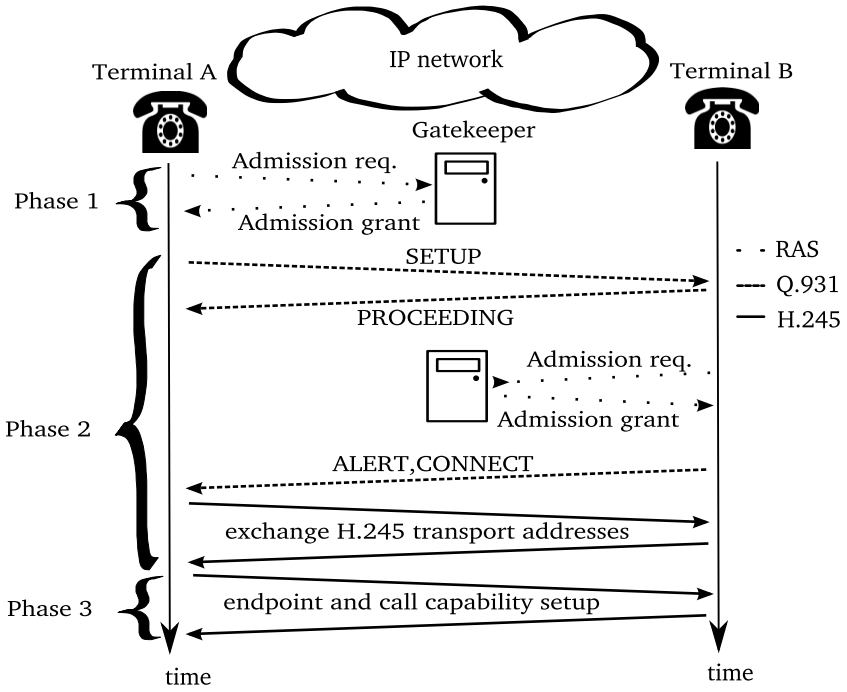


Figure 2.2: H.323 Call Setup

Basically, there are two ways to setup a call between two parties in H.323. On the one hand, the call setup may take place over a Gatekeeper. On the other hand, the caller might contact the callee directly. If the Gatekeeper is used, the call setup consists of three phases. First, the Gatekeeper is queried in order to obtain the call admission and to retrieve the Q.931 address of the remote peer. Q.931 is a protocol recommended to perform the call setup and call teardown hand-shake in a H.323 VOIP session. This is done with the Remote Access Service (RAS) protocol, which is a subset of the H.225.0 signaling specification for H.323. If the callee's address

is not known by the Gatekeeper, it may query another Gatekeeper in order to obtain the address. This is known as *Call Admission*. If it is not necessary to get a call admission or the remote peer's Q.931 address is already known, the connection for the second phase may be established directly.

In the second phase, *Call Setup*, the call setup between the two parties takes place. This call setup is performed by the Q.931 protocol, which is also used by Integrated Services Digital Network (ISDN) [92]. Since ISDN is a circuit switched service which also provides access to packet switched networks, Q.931 has the abilities to perform its signaling services in both network types. In the case of packet switched networks, Q.931 exchanges a number of messages in the following way: After the call is admitted the terminal sends a SETUP message to the destination, which responds with a CALL PROCEEDING message. This message includes informations about the endpoints capabilities. If there are any Gatekeepers involved in the communication these messages are relayed by the Gatekeeper. Subsequently, if the receiver is not already admitted at its Gatekeeper, it does so. If the admission has already been approved, the callee sends an ALERT message to the caller and subsequently, a CONNECT message which contains the needed call parameters in order to setup a media channel. Now the H.245 transport addresses are exchanged. This ends the second phase.

In the third and last phase, *Endpoint and Call Capability Setup*, the protocol H.245 is used to mediate the endpoints for the transport protocol and to set up a logical channel between the two parties. Moreover, Q.931 and H.245 are also subsets of the H.225.0 signaling specification for H.323. [FCP09, LM00]. A call tear-down works the other way round. The transport protocol relevant data exchanged by the H.323 signaling phase are [Gra04]:

- Used Voice Codecs
- Determine slave-master relation ship between the two communication partners
- Determine Round-Trip-Delay in order to sense whether a peer is alive or not. This information can be retrieved periodically.
- Logical Channel Signaling is used to exchange the remote peers transport protocol parameters like IP-addresses and port numbers.

2.1.2 Jingle

The second protocol described in this section is Jingle, an extension to the Extensible Messaging and Presence Protocol (XMPP) which is used to signal and control data streams like voice over IP networks [SA07]. XMPP is a messaging protocol used to implement the instant messaging service Jabber, an open alternative to close source instant messaging services like ICQ (Instant Messaging protocol from ICQ LLC) and IM (Instant Messaging protocol from Microsoft). Although, XMPP is an Extensible Markup Language (XML) based protocol it does not communicate with a peer by exchanging whole XML-documents or utilizing the Extensible Hypertext Markup Language (XHTML). In the case of XMPP, the two communication partners exchange messages, by transmitting so called stanzas, XML tag snippets, which can be seen as the packets used in the communication on top of the transport layer protocol [SA07]. However, XML tags are not well suited to transfer large amounts of binary data. To overcome this limitation, Jingle has been introduced in 2004 [SA05]. Jingle defines a signalisation protocol and uses the Real-time Transport Protocol (RTP) as a transport protocol after the communication parameters are exchanged.

Since Jingle bases on XMPP it also uses its infrastructure to set up a call between two peer. Therefore, the basic XMPP stanzas, presented in [SA05] are explained here:

Message A Message is used to propagate information from a source to one or several sinks. These stanzas may not be acknowledged. They are commonly used in Instant Messages, Alerts, and Notifications.

Presence This stanza is used to notify several destinations about own state updates.

IQ IQ Means Info/Query and is used like the GET/POST methods in Hypertext Transfer Protocol (HTTP) for simple information request and post messages.

Due to the extensible nature of XML, it is easily possible to develop further protocols on the top of XMP, like Jingle [SA07]. A Jingle call signalisation consists of the following steps: A session example including call setup and session tear-down can be seen in Figure 2.3.

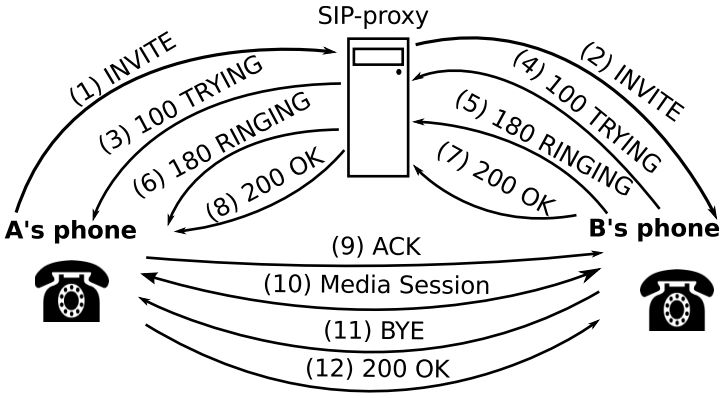


Figure 2.3: Jingle Session

First, a so called *Initiator* sends an IQ-stanza with an encapsulated "session-initiate" Jingle-Action to a *Responder*. The "session-initiate" message consists of two parts:

- An application type which describes the media type of the Jingle Session initially.
- A transport method which describes the transport layer protocol to be used for media exchange.

If the *Responder* accepts, it answers with another IQ-stanza with an "session-accept" (ACK) message which contains the used voice codec, bit-rates, and port-numbers it agreed on with the *Initiator*. When the *Initiator* agrees on the *Responder's* offer it sends an acknowledge message. Finally, the media session is set up. Moreover, Jingle is able to change an ongoing media sessions with Jingle-Actions like "content-add", "content-remove", "content-modify", and "transport-replace" [SA07]. When a Jingle session is to be teared down, one of the communication partners sends a "session-terminate" message and the other clients acknowledge.

2.1.3 Session Initiation Protocol

The last signaling protocol discussed in detail is the Session Initiation Protocol (SIP). It has been proposed by Schulzrinne et. al. in [95]. However, RFC-2543 has been obsoleted by RFC-3261 in 2002 and has been updated several times since then [93]. In contrast to H.323, SIP is an application layer protocol. SIPs primary intension is to provide signaling methods which apply to the communication demands of IP networks, like dynamically changing IP addresses and locality changing communication partners, for example. These are challenges for other protocols like H.323. [SR00]. To support these functional demands of IP networks, a SIP network consists of several components which can be split into two groups, namely User Agents (UAs)s, and Servers [93]. All components are listed in the following.

User Agent Client The UAC is a part of a SIP User Agent which sends requests to each communication partner and awaits its answer.

User Agent Server A UAS is a part of a SIP User Agent which waits for SIP requests and replies to the corresponding UAC. Both, UAC and UAS are present in any SIP capable phone solution.

Registrar Server A Registrar-Server is an optional component in a SIP network. However, it is needed to utilize the whole capabilities provided by SIP. It associates the IP-address of an UA with a SIP-URI[2](Uniform Resource Identifier), therefore it administers its name-space domain, on the one hand. On the other hand, a registrar can be used for billing purposes in conjunction with a SIP-Proxy Server, and selective call-setup policy enforcement, like letting only a subset of employes place calls outside a companies phone-network, for example.

Redirect Server A Redirect Server's task is to inform calling parties about their callee's location change and reroute their call to the callee's new address.

Proxy Server Two different kinds of SIP-Proxy Servers exist, stateful and stateless proxies. A stateful proxy kepdf track on every SIP connection running over it. Where a stateless proxy only redirects requests and their answers. A SIP-Proxy is a main node however, not necessarily needed in every SIP network. It provides services to inform communication partners about the availability of each other and is responsible for routing call-setup requests throughout the borders of several SIP-domains. Moreover, a SIP-Proxy is able to retrieve a callee's SIP-URI by processing a query on another proxy with the help of a name resolution backend as mentioned by Schulzrinne & Rosenberg [SR00].

Usually, all three types of SIP-Servers are deployed on one physical machine in order to provide the best service quality [93, SR00]. Considering the call parameter exchange during the call setup-phase of a VOIP-session, SIP utilizes the Session Description Protocol (SDP). SDP has been proposed in RFC-2327 in 1998 [96]. Its latest revision, RFC-4566 has been enforced in 2006 [97]. SDP is not a full featured protocol like Q.931 from the H.323 standard. However, it can be compared with Jingle's XML-stanza exchange mechanism since it lacks addressing schemes but defines a set of fields which can be used to exchange call parameters like peer-addresses port-numbers and voice-codecs. Its intention is to provide a general transportation independent way to represent data like call parameters and session announcements [97]. However, although

[2]A SIP-URI is similar to an e-mail address. It has the following form: Username@Provider.TLD E.g.: sigma@sip-provider.at

comparable, in contrast to Jingle, SDP does not define any actions, it serves as a message exchange format only. The appropriate action taken on the reception of a SDP packet are defined by the protocol utilizing SDP, which is SIP in this case. Yet, SDP is not bound to SIP. It can also be used with other protocols like Session Announcement Protocol (SAP) and Real-time Streaming Protocol (RTSP) [94, 85].

A SIP session is characterized by the exchange of messages, where the responses to queries are categorized by a set of numerical code intervals known as Response Types [93]. Furthermore, it is classified into transactions. Each transaction begins with a message and ends with a *2xx* response in order to end the transaction. The numerical codes of SIP messages are grouped like in HTTP. Thus, a code beginning with two, like in *2xx* denotes a success message. However, an *x* can be a number from zero to nine specifying the meaning of the message more detally. A message starting with six, like *6xx*, indicates global call failures [93]. In the case of an initial INVITE, the ACK message is part of the transaction if the callee's response was not of Response Type *2xx*. Otherwise it is not part of the transaction [95]. According to [SR00], the most important message types are:

INVITE The Invite message indicates an initial session announcement. It also provides an SDP payload with the caller's call-parameters.

100 TRYING TRYING is the response of a callee on an initial invite. It serves as a INVITE reception confirmation.

180 RINGING RINGING indicates to the caller that the callees SIP-UAS informs the user interface of an incoming call.

200 OK This SIP message is known as a positive response to a request. If the OK message succeeds on an INVITE message, it contains an SDP payload describing the callee's call-parameters, respectively the possibly changed call parameters if it follows on another SIP message type. See [72] for a complete list of SIP parameters.

ACK The ACK message is used to notify a communication partner on the completion of an initial INVITE transaction. This separately sent message is the first message sent peer-to-peer within a SIP-session being processed over SIP-proxies only so far [93].

BYE This message is sent upon a communication partner is willing to end an ongoing VOIP session.

Other possible requests are:

REDIRECT This request indicates that a call is redirected to another endpoint.

REGISTER This SIP message is used to register a user at a Registrar-Server.

OPTION The OPTION request is sent to gain information on the capabilities of a server.

CANCEL This message is used to cancel an ongoing request.

A SIP based VOIP call setup session consists of at least a caller and a callee represented by two UAs, forming the endpoints of the session. The remaining SIP network nodes, like Registrar-Server, Redirect-Server, Proxy-Server, are optional. Furthermore, an ongoing session is known as

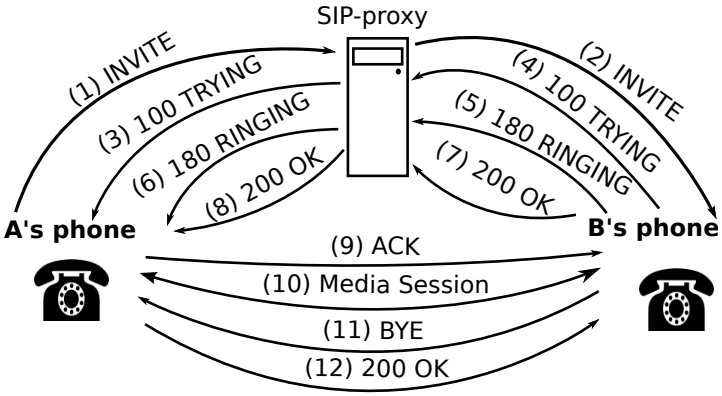


Figure 2.4: SIP Session

a call-leg. Moreover, a SIP session is usually characterized by the SIP trapezoid. This shows that a call partner's IP address might not be known in advance, but its SIP-URI is. A VOIP session using SIP as a signaling protocol can be seen in Figure 2.4.

Therefore, a SIP client may contact a Proxy-Server[3] by sending an initial INVITE containing a SDP packet with the caller's capabilities first. If the Proxy-Server knows the callee's IP-address it relays the caller's INVITE directly to the callee. Otherwise it queries another known Proxy-Server or informs the caller about the fact that his call is not possible. The callee responds with a TRYING response to the caller by using the same route back to the caller over the SIP-Proxy. Subsequently, a RINGING response is sent. Finally, the callee responds with an OK response containing the complete call parameters, also utilizing the SIP trapezoid. The caller, however, acknowledges this response with an ACK message. This ACK message is the first message sent peer-to-peer between caller and callee. Finally, a media transport session can be set up in order to exchange communication data. When one call-party wants to end the call, it sends a BYE message which is acknowledged by an OK response.

2.1.4 Signaling Protocol Summary

In the previous Sections, an overview on a selection of signaling protocols was given. Although the main aims of the introduced protocols are the same, they differ in their architectural design. H.323 is the oldest of the presented protocols which can be seen, besides its publishing date, by the fact that it contains the Gateway component, which is used to directly interfere with PSTN networks. At this time internet connections with appropriately high bandwidth were not that common, thus H.323 networks were mainly deployed in companies or public facilities. Moreover, H.323 does not specify a protocol on its own, it rather gives recommendations on which already available and wide spread protocols to use when building H.323 networks.

In contrast to H.323, SIP is an entirely new developed signaling protocol, which has been initially designed to provide services in parallel to the conventional PSTN. Furthermore, SIP does not specify, or recommend any given transport protocol, therefore not introducing any kind of limitation when considering media transport [93]. Jingle, however, is different to SIP and H.323

[3]Due to the fact that all SIP-Servers are commonly implemented on one computer, the term Proxy-Server may be used as replacement for any SIP-Server node, regardless of being a Registrar-Server, Redirect-Server or Proxy-Server [SR00].

in its origin, as it is proposed as an extension to a text chat protocol popular in the late 1990's [SA07].

Moreover, the three mentioned protocols differ in the way they perform their tasks. On the one hand, H.323 uses Q.931 as a call establishment protocol which utilizes TCP as a transport layer protocol to exchange call parameters and provides its own addressing scheme. On the other hand, SIP and Jingle use the addressing methods provided by their transport protocols. Furthermore, SIP uses SDP, which is linked to SIP by SIP-messages and Jingle uses XML-stanzas to implement an according call establishment handshake. Also closed source solutions exist, where Skype is one of the most wide spread solutions. However, since it is not an open implementation only speculations and analyses upon its utilized signaling measurements exist [BMMR09].

2.2 Transport

After a call has been established by exchanging an agreed voice codec and corresponding communication endpoint parameters by the means of signaling protocols, an appropriate transport protocol is needed. Although, the term transport protocol in section 2.1 is used to describe protocols which actually encapsulate the encoded voice samples, it is used in the context of the Open System Initiative (OSI). So when we speak about transport protocols we mean protocols on the transport layer like TCP and Universal Datagram Protocol (UDP) for example. Furthermore, when one refers to voice-sample transporting protocols they are addressed as media-transport protocols in this Section. Since voice over IP sessions have to deal with certain QoS issues, which are addressed in section 2.3, in order to maintain a convenient call process, the chosen transport protocol is important as well, since its behavior in certain environmental conditions has major influence on the quality of a call. A corresponding survey has been performed by Sreekanth et. al. in [AGSS09]. In their survey, the authors compare TCP, UDP and Stream Control Transmission Protocol (SCTP) in order to give recommendations on which one of these is well suited as a transport protocol for speech. Before the studies results are shown, a short overview on the analyzed protocols is given.

2.2.1 Transport Protocols

TCP is used, if the connection demands guaranteed delivery. It uses a sliding window protocol which takes care on timeouts and retransmission requests due to packet loss. Moreover, TCP establishes a full duplex channel between two communication partners. Each channel is characterized by IP-addresses and port-numbers. The protocol transfers the payload as byte-stream which is split into segments. The number of bytes transported in a packet is determined by the window size, which is controlled by the sliding window protocol as mentioned above. Furthermore, each packet has to be acknowledged with a special packet from the receiver. [84].

UDP provides a non-guaranteed packet delivery. It is a lightweight protocol since it only provides header information for port numbers to identify the endpoints in the communication and a checksum field. However, since UDP lacks features like retransmission and reliable delivery a user application would have to handle these. Nevertheless, in contradiction to TCP, UDP is able to perform broadcast and multi-cast. [99].

Originally, **SCTP** has been designed to transport PSTN signaling messages over IP networks. It can be described as a mixture of UDP and TCP since it is a connection-less protocol which

provides reliability features. It also provides accumulation of several user messages in one packet, data fragmentation and congestion avoidance behavior as well as prevention mechanisms to flooding and masquerade attacks [100]. In a masquerade attack an attacker pretends to be the actual destination of an ongoing communication in order to gain sensitive data according to [GR05].

Sreekanth et al. [AGSS09] set up a test network consisting of several subnets with different bandwidth and traffic. Beside VOIP communication, TCP connections and HTTP connections were run on the network. Furthermore, the authors defined the scenarios high latency high bandwidth), high latency low bandwidth) and low Latency high bandwidth in order to measure their metrics packet loss[4], delay[5] and delay variation[6]. The measured values were used to build a statistic in order to argue which of the mentioned transport protocol is the most suitable for voice transmission. The conclusion of the authors was that none of the protocols are best suited to perform this task. However, the authors recommend UDP as a transport protocol for voice since it performs best by the means of jitter introduction due to line congestion. SCTP's performance is comparable to UDP's, but could be better with slight protocol changes according to Sreekanth et. al. . Finally, TCP is not very well suited for voice communication due to its connection oriented operation which introduces too high jitter due to retransmissions [AGSS09].

2.2.2 Media-Transport Protocols

The only media-transport protocol and its extensions used throughout several VOIP communication specifications is **RTP** [Gra04, SA07, SR00, Bad07]. Therefore, it is the only media-transport protocol introduced. The protocol fulfills a number of properties needed in order to transmit real-time data over IP-networks (see [Bad07, 87]).

- Due to the usage of sequence numbers the ordering of received packets is guaranteed. Even if they arrive out of order at the destination, it is possible to reorder them due to their sequence number.
- Moreover, RTP uses times-tamps to guarantee a certain level of isochronous behavior. Isochronous means that a constant amount of time passes between two suceeding samples [Bad07, p.166]. Therefore, the arriving voice packets can be played back with the same relative time difference as they arrived.
- Because RTP uses profiles to determine its payload, it is capable of transporting several media data types which have been encoded with different codecs.
- When using RTP, one can employ translators and mixers. A translator can be used to convert media data from one type to another, where a mixer can be used to merge RTP streams from several sources into one stream.

RTP resides on the session layer of the Open System Interconnection (OSI) ISO standard [ISO] reference layer model. It is a peer-to-peer media-transport protocol especially designed to carry real-time data like audio, video or simulation data. It is capable of broadcast, multi-cast and unicast communications in IP networks. However, RTP does not support QoS mechanisms. These

[4]The amount of lost packets during a communication session
[5]The fixed transmission latency, introduced by packet travel time.
[6]The amount of time added to to delay because of routing strategies, traffic priorisation and line congestion for example.

tasks are usually handled by the Real-time Control Protocol (RTCP) protocol, an augmentation to the RTP protocol, which is addressed in section 2.3.

The RTP protocol uses UDP as its underlying transport protocol. However, it is not bound to it [87]. In order to gain a better understanding on the decisions made in the succeeding Chapters of this theses, RTP is described in greater detail. First the header is examined. Afterwards, the packet body is described. Furthermore, the overall functionality of the protocol and its capabilities are introduced. The header consists of a fixed and a variable part. The fixed part must be present in any RTP packet, whereas the presence of the variable part depends on the scenario the protocol is used in.

The single header fields have the following meaning [87]:

Version This field denotes the protocols version. It is always set to the value 2.

Padding If set, the packet contains one ore more padding bytes at the and of the payload. This may be neccessary if encryption algorithms are used which require a fixed block size.

eXtension If set, the static part of the header is followed by exactly one header extension.

Counting Contributing Source (CSRC) CC, denotes the amount of CSRCs sent in the variable part of the header.

Marker If set, the Marker bit is interpreted by the profile contained in the packet's body.

Payload Type This field indicates the payload type transported by the packet's body. This identifier specifies the used profile in the packet body. The structure of a payloads profile is defined in [86].

Sequence Number A monotonically incremented counter which represents the amount of actually transmitted packets.

Timestamp The value of a monotonically and linearly incremented clock which represents the instant when the first payload octet has been sampled. It is used for jitter calculation, which is addressed in Section 2.3.

Synchronization Source The SSRC is used as a media session unique identifier.

Contributing Source The CSRC denotes the media sources participating in the stream. It is used for Quality of Service message delivering.

The RTP payload is entirely described by profiles. The used profile is transported in the header field payload type. Furthermore, the marker bit can be used if an additional interpretation of some profile informations may be necessary [87]. There are several types of payload profiles available, for example Pulse Code Modulation based on Alaw (PCMA) [91] or Pulse Code Modulation based on μlaw (PCMU) [91] coded voice. For a detailed list and profile description see RFC-3551 [86].

As mentioned above, RTP features the deployment of mixers and translators. A **translator** can be used to convert media data from one format to another. This is done without interfering into the ongoing session, except for the introduction of a short delay, due to the conversion task. Therefore, the Synchronisation Source (SSRC) of an ongoing session is not changed [87]. This procedure is shown in Figure 2.5.

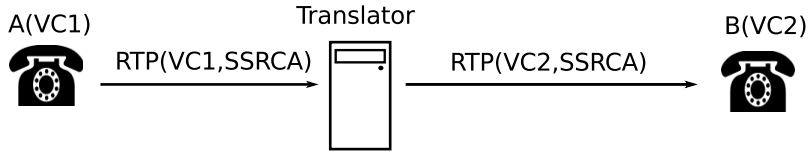


Figure 2.5: RTP Translator [Bad07, p.156]

In Figure 2.5, a source A using voice codec $VC1$ communicates with a destination B using voice codec $VC2$. Both peers utilize a translator in order to convert their codec used into the one applied by its callee. The translator converts their codecs without changing their SSRC. Therefore, it seems like A and B are communicating directly with each other, thus making the translator transparent by the means of the RTP session.

A mixer is a device which captures media data from different sources with different SSRCs and combines these streams to a new stream with a new SSRC [87]. This procedure can be seen in Figure 2.6.

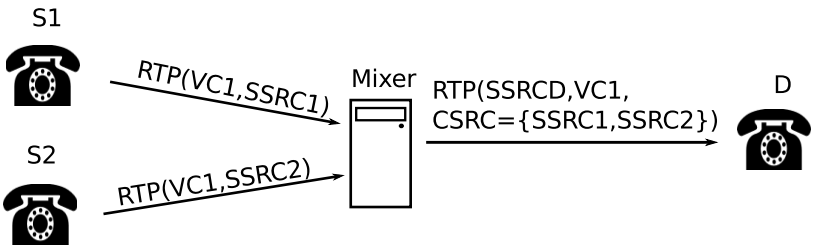


Figure 2.6: RTP Mixer [Bad07, p.157]

The sources $S1$ and $S2$ use voice codec $VC1$ to communicate with a destination D as shown in Figure 2.6. However, since a mixer M is deployed, the RTP-packets of $S1$ and $S2$ are merged into a new RTP-stream set up by the mixer to communicate with the destination D. Where the RTP-streams between the sources $S1$, $S2$ and the mixer M can be identified with their corresponding SSRC directly, the destination D uses the CSRCs sent by the mixer to identify the sources. Moreover, the mixer deploys a new SSRC for the communication with D. A mixer may be useful in an environment where an audio stream is created from different sources. Furthermore, mixers and translators may be combined in their application.

The procedure for setting up an RTP based peer-to-peer communication works as follows. The network addresses, port numbers, and voice codecs needed to set up an RTP based communication are exchanged by the signaling protocols described in Section 2.1. However, the parameters for the RTCP protocol may not be transmitted in that way. In this case, RFC 3550 [87] specifies that the port number used for RTP communication is even and RTCPs port-number is RTPs port-number +1, therefore odd.

2.2.3 Transport Summary

In the previous section the results of a survey on the usability of certain transport protocols for real-time media data were presented [AGSS09]. According to this study the UDP protocol is one of the best suited for this task. Considering media-transport protocols the RTP/RTCP protocol suite seems to be best suited for this purpose, since it is the only open wide-spread media-transport protocol [Gra04, SA07, SR00, APSL05].

2.3 Quality of Service

The main difference between a packet and a circuit switched network is that in a circuit switched network, once a resource is allocated, it is guaranteed to be available to a user. Since all data transported within a circuit switched network use the same path through the network, the end-to-end delay between two communicating parties is constant. Furthermore, the communication channel is able to use the whole amount of provided bandwidth throughout its lifetime. A major drawback of circuit switched networks is that the allocated channel resources must also be reserved if no communication is taking place within an existing connection. (E.g., if no one speaks within a phone call.) Moreover, if a node in the channel's path fails, the whole channel fails, even if another path from source to sink would be available [Gra10, p.682]. Considering packet switched networks, however, these issues do not arise. Packet switched networks are able to react on changes of certain operational conditions like the loss of a node in a path by rerouting the packets of an ongoing communication through another node to their intended destination. The allocated bandwidth can be provided dynamically according to the needs of the application using it with respect to the overall amount of available bandwidth in a networks segment.

A packet's route and a communication link's bandwidth are not determined in a packet switched network. Therefore, a packets end-to-end delay is not defined a priori, because packets may take arbitrarily routes to their destination. Thus, special needs for certain applications which have time constraints in their functional requirements arise [OS00]. In order to cope with this issues, so called QoS mechanisms have been proposed and implemented in order to provide services for time or bandwidth sensitive applications to fulfill their requirements. Moreover, applications of packet switched networks can be classified in the following ways: Most of them are loss-sensitive. Some more real-time aware services are also delay- and sometimes also bandwidth-sensitive [OS00]. Although most data applications can handle a certain amount of packet loss generally, it is known that a loss-rate of larger than 5% leads to poor performance [OS00]. However, media applications, such as VOIP communication, are often delay-sensitive and require a fixed amount of minimum bandwidth. This is in contrast to the quality of service mechanisms provided throughout the Internet and similar IP-networks, as mentioned above [CWX+03, OS00]. Therefore, a set of techniques introduced in 2.3.1 have been developed to cope with this issue.

2.3.1 QoS Techniques

In order to achieve a better service quality for certain applications a numerous quantity of methods exist. Some of them are presented here [OS00, CWX+03, Bad07]:

Prioritization of Media Access Control (MAC) Frames This method works by assigning priorities to certain Ethernet frames demanding a higher priority on their network usage attempts. However, in order to use this technique, all switches located in the network must support this technique.

Prioritization of IP Packets This QoS mechanism is applied on the network layer and is also suitable for wide area networks. The packet prioritization is done on the service type. Therefore, this technique is also called Differentiated Service or *DiffServ*. However, in order to enforce this technique successfully, all networks used by a VOIP session must support this QoS method.

Queue Management Each relaying unit in a packet switched network utilizes queues to buffer outgoing packets before the line on their destination network segment is ready to receive another packet. These queues are implemented as First In First Out (FIFO)s in the simplest case. However, since the packets need to wait a short amount of time inside these queues the method of Queue Management uses this time to reorder the packets in the output buffers according to defined priorities, thus providing a shorter waiting time for a packet participating in a VOIP session, for example.

Resource Reservation Protocol (RSVP) RSVP is a protocol to mimic resource reservation in a circuit switched network way. At the beginning of a media session it is the RSVP's duty to reserve a certain amount of bandwidth on the path to the communication partner. However, since all network segments have to support RSVP to let it perform its task properly, it can only be used in limited environments today because of its short acceptance.

Multi-Protocol Label Switching (MPLS) In IP networks several packets of an IP connection may take several routes from a source to a destination due to several environmental conditions like queue congestion or a link failure. However, considering VOIP communication, this ability of IP networks may lead to poor call quality because the inter-arrival time of voice packets should be as small as possible. To cope with this application specific issue, MPLS sets up a virtual channel so that a sessions packets can take the same route between two communication partners and therefore shortens the difference of adjacent packets inter arrival times.

However, since the mentioned methods provide no feedback on the call quality of a particular VOIP session, the introduction of application specific metrics and call quality feedback protocols is necessary.

Before this is done another way to categorize QoS mechanisms is shown which can be divided into qualitative and relative Quality of Service mechanism [OS00]. On the one hand, qualitative, or relative QoS mechanisms relate two categories of packets to each other, like media-streaming and HTTP traffic, for example. On the other hand, quantitative, or absolute QoS mechanisms provide metrics for the QoS demand like that not more that 5% of the packets in a session must get lost. Since applications demand different QoS requirements the current approach to offer all types of connection the same "best-effort" service seems not sufficient according to [OS00].

In order to provide feedback on a call's quality, the following metrics are used to determine the performance of a VOIP session [Bad07, p.103], [KT01].

End-to-End Delay Is defined as the amount of time needed for voice to get from the mouth of a call participant to the ear of another. This includes the delays introduced by capturing and encoding the voice samples ($T_{capture\&encode}$) as well as the time needed to add them as a payload to a packet (T_{build_packet}). Moreover the actual transmission time is taken into account ($T_{transmission}$), as well as the time the packet stays in the jitter buffer at the receiver side (T_{jitter_buffer}). This is shown in Figure 2.7.

Jitter In order to simulate a real face-to-face communication, a VOIP call's data packages should be received and played back with a constant delay between each other, therefore being isochrone. Jitter is the relative difference in time upon the reception of two subsequently sent network packets from the same source. Jitter is produced by transporting e.g. voice samples on packet switched networks. The only significant source of jitter is the time voice

packets spend in queues of network devices. However, to compensate this effect, a so called jitter buffer is employed at the receiver's side. Its purpose is to capture a number of voice packets and hand them to the receiving user with equal delays between two succeeding samples. Moreover, Toral-Cruz et al. [TCTR05] observed that media transport-protocols carrying smaller voice payloads generate smaller jitter throughout VOIP session.

Packet Loss Rate This metric denotes the number of so far lost packets in a session. It may be used by a sender to change the packet rate of payload size used to transmit its data, or it may be used by a receiver to adjust its jitter buffer for example.

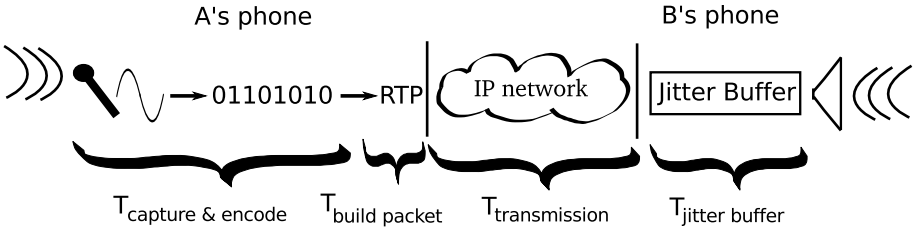


Figure 2.7: End-to-End Delay [Bad07, p.103]

To exchange these QoS feedback metrics additional protocols are needed. Such a protocol is RTCP which is specified as an addition to the RTP [87].

2.3.2 QoS Feedback Exchange Protocol RTCP

RTCP has been developed and specified by Schulzrinne et al in RFC-3550 [87]. The Real Time Control Protocol has been designed as a augmentation to its media transport pendant RTP. Therefore, its structure and functionality are tightened to the additional needs of RTP. RTCP, as well as RTP is a session layer protocol and does not rely on UDP as a transport protocol as well. However, its main purpose is to exchange information upon the packet loss, jitter and end-to-end delay of an ongoing RTP session. It achieves this by transmitting so called reports.

A report is an RTCP packet consisting of several blocks. There exist report blocks and other message blocks. A report block is either a sender or a receiver report block. A report block is called a sender report block if the participant sending an RTCP packet is also an active sender in a RTP session.

Sender Report block A sender report block contains information on the sent RTP packets, as well as a timestamp up to the time this report block is generated. See Table 2.1 for a detailed report block description.

Receiver Report block A receiver report block contains information on the received RTP packets, as well as the jitter value and some delay information up to the time this report block is generated. See Table 2.2 for a detailed report block description.

Valid message blocks are Source Description (SDES) blocks, Application (APP) specific blocks and Bye (BYE) blocks. Message blocks contain no call quality information. However, they may serve special additional needs in a session.

SDES block A Source Description message block may consist of several fields providing information about the participant in a session. A block containing a CNAME field is mandatory in every sent report. It is used to identify an RTP session(see section 2.2) including a corresponding RTCP session between two peers throughout the lifetime of the RTP/RTCP session. This is necessary since it is possible that SSRCs[7] change due to a collision. Therefore, the CNAME serves as a unique identifier for a call's duration, even if the SSRC changes. The CNAME should at least consist of the sender's IP address in dotted decimal representation (e.g. 192.168.0.1). Other fields like PHONE or NAME[8] are optional. For a more detailed description see [87, section 6.5].

APP block An Application specific block is intended for experimental use and may contain arbitrary information.

BYE block A Bye block is used to indicate the end of a RTP/RTCP session. It may contain a descriptive information for the reason a participant left.

Table 2.1: Sender Report Fields

Field-name	Description
Network Time Protocol (NTP) Time stamp MSW (32 bit)	Most significant word. This field must not reflect the actual wall-clock time since it may be used for round-trip time calculation in conjunction with the timestamps received from a corresponding receiver report block. In fact, it could be set to zero as well
NTP timestamp LSW (32 bit)	Least significant word.
RTP tim-stamp (32 bit)	This timestamp may be used for inter-media synchronization. It is calculated from the NTP timestamp with the same clock resolution as used by the clock retrieving the voice samples. In the case of Pulse Code Modulation A-Law (PCMA) as RTP payload type this would be 8000 Hz.
Sender's packet count (32 bit)	The amount of RTP packets sent until the time this report is built.
Sender's octet count (32 bit)	The amount of octets sent until the time this report was built.

Any of the mentioned report and message blocks may be sent as single packets as well. In this case, the blocks are equipped with a corresponding header. For further information on these headers see [87, section 6.3-6.7].

It is not possible to send RTCP packets whenever an application desires to. Instead, it must be guaranteed that an RTCP session does not use more then one fourth of a RTP session's bandwidth. This restriction is enforced by the RTCP protocol stack which schedules to be sent packets periodically. Therefore, the so called message and report blocks are sent in compound RTCP packets. A compound packet consists of several blocks which are assembled by the RTCP stack [87]. Moreover, there exist certain rules concerning the structure of a compound RTCP

[7] RTP and RTCP packets share the same SSRC in a session

[8] PHONE represents the Phone Address the user was reached at. NAME may show its name.

Table 2.2: Receiver Report Fields

Field-name	Description
SSRC (32 bit)	SSRC of the sender this report belongs to.
Fraction Lost (8bit)	Fraction of the lost RTP packets since the sent time of the last report. This field is interpreted as the fraction part of a a fixed point.
Cumulative number of packets lost (24 bit)	Total number of not received but expected RTP packets throughout this RTP session.
Extended highest Sequence Number (32 bit)	The highest sequence number of an RTP packet received so far. Since a sequence number is 16 bit wide and thus may wrap around fast if a high initial random offset has been used, a 16 bit wide cycle count is added to the sequence number as a high half word reflecting such overruns.
Jitter (32 bit)	Jitter value of the received RTP packets. It is calculated as a running average value.
Last Sent Report (LSR) (32 bit)	A simplified NTP timestamp consisting only of the low 16 bits of NTPs Most Significant Word (MSW) and the high 16 bits of NTPs Least Significant Word (LSW) of the last sender report received by that source.
Difference Last Sent Report (DLSR) (32 bit)	The delay between the last received sender report and the built time of this report. It is represented as a 32 bit wide fix point number with 16 bit fraction and 16 bit integer.

packet. A Sender Report block is always located first in a compound packet if the participant is a sender. Otherwise one or more Receiver Report blocks are encapsulated in the packet. The Receiver Report block is committed if the participant has not yet received any Sender Report from a peer. If it is the case that there are more active participants in a session than Receiver Reports can be sent in a single packet, a round-robin algorithm is applied to serve all session members equally. Next follows a mandatory SDES block consisting of at least a CNAME chunk[9]. APP and BYE blocks are optional, however, BYE blocks must always be located at the end of an RTCP packet.

RTCP packets are distinguished by their Packet Type field in the common RTCP header. Furthermore, the various RTCP types can be seen in Table 2.3).

The header of an RTCP packet consists of the following parts [87]:

Version Field The Version Field is statically set to the value of 2, like in RTP.

Padding Field This field denotes that there are padding bytes after the last compound block. The last padding byte reflects the number of padding bytes to be expected. This field might be needed by encryption algorithms with a fixed block size.

Report Count The RC field indicates the number of receiver report blocks in this packet. If there is also a sender report block available also this is denoted by the Payload Type field.

[9] An SDES block consists of several chunks. One chunk would be CNAME or PHONE for example. See [87, section 6.5].

Table 2.3: Valid RTCP Packet Types

Name	Identifier
Sender Report (SR)	200
Receiver Report (RR)	201
Source Description (SDES)	202
Bye block (BYE)	203
Application Specific block (APP)	204

Packet Type The PT field reflects the RTCP message type. Valid values are shown in Table 2.3

Length Indicates the packets length minus one in 32 bit words.

Synchronization Source The SSRC has the same value as in the associated RTP packets.

A packet may consist of several compound blocks as mentioned above. In this case, the packets type is determined by the first block contained in the packet. In a compound packet consisting of a Sender Report Block and a SDES message block the Packet Type would be "Sender Report", for example.

Considering the tear-down of an RTP/RTCP session the quitting party simply ceases sending RTP packets and RTCP reports. The call termination is then handled by the signaling protocol as mentioned in Section 2.1. where an overview on Quality of Service measurements in IP networks has been given. However, although numerous techniques are available it is not possible to guarantee certain resources to achieve isochronous behavior for a given VOIP session crossing several IP network borders. This is mainly the reason because of the vast variety of not necessarily compatible QoS services. Therefore, media applications like VOIP-phones have to provide several methods to achieve isochronous media-transport sessions on their own. This is done by using a jitter buffer as mentioned above. Therefore, in order to inform their communication partners, Quality of Service feedback protocols like RTCP are employed.

2.3.3 QoS Summary

The previous Sections introduced general QoS measurements on the one hand, and call-quality feedback mechanisms on the other hand. It has been shown that protocols like RTCP are not able to actually enforce QoS requirement, but are able to monitor their appliance. Thus, providing the gained information to algorithms deciding upon the redimension of the jitter buffer of a VOIP-phone for example.

Although RTP/RTCP has been introduced as the only wide-spread open media-transport and call-quality feedback protocols there exist closed or commercial solutions like Skype, which also relies on other methods to perform these tasks [BMMR09]. By now, the fundamental parts of a VOIP communication from the signaling phase until the actual call have been introduced by taking a look on transport protocols as well as their quality of service feedback protocols.

3 Designing a VOIP Stack in an FPGA

In the previous Chapter an overview on the processing of a Voice over Internet Protocol (VOIP) call was given. Signaling, media-transport and call-quality feedback protocols have been introduced. However, a description of the audio capturing, encode, and playback phases is still missing. The capture phase denotes the sampling and digitization process of analogue audio samples. Afterwards the digitalized speech is encoded with an appropriate audio code like G.711 (e.g. Pulse Code Modulation A-Law (PCMA)) and transmitted to the receiver. At the receiver's side the encoded voice samples have to be decoded and played back by speakers. [Bad07, p.132ff], [Gra10, p.192ff]. Therefore, a VOIP application consists of two major parts:

Data Processing part The data processing part of a VOIP phone includes signaling, media-transport and call-quality feedback protocols so far introduced, as well as the voice sample handling in the jitter buffer.

Signal Processing part The signal processing part includes the capture and encode phase, as well as the playback phase for voice samples.

If a VOIP-phone is implemented as a common Personal Computer (PC) application the hardware-software partitioning is straight forward. It is shown in Figure 3.1. The signal processing part is usually realized in hardware, by utilizing a sound-codec, like the AC97 from Intel, to perform the capture and playback functions. Furthermore, the encoding and decoding of voice samples in a specific codec like G.711, respectively, is primarily realized in software. This is also true for the data processing part. Common softphones acquiring this architecture are e.g., Ekiga [51], Kphone [59], Xlite [81], and Linphone [62] Moreover, Figure 3.1 shows the communication frequency of the mentioned modules throughout a VOIP session.

This data is derived from the SIP, RTP, and RTCP Request For Comments (RFCs). The thicker an arrow in Figure 3.1 is the higher is the communication frequency. The communication frequency is determined by the media-packet and the call quality feedback report transmission interval. As it can be seen in Figure 3.1 although the media transport protocol and associated function blocks produce the highest communication frequency, especially the call-quality feedback protocol block needs to access the media-transport block more frequently than it transmits reports. This is because the call-quality feedback protocol needs to gather statistical information from the media-transport protocol every time a new media-packet is sent or received. Thus, introducing a higher communication frequency. Furthermore, the arrows are divided into control (dashed) and data (continuous) arrows.

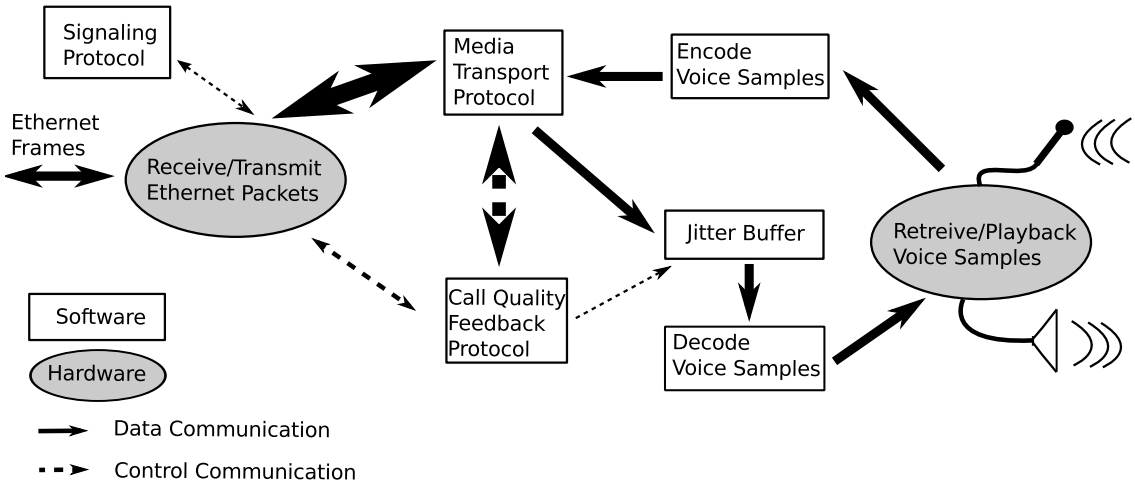


Figure 3.1: PC VOIP-Phone Architecture

When a VOIP phone has to be implemented in an embedded device, hardware-software partitioning depends largely on the used scenario. Therefore, a variety of VOIP related implementations on embedded platforms is presented in the Sections 3.1 and 3.2. Basically, it is possible to distinguish the mentioned approaches in the following way:

- General purpose processor based embedded platforms with dedicated Digital Signal Processor (DSP) or Intellectual Property Core (IP-Core) support to perform calculation intense work like transforming speech samples into voice codec samples are addressed in Section 3.1.
- System on Chips (SOCs) on Field Programmable Gate Array (FPGA) platforms with different dedicated hardware support according to the target scenario are introduced in Section 3.2.

Section 3.3 introduces the requirements and constraints induced by the target platform. Furthermore, Section 3.5 describes the provided IP-Core RTP-Engine, which implies a major constraint on the design decisions when implementing a VOIP stack on the target device. Finally, the Sections 3.6 and 3.7 introduce the abstract component interfaces as well as the HW/SW partitioning of the interfaces used by the VOIP stack. Moreover, several alternatives on implementing certain interface parts are shown.

3.1 CPU based Embedded Platforms for VOIP

This Section gives an overview on selected projects building a VOIP capable embedded system utilizing a given general purpose processor based platform. The first example introduced has been developed by Xuejing, Jinping and Xiulan [XJX10]. In their approach a TMS320DM642 DSP (with up to 650 MHz core clock frequency) development board from Texas Instruments in conjunction with an ARM9 processor is used to implement a VOIP unit, which is also capable of Video over Internet Protocol (IP) transmission. The DSP platform is used to implement the signal processing part of the VOIP solution. This does also include the speech sample to voice codec transformation process. The ARM9 processor operates a real-time operating system which performs the data processing part as well as the scheduling between the video related and the

audio related processes using the DSP. The communication between the DSP and the ARM9 processor is performed by the utilization of the High Performance Interface (HPI)[1]. Necessary synchronization messages are exchanged by mailboxes. Package reception and transmission is done by a regular Ethernet interface. Furthermore, the filtering and packet encoding process of media transport packets is done in software.

Another approach presented by Ho et al. [HTL+03], uses a similar method in comparison to the one by Jinping and Xiulan [XJX10] presented above. Again, general purpose processor based platform including a DSP and an ARM processor is used to implement a VOIP only solution. The utilized DSP is a TMS320C54x (100 Million Instruction Per Seconds (MIPS)) from Texas Instruments. It performs the signal processing part including encoding and decoding of media transport payload packets. To do so, the processor forwards the entire media-transport protocol packets to the DSP, which decides if a packet is a speech containing packet or from another type. In this approach the DSP is capable of providing several codec algorithms like, e.g., G.711, G.723.1 and G.729. The data processing part is performed by an ARM7 processor. A real-time operating system is used to schedule the needed tasks.

In [JF10], Jijiang and Fiu introduce a VOIP solution with a bluetooth interface for call control tasks, as well as headset and microphone connection. They use an ADM5120 VOIP platform from Infineon, housing a DSP, a MIPS processor and several UARTs to interface with the bluetooth transceiver. This time an embedded Linux operating system is deployed. Furthermore, the whole signal processing part, including encoding and decoding of speech samples to and from voice codecs respectively, is done by the DSP. The platform's DSP is capable of several voice codecs provided as software libraries to the DSP. The according voice samples sent to and received from the DSP are encapsulated into a voice transport protocol by the MIPS processor. Furthermore, a general purpose Ethernet interface is used to exchange media quality-feedback and signaling packets.

The last approach presented in this Section has been developed by Zhai and Wang [ZW10]. The authors utilize a Samsung S3C2410 platform containing an ARM9 processor and a Philips UDA1341 sound codec. In this approach, the ARM9 is directly connected to the sound codec with a dedicated interface. The whole signal processing part of a VOIP session including encoding and decoding of voice samples and packets respectively is done by the sound-codec and its DSP. The encapsulation of encoded voice samples is done by the ARM9 processor running an embedded Linux distribution. Furthermore, the utilized sound codec can be controlled with native Linux sound-drivers from the Advanced Linux Sound Architecture (ALSA) sound system. The utilized audio codecs are PCM encoded and range from 8 kHz to 32 kHz sampling rate. Again, a general purpose Ethernet interface is used to exchange media, quality-feedback and signaling packets.

The four approaches presented so far have in common that they use a given development platform including a DSP and a general purpose processor to perform their intended tasks. Apart from the fact that one of the presented approaches [XJX10] implies a video functionality, they differ from each other in several aspects. First, there were two methods used to handle media-packets. On the one hand, the entire packet was forwarded to the DSP by the Central Processing Unit (CPU) which extracted the payload. On the other hand, the CPU already performed the payload extraction and the DSP was only responsible for decoding the received samples. Second, the utilized operating systems differed. Two solutions [XJX10, HTL+03] utilized real-time operating systems to perform their tasks, whereas [JF10] and [ZW10] use an Embedded Linux distributions.

[1] HPI is an On-Chip interface being able to access the internal as well as the external memory of the DSP directly from the outside.

However, [ZW10] uses native Linux sound drivers from the ALSA package to communicate with the audio codec device. Considering the general purpose processors, [XJX10, HTL+03, ZW10] use ARM microcontrollers and [JF10] utilizes a MIPS processor.

The authors focused on general purpose scenarios, where no special constraints on terminal device delays exist. Furthermore, the utilization of software components to retrieve voice samples and encode them into appropriate packets introduces a minimum feasible packet size which might not be small enough for certain environments. The minimum feasible packet size is determined by the sampling rate of an audio codec, on the one hand. On the other hand, by the time a voice packet needs to be assembled, put into the transmission queue of a network interface card and transmitted to a communication partner. This has been described in Section 2.3.1. The smaller the amount of encoded voice samples in a packet, the smaller is the jitter. This has already been observed by Toral-Cruz et al. [TCTR05], as mentioned in Section 2.3.1. By implementing the parts of a VOIP stack in hardware which have direct impact in the minimum feasible packet size, the constraints on isochronous VOIP sessions mentioned in Chapter 1 can be fulfilled.

3.2 FPGA Embedded Platforms for VOIP

The first approach by Apostolakos et al. [AML+10] describes an attempt to implement an Internet Protocol Private Branch exchange (IP-PBX)[2] to VOIP gateway in an FPGA. The target platform is a Xilinx Spartan3 4000 FPGA, consisting of 4M gates, 27.648 slices and 1.728 Kbits of On-Chip Block-Random Access Memory (RAM). The SOC deployed on this FPGA contains two LEON3 32 bit RISC CPUs, whereas each CPU is a master on a discrete Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB). The two bus systems are connected via an AHB bridge. One of the LEON3 microcontrolers performs the data processing part and interacts with the Ethernet interface associated with the VOIP capable network. The other microcontrolers performs the signal processing part. However, in order to accelerate the signal processing tasks, the processor responsible for signal processing is equipped with a DSP extension realized as dedicated IP-core. Moreover, the signal processing capable processor interacts with the PSTN interface. Thus, the data flow of the system works as follows: Since the purpose of the system is to implement a VOIP to PBX Gateway, no sampling and playback functions of speech are necessary. Each incoming VOIP session related request is handled by the data processing microcontroller. Payloads are extracted from media-transport packets and forwarded to the signal processing controller. It performs the needed translations and interfaces with the PSTN network. The bridging between PSTN and VOIP networks is performed the other way round. From the software's point of view, an embedded Linux distribution is used on the data processor. The signal processing microcontroller runs a custom standalone implementation of the required algorithms. The supported voice codecs are G.711 and G.729.

Another approach to implement a VOIP to PSTN gateway on an FPGA has been introduced by Faroudja et al. [FIT+09]. The authors use open hardware IP-cores from the Opencores website[3] and a Xilinx Virtex II XC2V300 FPGA as platform device. This FPGA consists of 14.336 slices and 768 Kbit of On-Chip Block-RAM. The signal processing part is performed by a dedicated audio codec located on the FPGA development board. Considering the data processing part, an

[2] An IP-PBX gateway is used to connect Public Switched Telephone Network (PSTN) telephone networks with IP telephone networks [Bad07, p.14f].

[3] `www.opencores.org` - Visited on 01/17/2011

open 32 bit RISC processor is utilized. This processor is running uClinux to provide a runtime-environment for the appropriate algorithms and drivers. The IP-core interconnect is realized as a wishbone bus housing a 10/100 Ethernet interface to interconnect with appropriate IP networks. However, the paper does not mention any supported codecs and an interface to a PSTN network.

An approach from 1999 [SHR99], does not fit inside the System On Chip criteria of this Section. Nevertheless, it is mentioned here because of an interesting combination of hardware components. The authors propose the development of a VOIP capable cable modem. The design is split onto two printed circuit boards (PCBs). On the first board, the base system consisting of a general purpose processor and the needed peripherals to run a cable modem are deployed. An appropriate real-time operating system is installed here as well. The second board is designed as an add-on board housing an FPGA controlling two chips, which are responsible for FPGA control aspects and speech to G.711 conversion. Moreover, a voice compression chip is used which transforms G.711 encoded voice into a G.729 or G.723.1 compressed format.

The last paper introduced in this Section shows a VOIP phone implementation on a Xilinx XUP development board providing a Virtex II Pro FPGA [BW05].

The FPGA includes two Power-PC 405 32 bit general purpose processors, where one is used. The signal processing part of the VOIP stack is performed by an AC97 audio codec. Transceiving of network packets is done via an on-board Ethernet interface. However, the aim of Van den Braak's and Wong's work is to measure end-to-end delays in VOIP applications and to discover those components introducing the largest delay. Therefore, two parallel implementations are presented. The supported and tested codecs were G.711, G.726, G.723.1 and GSM. On the one hand, the whole decoding and encoding process of voice sample has been implemented in software. This has been identified to be the largest source of delay in terminal devices when using a GSM codec. On the other hand, the encoding and decoding phase were implemented in hardware, which lead to a significant lower delay on the terminal side. The whole data processing part is realized in software.

In this Section, four approaches to implement VOIP related application in FPGAs have been presented. However, they had different aims. Two projects [AML+10, FIT+09] proposed methods to implement PSTN to VOIP gateways inside FPGAs. Considering the signal processing related tasks both projects used different solutions. On the one hand, a LEON3 soft core processor with exclusive access to a DSP-like IP-core was used to perform these tasks [AML+10]. On the other hand, a dedicated sound-codec chip located on the FPGA development board was used [FIT+09]. Furthermore, the remaining two projects also differ in their application. In [SHR99] the design of a cable modem which may serve as an interface device for VOIP phones was introduced. An approach to identify the largest terminal device delay element in a VOIP call was shown in [BW05]. In this paper, two implementations were presented showing that the largest delay was created by a speech to GSM encoding/decoding block realized in software. However, the authors focused on general purpose scenarios, where no special constraints on terminal device delays exist. The autors of [BW05] proposed that an implementation of the encoding/decoding algorithms in hardware can reduce the terminal device delay in a VOIP session significantly. Nevertheless, Van den Braak and Wong did not mention that the the minimum feasible packet size is determined by the sampling rate of an audio codec. By the time a voice packet needs to be assembled, put into the transmission queue of a network interface card and transmitted to a communication partner. This has been described in Section 2.3.1. Although saving time by increasing the efficiency of encoding/decoding algorithms, Van den Braak and Wong could not reduce their media-transport packet payload sized in order to reduce jitter further. This is required for the use in sensitive

environments, as mentioned in Chapter 1. The smaller the amount of encoded voice samples in a packet, the smaller is the jitter. This has already been observed by Toral-Cruz et al. [TCTR05], as mentioned in Section 2.3.1. By implementing the parts of a VOIP stack in hardware which have direct impact on the minimum feasible packet size, the constraints on isochronous VOIP sessions mentioned in Chapter 1 can be fulfilled.

3.3 System Requirements and Constraints

Implementing VOIP stacks for various purposes in embedded devices is a well mastered task by industry and academia. Especially, when it comes to dedicated general purpose processor based VOIP platforms as shown in Section 3.1. However, the presented solutions' aim is to implement VOIP applications for general purpose use mainly. The only exception is the approach presented by [BW05]. The authors implemented a VOIP stack into an FPGA development board in order to identify certain sources of terminal device delays. As a consequence Braak and Wong proposed that especially the encode and decode blocks implemented in software should be realized in hardware instead [BW05].

However, to reduce the end-to-end delay on the terminal device further, it would be of use to implement the whole signal processing part as well as the encoding and packet building and packet reception functions in hardware. This will reduce possible non-determinism delays to an insignificant value, as mentioned in Chapter 1. Real-time operating systems can reduce this issue. However, all presented VOIP stack applications implemented the sample, packetizing, packet reception and transmission scheduling functions in software.

Therefore, a hardware-only implementation of the following functional blocks might be useful to make terminal device delays more deterministic:

- Capture and Encode of Voice Samples,
- Packetizing and Scheduling of Media Packets,
- Transmission and Reception of Media Packets,
- Filling and Mangling the Jitter Buffer,

Furthermore, this will also reduce computational pressure on low performance and low power embedded devices. In order to accomplish this task a specific IP-core fulfilling the stated requirements has been developed by Brunmayr et. al [BHS09].

However, what is still missing is an interface connecting the IP-core with a Base-System providing a general purpose CPU, memory, and several peripherals. The aim of the subsequent Sections is to perform a hardware/software partitioning of such an interface. Furthermore, the remaining parts of a functional VOIP stack like signaling and call-quality feedback protocol must be implemented to verify the interface approach.

In order to partition a hardware/software interface of a dedicated IP-core, as well as implement certain functions of a VOIP stack in software a reconfigurable hardware platform is required. The advantages of a reconfigurable hardware platform are that it is possible to exchange parts of the system quickly. Thus leading to shorter development cycles. A reconfigurable platform

offers the possibility to evaluate certain architectures and system components. Thus, an FPGA development board fits best to perform this task. However, certain constraints are opposed on the platform board. First of all, the FPGA must be large enough to fit a System-on-Chip, consisting of at least the following components, as shown in Figure fig:base-schem.

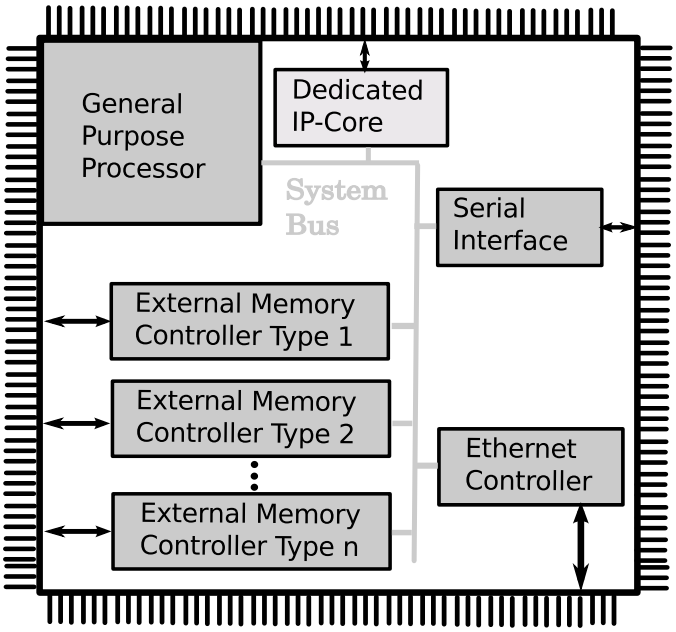


Figure 3.2: System on Chip base components

General Purpose Processor The CPU must be fast enough to handle a signaling protocol stack, a call-quality feedback protocol implementation, and certain administrative tasks.

Serial Interface This module is used for debugging and user I/O purposes.

External Memory Controllers In order to provide a variety of ways to run software on the Base-System's processor, several memory controllers should fit inside the FPGA in the development phase. Therefore, the platform board should provide various memory chips accessible from the FPGA.

Ethernet Controller An Ethernet controller is needed to let the system run a VOIP session, eventually.

Dedicated IP-Core This will be an extended version of the IP-Core introduced in [BHS09]. It is described in Section 3.5 in greater detail with a focus on its interface behavior.

Furthermore, additional space for data and instruction caches to the Base-System's processor would be an optional requirement. This would result in a faster program execution. Considering the target platform, the decision fell on a Xilinx ML405 development board, housing a Xilinx Virtex4 FPGA[108], because it fulfills all constraints mentioned above. Furthermore, the ML405 was easily available.. It provides a Virtex4 XC4VFX20-FF672 FPGA with 19.224 logic cells, 1.224 Kb on-chip Block, 128 Megabyte (MB) Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) with a 32-bit interface, 9 Megabit (Mb) Static Random Access Memory (SRAM), 8MB Flash, a Type I/II Compact Flash disk controller, an RS-232 serial

interface, an Audio Codec 97 (AC97) Audio Codec, several general purpose leds and buttons, as well as a Joint Test Action Group (JTAG) interface for programming and debugging purposes, and many more interface components [106].

3.4 Base-System Design

Although the development platform was predefined, other degrees of freedom exist considering the Base-System's general purpose processor and the way the VOIP stack concluding software elements are executed on it. Therefore, Section 3.4 deals with the evaluation of possible hardware and software components composing a Base-System.

As mentioned in Section 3.3, this Section surveys a number of possible design decisions considering the Base-System. The Section is logically divided into a hardware oriented part and a software oriented part. The hardware part shows implementation options for the deploying of a general purpose processor and consequences it will impose on the Base-System. The software part presents ways to run appropriate software on the system-on chip inside the FPGA. The first subsection introduces a survey on processors which can be used within the ML405 development board. Furthermore, this subsection shows various methods to boot software programs on the development board from a hardware point of view. The second subsection describes a selection of possible ways to boot and run software programs on a chosen processor.

3.4.1 Hardware Components

If general purpose processors are used within FPGAs one can distinct between two types of CPUs basically. On the one hand, there exist hard core processors. On the other hand soft core processors can be used. A hard core processor is a a processor already implemented within the FPGA [DB06]. In today's FPGAs not only processors are available as hard cores, but embedded multipliers and Block-RAMs, as well as other components are. Multiplier and Block Random Access Memory (BRAM) components do not have to be mapped to reconfigurable FPGA resources. Instead, they can directly be represented in special function blocks inside the FPGA. However, an appropriate interface must be provided by the designer. Their advantages are that they run at usually higher speed than soft core processors. Moreover, hard cores do not take up any reconfigurable FPGA resources, so called Look-Up-Tables (LUTs). Instead they are located on a separate partition on the chip. Furthermore, they are directly supported by the FPGA and toolchain vendor. Therefore, no porting and excessive testing must be performed. The disadvantages are that hard cores are mostly available on large and thus expensive FPGA and development platforms only. Hard core processors provide an interface to a system bus realized inside the programmable part of the FPGA. Usually, vendors provide a series of on-chip peripherals and off-chip peripheral interfaces which can be attached to the bus. However, the bus interface of a hard core processor can not be changed, which results in binding the user to a vendor specific bus-system. Nevertheless, bus-bridges translating the commands of one bus to the commands of another bus may be deployed to overcome this issue.

On the other hand, soft core processors can be used. They are processor models described in Hardware Description Language (HDL)s like any other component developed to run inside an FPGA [TAK06]. Therefore, these IP-Cores have to be synthesized and take up FPGA resources. A soft core processor's advantage is, that it is possible to tailor it to specific needs, like changing

it's interface to a specific bus-system, if allowed by its license. Furthermore, there are two types of soft core processors. Those, directly support by the FPGA, thus no porting and testing is needed, and those who are designed by the open-source community, or provided by other vendors. However, in this case it might be possible that the processor's model has to be adapted for the use in a specific FPGA, as mentioned above.

Considering our target platform both types of processors may be used, because the Virtex4 FPGA on the development board provides a hard core processor. In this case it is a 32 bit Power-PC 405 processor from IBM, supporting the IBM Processor Local Bus as an on-chip bus. The processor runs up to a speed of 300 Mhz and provides a Memory Management Unit (MMU). It offers no integrated Floating Point Unit (FPU). Moreover, it provides the ability to configure instruction and data cache separately by utilizing available Block-RAM on the FPGA [104].

Furthermore, Xilinx provides its soft core processor Microblaze. This processor runs with a speed up to 200 Mhz on a Virtex 4 FPGA and provides no MMU. It needs 1269 resource elements on a Virtex4 FPGA[TAK06]. It provides an optional interface to IBMs Processor Local Bus (PLB), as well as to Xilinx's Fast Simplex Link (FSL) which can be used to build Multi-Core systems [105]. Furthermore, no FPU is available to the processor directly. However, this soft core cannot be modified due to its license.

An open-source soft core processor would be the LEON3. It can be modified, provides no MMU, but an FPU [TAK06]. The LEON3 is entirely developed in Very High Speed Integrated Circuit Hardware Description Language (VHDL) and provides an interface to the AMBA bus from ARM ltd. Moreover, it runs with up to 125Mhz in an FPGA [TAK06].

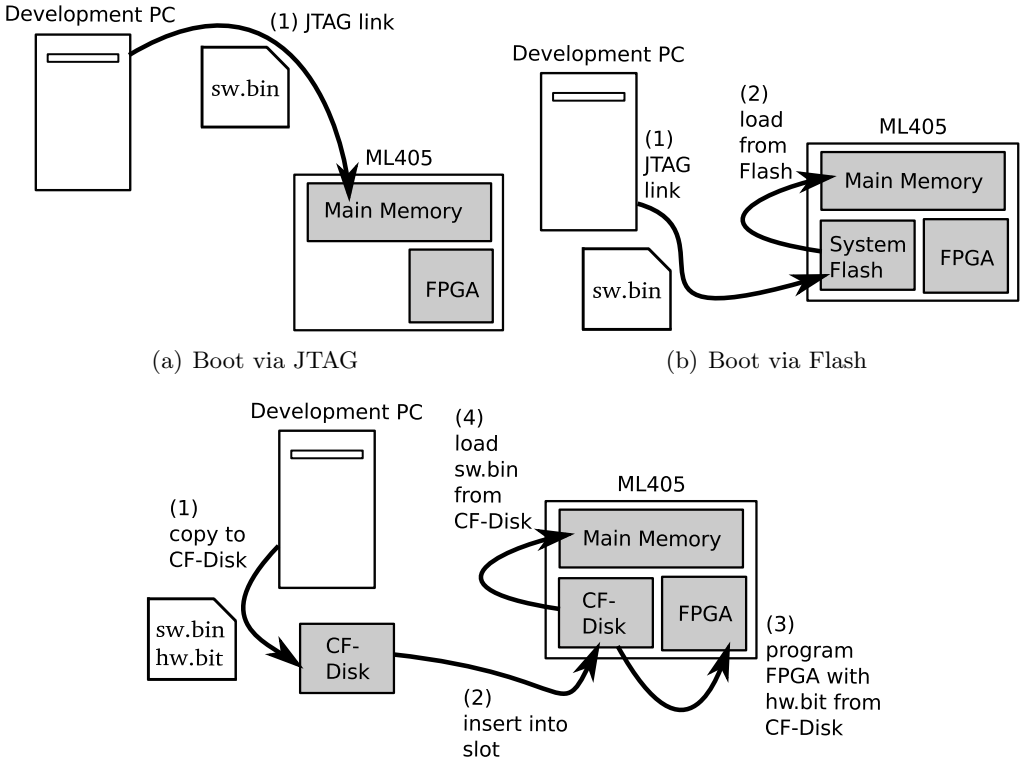
3.4.2 Booting the Base System

The second part deals with several ways to start software on a processor running inside the Virtex4 FPGA. Basically, ML405 development board provides three ways to provide software programs to a utilized processor. The three approaches are presented in Figure 3.3.

The first method is to download a program in the appropriate binary format from the development workstation directly to the main memory of the processor utilizing the JTAG Debug interface **(1)** [108]. This is illustrated in Figure 3.3(a). After downloading the processor's program counter is set to the start address of the program. This method is mainly used in the development phase of a project, where frequent changes to software and possibly hardware occur. Moreover, every time the target device shall be used, a workstation capable of downloading the desired software is needed.

The second way to load a program into a processor's main memory is to utilize the development board's flash chip. This is illustrated in Figure 3.3(b). In this approach a program has to be available in an appropriate binary hex format, which is then downloaded into the flash via the JTAG Debug interface once **(1)**. From now on, if the Base-System is configured appropriately, every time the processor is powered up it loads the program from flash **(2)** into main memory [108]. This can be achieved with a small small bootloader located in on-chip BRAM on the ML405.

The third, and last method mentioned here, utilizes the Compact Flash (CF) disk to store the program. This is illustrated in Figure 3.3(c). In this special configuration an FPGA's configuration bit-stream and the program to be executed are both located in a special archive on the flash



(a) Boot via JTAG

(b) Boot via Flash

(c) Boot via CF-Disk

Figure 3.3: Ways to boot software on the ML405

disk **(1)**. The the flash disk has to be inserted in the flash disk slot of the development board **(2)**. If the development board is empowered and configured properly the archive is read from the flash disk. In the next step, the bit-stream is extracted and the FPGA is configured with it **(3)**. The last step stores the program, which must be available in the appropriate binary format, into the processors main memory **(4)**. The processor's program counter is set to the appropriate address and the program execution starts [108]. A map of available booting methods and their support by the presented processors can be seen in Table 3.1.

Table 3.1: Boot Methods

Boot Method	PPC405	Microblaze	LEON3
JTAG Debug Interface	Native	Native	Requires Porting
Flash Memory	Native	Native	Native
CF-Disk	Native	Native	Requires Porting

The word "Native" in a cell of the table means that no additional hardware must be developed to use this boot methods with the corresponding processor. "Requires Porting" means that it is necessary to adjust parts of the processor, develop additional hardware, or change certain configuration parameters of the vendor's toolchain to operate this boot method with the corresponding processor. An overview on the decision relevant features of the processors is given in Table 3.2.

Table 3.2: Decision Relevant CPU Features

Name	Hard/soft core	Availability	MMU/No MMU	FPU/No FPU	Bus Interface
PPC405	hard core	YES	MMU	No FPU	PLB
Microblaze	soft core	YES	No MMU	No FPU	PLB, FSL
LEON3	soft core	NO	No MMU	FPU	AMBA

The fields used in the table have the flowing meaning:

Hard/soft core It has direct impact FPGA resource usage, and the adaptability of the design. Furthermore, this decision has influence on the maximum frequency the processor can operate with [TAK06, DB06].

Availability This field indicates wether a core is natively available on the desired platform, or if the processor has to be ported to it. This includes pin assignment tasks as well.

MMU/No MMU The presence or absence of a Memory Management Unit has direct impact on the usability of certain operating systems.

FPU/No FPU An FPU is especially needed if intense calculation like encode/decode operations of certain voice codecs are necessary.

Bus Interface Describes the natively available interface of the on-chip peripheral bus. This has direct impact on the usable IP-cores.

3.4.3 Choosing the CPU

Eventually, the **Power-PC 405** hard core is chosen as a general purpose processor for the System-on-Chip realized in this theses, because of the following reasons:

- It is natively supported by the vendor, therefore, no porting is required.
- The processor uses the PLB bus to interact with peripheral devices. The PLB is directly supported by peripherals provided by the FPGA vendor.
- It does not need any reconfigurable FPGA resources, except for an interface wrapper connecting the processor to the FPGA-design, since it is implemented as a hard-macro.
- The processor features an MMU, thus standalone applications and full-featured operating systems may be utilized.
- Although the CPU does not provide a FPU, it is well suitable for this task because no complex arithmetic calculation needs to be done by the data processing part of the VOIP stack in software.

The used boot method will be "CF-Disk" from Table 3.1. In this method the media is split into two partitions. The first partition provides the FPGA's configuration file, as well as the bootloader, or standalone application, concatenated in one single archive. The second partition

is able to hold the operating system's root file-system, for example. Due to this approach, all relevant data considering hardware and software of the employed system are kept in one place, therefore, decreasing the time needed to prepare and download hard- and software components during development.

3.4.4 Software Components

The subsequent Section 3.4.4 gives an overview on the options available when one wants to run a VOIP stack on the ML405 development board. In Section 3.4.1 a number of proper processors, as well as boot methods available to run the Base-System were introduced. The decision fell on the Power-PC 405 processor. This Section will show a variety of ways to operate a VOIP stack's software components on this Base-System.

The first possible approach presented here is the so called standalone application. In this case only the programming language of choice's standard library and some vendor contributed support libraries are available, assuming that an appropriate toolchain (compiler, linker) exists for the target processor. The advantage of this method is that the developer has full control on the used processor, since no privilege separation is enforced by an operating system. Furthermore, the delay introduced by a system call's overhead and the non-determinism introduced by an operating system's scheduler must not be taken into account. Therefore, for projects with hard real-time requirements and low task parallelism, the standalone application approach may be useful. However, the disadvantages are, that a large amount of the application and possible supportive libraries might have to be ported to the target board, or entirely written from scratch. This would result in longer development times, due to a possibly higher level of complexity. Moreover, the possibility to introduce severe bugs rises. Furthermore, all protocols and components needed for a VOIP stack implementation must be designed and implemented from scratch. Nevertheless, another advantage of the standalone application is, that it makes no a priori assumption on the provided features by a processor. The standalone application would be fully supported by the PPC-405 based platform since a complete toolchain exists from the vendor. Moreover, several drivers and support libraries are available.

The second way to run a VOIP stack on the ML405 development board is to use the Xilkernelenvironment [103]. It can be seen as an example for a Real-Time Operating system. The Xilkernelenvironment provides a real-time capable Portable Operating System Interface (for Unix) (POSIX) compliant abstraction layer to the underneath hardware. It serves a thread model, socket interface, software timers and amongst others, a user space interrupt handling Application Programming Interface (API) . This API makes it possible to directly attach interrupt service routines to the interrupt controller of the used processor. Thus no abstraction layer is blocking direct access to an Interrupt Service Routine (ISR) . This results in a lower overhead when interfering with interrupt driven hardware from the application layer. Furthermore, the Xilkernel can be configured to a certain amount, to resize its footprint and features to a projects direct needs. It provides its own toolchain for the Microblaze and the Power-PC processors. Furthermore, it does not necessarily require a MMU capable processor. Although providing an operational run-time environment to develop applications, all software components and protocols needed by a VOIP stack would need to be ported or entirely engineered from the ground up to run with Xilkernel. Moreover, an implementation would be highly dependent on Xilinx specific target devices.

The third method presented in this Section is to utilize a wide-spread operating system available for the platform. In this thesis, Embedded Linux is mentioned as an example. First, it must

be noted that choosing Embedded Linux does not make any assumption on the availability of an MMU on a processor. Moreover, it does not make any assumption on the chosen processor mentioned in the previous subsection. Linux is also available for the architecture of the LEON3 processor, which is fully SparcV8 compliant [TAK06]. However, it is necessary to use a modified version of the Linux kernel to operate it on a MMU-less CPU. Therefore, uCLinux has been developed, which is integrated in the mainstream kernel by now [75]. Nevertheless, also full featured Embedded Linux distributions can be taken into account if the Power-PC hard-macro is chosen. Regardless of the utilized Linux kernel it provides a fully POSIX compliant API. However, when using uCLinux it must be mentioned, that no user/kernelspace separation is enforced. Thus a crashing application might crash the whole system, making uCLinux a bit more fragile to handle. Moreover, the *fork()*[4] system call is not implemented. Nevertheless, there are several supportive libraries available when designing a VOIP stack. Considering a Vanilla Linux Kernel[5] these restrictions do not apply.

However, when utilizing an embedded operating system like Linux the need for a bootloader arises. Depending on the used boot method a bootloader is also necessary when using the Xilkernel and the standalone application method. Nevertheless, when using an Embedded Linux distribution on the ML405 the bootloader needs to perform the following key-task as well. The bootloader's purpose is to download the Linux kernel from a boot media, unpack it, start it, and hand control over to the kernel. Considering embedded bootloaders, the only one which is available on a variety of platforms is U-Boot [CCA07]. Moreover, the bootloader needs to pass platform specific hardware information to the kernel. In the case of a PC, this information is retrieved and handed over to the operating system by the Basic Input Output System (BIOS), or the Uniform Extensible Firmware Interface (UEFI) [76]. However, in embedded devices no such unified approach exists. Therefore, board and vendor dependent data structures must be used in order to provide these information. Because this non-uniform approach on handling such sensitive platform data leads to various problems and large complexity when porting a bootloader to a new platform, the open firmware flattened device tree has been introduced [GL08]. The flattened device tree represents a uniform data structure in a tree-like shape, where each node is a hardware component present in the system and each leave is a property of such a component. It is built once in the development phase and retrieved by the bootloader from an available media. Subsequently, it is passed to the Linux kernel. The flattened device-tree is supported by U-Boot.

Finally, a summary of the different software approaches showing the dependencies of certain hardware and software components is presented in Table 3.3 In this table the field *bootloader* shows whether it is necessary to deploy a bootloader in order to use the corresponding approach. Furthermore, the filed MMU reflects whether an MMU must be present in order to use the according software approach. If *Not Needed* is written in any of these fields, the presence of this component is optional. Therefore, the approach would also succeed if the corresponding component is present in the design.

This Section introduced a number of available options when designing a Base-System on an a priori chosen ML405 development board. The tables 3.2 - 3.3 present the available choices and their dependencies upon each other. In Section 3.4.1 the Power-PC 405 processor and the corresponding PLB peripheral bus has been chosen to operate the Base-System. Considering software an Embedded Linux distribution will be used to provide the necessary run time environment for

[4] Is a system call to create a child-process differing only in the Process Identifier (PID). However, file-locks and pending signals are not passed on. [55]

[5] The official maintained kernelsource-tree by Linus Torvalds [64].

Table 3.3: Software Options

SW Approach	bootloader	MMU	Processor
Standalone	Not needed	Not needed	Microblaze, Power-PC, LEON3
Xilkernel	Not Needed	Not Needed	Microblaze, Power-PC
uCLinux	Needed	Not Needed	Microblaze, Power-PC, LEON3
Linux	Needed	Needed	Power-PC

the VOIP stack. The decision fell on embedded Linux because it natively supports a wide variety of peripheral drivers and libraries. Furthermore, there exists a port to the ML405 development board. Thus also a complete toolchain is available. Using embedded Linux will result in a more platform independent VOIP stack implementation, considering the software implemented part. Moreover, the processor features an MMU, thus there is no need for uCLinux, Xilkernel, and standalone applications. Deciding on embedded Linux implies to use the bootloader U-Boot, because certain platform specific information must be supplied to the Linux kernel by the bootloader in order to work properly. This information is stored in the open firmware device tree data structure as mentioned above. This leads to the following Base-System configuration:

CPU/BUS Power-PC 405 hard core including several PLB bus masters.

Software approach An Embedded Linux distribution including an appropriate toolchain for the Power-PC 405 processor.

Bootloader U-Boot with scripting support and support for the flattened device-tree.

3.5 RTP-Engine Description

The previous Section introduced a number of ways to configure a Base-System to interface with a dedicated IP-Core performing the signal processing, as well as media-packet reception and transmission part of a VOIP session. Furthermore, the decisions on the main components of the Base-System have been presented. This Section describes the dedicated IP-Core developed by Brunmayr et al. [BHS09].

The IP-Core described by the authors is capable of receiving RTP-packets, extracting their payload, decoding it, and playing the voice samples back over a speaker. Thus, realizing the whole part of a VOIP session concerning data payload This IP-Core is called "RTP-Engine". To receive and playback voice sample encoded in RTP-packets an AC97 audio codec, which is located off-chip, and an Ethernet [89] Media Access Control (MAC) hard core is used. However, only the receiver's functionality is realized on the ML405 development board.

The remainder of this Section is divided into two parts. In the first part 3.5.1, the soft core's architecture and its behavior considering transmission and reception of network packets is described. In the second part of this Section the IP-Core's interface is examined. This also includes its external interface, as well as the interfaces performing sub-system communication.

3.5.1 Architecture and Behavior

From an architectural point of view the RTP-Engine can be divided into a sender and a receiver sub-system. The sender sub-system consists of the following function blocks, marked gray in Figure 3.4, showing the IP-Core's architecture:

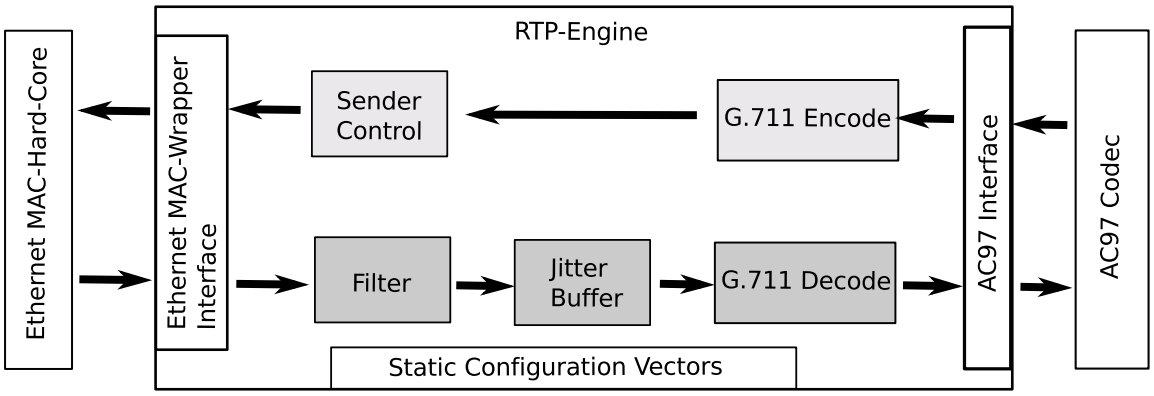


Figure 3.4: Standalone RTP-Engine

AC97 Interface This interface provides read access to the AC97 off-chip sound codec in the context of the sender sub-system.

G.711 Codec The codec encodes the raw voice samples retrieved from the AC97 sound codec.

RTP-Encoder This function block builds an appropriate RTP-packet consisting of an Ethernet, an IP, an UDP, and an Real-time Transport Protocol (RTP) header, as well as the voice samples as payload. All header information, except the RTP sequence numbers and timestamps are taken from static configuration vectors specified at synthesis time.

Sender Control This module queries the Ethernet MAC in order to hand a readily built frame for transmission.

Ethernet MAC Interface This module provides an interface to the Ethernet MAC IP-Core. However, a corresponding packet is not directly written to the Ethernet MAC, but is stored in a First In First Out (FIFO). This FIFO is needed if two parts of a circuit are supplied by different clocks. Thus, the FIFO serves as a buffer and synchronization memory. This is necessary when one circuit part supplied by clock A wants to transmit data to a part supplied by clock B, where the clocks A and B do not provide the same frequency [Gin03]. This is the case in the MAC-core, since the NIC-Wrapper is driven by a 100 MHz clock and the MAC-core is supplied by a 12.5 MHz clock if running at 100 Mb/s speed.

The functional blocks of the RTP-Engine's sender part presented above interact with each other in the following way to build a packet to be transmitted. First, the AC97 sound codec samples at an 8 kHz rate to retrieve the appropriate voice samples [86]. In the next step, a retrieved sample is fed into the PCMA (G.711) encoder, which is realized as a look-up-table. The voice sample gained is now stored in a sample buffer inside the RTP-Encoder module. These steps are repeated until enough samples are stored inside the buffer. In the subsequent step, the RTP-Encoder queries the Sender-Control module. If the MAC-Core signals the Sender-Control unit

that it is ready to send, the RTP-Encoder starts to move a data packet from its internal buffer to the MAC-Core, which then sends the RTP-packet of its Ethernet interface.

The receiver sub-system of the RTP-Engine is marked in a dark gray in Figure 3.4. It consists of the following sub-modules:

Ethernet MAC Interface This interface signals that an Ethernet frame has been received and stored inside its buffer memory.

RTP-Filter This function block checks whether a received Ethernet frame is an appropriate RTP-packet belonging to an ongoing RTP-session, thus having a correctly set SSRC. If yes, the packet is accepted, if no, it is discarded. Moreover, the Filter also discards packets which do not pass one of the implemented tests: Ethernet frame, IP, Universal Datagram Protocol (UDP) and RTP. The Filter is able to cope with IPv4 [83] and IPv6 [101] packets, as well as VLAN-tagged [88] Ethernet frames.

Jitter Buffer As mentioned in Section 2.3, the jitter buffer is used to restate isochronous behavior in a media-session.

G.711 Codec This time the G.711 codec is used to decode PCMA samples into raw voice samples and hand them over to the AC97 sound codec interface.

AC97 Interface The AC97 interface provides write access to the AC97 off-chip sound codec in the context of the receiver sub-system.

The reception of a RTP-packet is handled in the following way. When the Ethernet MAC Interface provides a new packet and the Filter is ready it subsequently tests two octets of a received packet. It is required to fulfill the following conditions imposed by the Filter block:

- Source and destination MAC address match the statically configured one inside the Filter.
- The IP header of the received packet is either an IPv4 or an IPv6 header containing the correct source and destination addresses.
- Source and destination port of a received UDP packet equal the ones configured inside the Filter.
- The payload of the UPD packet is of type RTP. Furthermore, the Synchronisation Source (SSRC) of the received packet must match the one expected by the filter.

If any of these properties are not fulfilled the received packet is discarded. If all properties match, the packet's payload is forwarded to the Jitter buffer, which does not support packet reordering by now. The Jitter buffer is realized in on-chip Block-RAM and is emptied upon overflow. When no overflow occurs, the payload is handed over to the G.711 codec which processes the voice samples to the AC97 audio codec, where it is played back via a speaker.

3.5.2 Interfaces

In the previous part it is shown how the components of the RTP-Engine interact with each other. However, it was not stated which interfaces the components use to exchange data. Moreover, the RTP-Engine's interface to other system components is yet to be discussed. This is the purpose of this Section.

First, the IP-Core's external interfaces are presented. The RTP-Engine provides several User IOs like buttons and leds amongst the usual clock and reset inputs. The buttons are used to control the AC97 sound codec's volume control. The leds are provided as visual feedback on certain operational conditions like a discarded incoming packets, or a Jitter buffer overflow. Furthermore, the soft core provides an interface to the Ethernet MAC hard core and to the AC97 sound codec. The AC97 audio interface is realized as a serial interface, complying to the AC97 standard [90]. Since the MAC hard core is interfaced through a wrapper a special interface is deployed by the FPGA vendor to interface with this wrapper, the Local Link Interface (LLI).

The LLI is a synchronous point-to-point handshake protocol, where a sender side can issue the availability of data to be written, and a receiver can indicate that it is ready to receive data. Furthermore, message start word and a message end word signal is used to indicate the beginning and the end of a transmitted message. This is necessary because it is possible that either receiver or sender postpones an ongoing transmission. The interface consists of the following signals, where the clock signal is omitted [110]:

Data This signal represents the data line. Usually, this is a bit-vector, but it can also be configured as a serial line. Considering the Ethernet MAC interface an eight bit wide word is transmitted every clock cycle.

Start of Frame If this signal's level is low, it indicates that the word just sent is the beginning of a message. A message may consist of one or more words. Furthermore, Start Of Frame (SOF) and End Of Frame (EOF) are not required for a correct operation of the LLI. However, they can be used as additional signals to recognize beginnings and endings of frames.

End of Frame If this signal has a low value it indicates that the word just sent is the last word of a message.

Source Ready The Source Ready signal indicates that the sender is ready to send a message. The signal is low active. A sender is able to de-assert this signal while a message is transmitted. In this case, the receiver knows that the sender postpones the transmission and stops receiving subsequent words until the Source Ready signal is asserted to low again.

Destination Ready This signal corresponds to the Source Ready Signal. Thus, the receiver may indicate that it is not ready to receive further words during a transmission, and therefore, postpone it by setting this signal to high.

An example transmission showing a Local Link Interface based message transmission is shown in Figure 3.5.

First, the sender sets its "SRC_RDY" signal to low, indicating it has new data available. In the next cycle the receiver answers with its "DST_RDY" signal set to low. By now the first word of the message can be transmitted. This is done in the next cycle, where the Start Of Frame

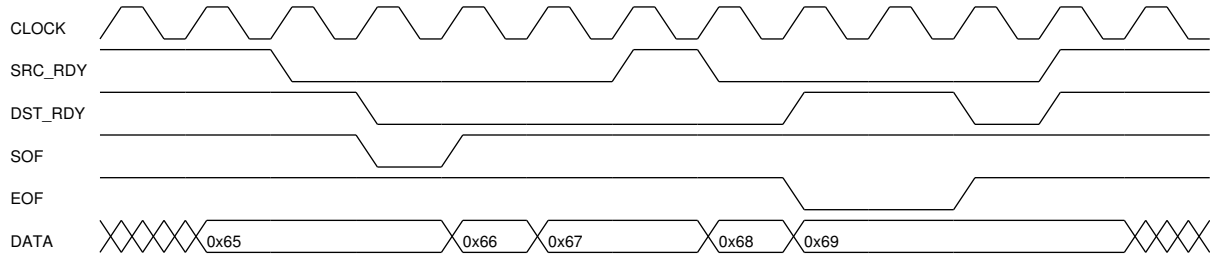


Figure 3.5: Local Link Interface Message Transmission Example

indicator "SOF" is set to low. However, a few cycles later the sender is not able to provide any more data, although, the message is not yet transmitted completely. Therefore, it sets its "SRC_RDY" signal to high again, to indicate the receiver that the transmission is postponed. In the subsequent cycle the "SRC_RDY" signal is set to low again, and the transmission can continue. One cycle later, the sender tells the receiver that the last word of this message is transmitted. However, the receiver postpones the reception by setting its "DST_RDY" signal to high for two cycles. After that, the transmission can be completed by the receiver asserting low to "DST_RDY" again. Because the transmission is over, all control signals are set to high again, thus being ready for the next transmission. Moreover, the sub-module communication within the RTP-Engine is realized with the LLI.

In this Section an overview on the RTP-Engine has been given. The structure and the behavior of the IP-Core is presented. Furthermore, the interfaces exported by the Engine are described, as well as the sub-module interfaces are mentioned.

3.6 Interface Concept

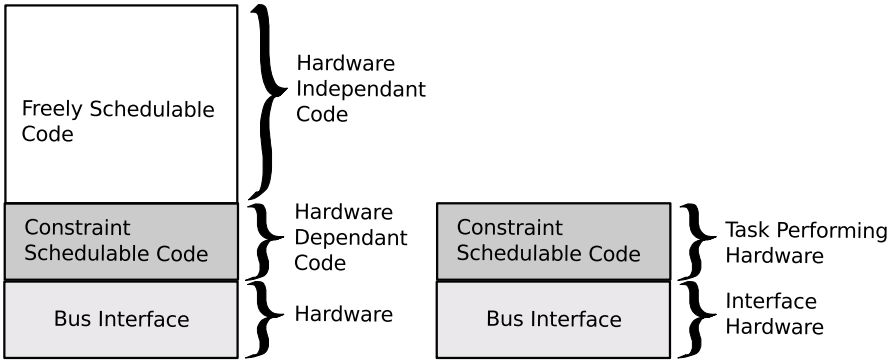
Although, the RTP-Engine provides the whole data-path of the media-transport relevant parts of a VOIP session, as well as the signal processing part, it is still missing an interface providing configuration and feedback data to and from the RTP-Engine. However, these decisions impose several constraints on the further design of the IP-Core's interface. However, before these constraints are examined, the general structure of an interface crossing the hardware/software boundary, according to Gajski et al. [G.09, p.255ff] is presented. The authors explain that an interface connecting a custom hardware module may consist of the following parts:

Freely Schedulable Code This part is intended to be software whose execution can be independently postponed and revoked by a scheduler. This can be the hardware independent part of a hardware driver for example. It usually calls functions contained in the Constraint Schedulable Code.

Constraint Schedulable Code The constraint schedulable code is software which has to obey certain timing compulsions. This can be the hardware dependent part of a driver, directly interfering with the Bus Interface.

Bus Interface This interface component is implemented in hardware. It may directly reflect the bus controller of a CPU.

This abstract representation shows a reference model when designing hardware/software interfaces. It can be seen in Figure 3.6(a). Furthermore, it can be interpreted as a model for an interface belonging to a general purpose processor, being the master of a peripheral bus. Considering the design of a peripheral, the layers "Bus Interface" and "Constraint Schedulable Code" can also be applied to a peripheral IP-Core, (see Figure 3.6(b)). As a consequence "Constraint Schedulable Code" may be realized in hardware as well, thus reflecting the parts in the peripheral performing the intended tasks.



(a) HW/SW Interface Model [G.09] (b) HW/HW Interface Model [G.09]

Figure 3.6: Interface Models

Although the presented model is primarily used to reflect certain needs when synthesizing interfaces, it may also be used as a reference model when designing a hardware/software interface for a given IP-Core like the RTP-Engine. Thus, providing a pattern for the partitioning of such an interface. A supportive argument for such a point of view is provided by Jensen, Madsen and Pedersen [JW05]. The authors conclude that there exists a need to model interfaces at a lower grade of abstraction, than the system model. This is necessary in order to gain information on certain performance bottlenecks introduced by an intuitive approach when designing hardware/software interfaces. Moreover, Jerraya and Wolf [JW05] mention, that due to the higher complexity in SOC design, and the tighter entanglement of hardware and software components, the need for a more interface focused design arises. Due to these reasons and their enhancing arguments presented above, this Section will give a detailed overview on the interface design of the RTP-Engine.

Since the ML405 development board does not provide two Ethernet interfaces, the available Ethernet hard core MAC has to be used by both, RTP-Engine and general purpose networking interface [106]. This makes the RTP-Engine a sub-module of a bigger IP-Core including the RTP-Engine's functionality and the network interface's functionality, leading to the architecture shown in Figure 3.7.

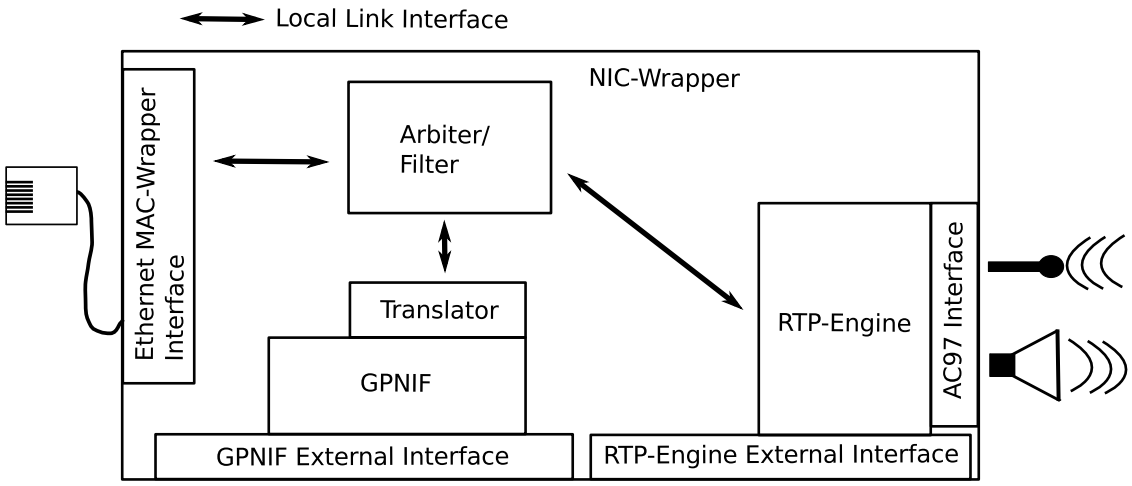


Figure 3.7: NIC-Wrapper Architecture

The reason for putting these two components in a wrapper block is to keep the interfaces clear and simple. Since both modules need the same resources and provide similar services, they can be integrated into a single wrapper module easily. The only additional control blocks to be added are an arbiter and a module receiving the network packets which are not recognized as RTP-session aware packets by the RTP-Engine's Filter module. The arbiter decides which sender, RTP-Engine or network interface, gets access to the MAC hard-macro transceiving the Ethernet frames. In order to clarify the terminology used the term NIC-Wrapper (Network Interface Card Wrapper) is introduced as an umbrella term for the whole IP-Core. The term RTP-Engine refers to the RTP-Engine including its interfaces. Furthermore, the term General Purpose Network Interface (GPNIF) is used to address the parts of the NIC-Wrapper belonging only to the GPNIF. Moreover, sub-modules not belonging to the RTP-Engine or the GPNIF are assigned to the NIC-Wrapper. Furthermore, the NIC-Wrapper is divided into external and internal interfaces. External interfaces are those interfering with components outside the NIC-Wrapper like hard cores and memory controllers. Internal interfaces are used inside the NIC-Wrapper to connect the arbiter to the RTP-Engine, for example. Note that, although the RTP-Engine's interfaces to the MAC hard core and the AC97 audio codec were introduced as external interfaces in Section 3.5, they are now propagated to the external interfaces of the NIC-Wrapper.

3.6.1 External Interfaces

This Section examines the data to be exchanged in order to let the NIC-Wrapper transmit and receive Ethernet packets. This includes the interfaces needed by the RTP-Engine and the GPNIF. Because the RTP-Engine performs the entire signal processing part, as well as the packet mangling in hardware a full set of network protocol stack headers has to be provided to the IP-Core as configuration data. The Tables 3.4 to 3.7 give an overview about the requirements imposed on the RTP-Engine's configuration interface. The field number (#) of octets describes the amount of memory octets needed to store the corresponding configuration data. If the addressed field is needed by both, the sender and receiver units of the RTP-Engine, the first number in the multiplication is set to 2. Otherwise, if the field is only needed by the sender, it is set to 1.

Ethernet Protocol

The remaining external interfaces of the NIC-Wrapper consisting of the interfaces to the MAC hard core and the AC97 Codec are discussed in this Section. First the external interface of the RTP-Engine is described, starting with an examination of the Ethernet protocol header. Since the target platform uses an Ethernet MAC hard core as a network transceiver the Ethernet protocol header is the first one needed. The RTP-Engine's configuration interface supports Ethernet frames of the type 802.3 only. Moreover, the preamble at the beginning of the frame, the Start Frame Delimiter (SFD) octet, as well as the Frame Check Sequence (FCS) fields are omitted because they are supplied by the hard core. Nevertheless, the fields Source, Destination, and Typ-Field must be provided. Table 3.4 shows the Ethernet protocol header information to be transmitted to the IP-Core, as well as the amount of memory needed in the RTP-Engine to store it [89].

Table 3.4: Ethernet Protocol Costs

Field	# of Octets	Comment
Source Address	$2 * 6$	The source address is needed by the module Sender Control and Filter.
Destination Address	$2 * 6$	The destination address is needed by the module Sender Control and Filter.
Type	$1 * 2$	The Type field is also needed by both modules, but is the same for Sender Control and Filter.
Sum	25	The amount of octets needed to store the configuration data for the Ethernet protocol.

However, because a sender's destination address is a receiver's source address and vice versa, the amount of memory needed can be reduced by 12 octets and therefore by 48%.

IP Protocol

In order to send properly formatted IP packets a complete header including a precalculated checksum is needed. The interface will support IP version 4 only. Therefore, only this protocol header has to be stored in the IP-Core. Nevertheless, Table 3.5 shows the data needed to be stored in the RTP-Engine [83]. The Table consists of the same fields with the same meanings as Table 3.4. However, because a sender's destination address is a receiver's source address and vice versa, the amount of needed memory can be reduced by 8 octets. Furthermore, the Protocol, and Version fields must also be the same for a sender and a receiver, which results in another interface memory reduction of 1.5 octets. Moreover, the "Options" field of the IP-header is unnecessary for the most common communications, according to [83]. Thus it will not be implemented in the RTP-Engine's interface. This will free another 40 octets, optimizing the interface by 75%.

Table 3.5: IPv4 Protocol Costs

Field	# of Octets	Comment
Version	$2 * 0.5$	The Version field of the IP header, statically set to 4. This field is needed by the modules Sender Control and Filter.
IHL	$1 * 0.5$	The length of the IP-header represented in multiples of 32-bit wide words.
TOS	$1 * 1$	Type Of Service field, can be used for Quality of Service Operations.
Length	$1 * 2$	Indicates the payload length of an IP datagram.
Fragmentation & Offset	$1 * 2$	Used to show and determine if a datagram has been split into smaller parts.
TTL	$1 * 1$	The Time To Live field shows how many hops a packet may take before it is discarded. A hop in an IP network is defined as a node which decides upon a packet's further transmission by examining its IP-header. This can be a router for example.
Protocol	$2 * 1$	Indicates which type of upper-layer protocol, like Transmission Control Protocol (TCP) or UDP is used. This field is also needed by the modules Sender Control and Filter.
Check-sum	$1 * 2$	Includes an error detecting code calculated over the IP-header. The payload is not included.
Source Address	$2 * 4$	The source address is needed by the units Sender Control and Filter.
Destination Address	$2 * 4$	This field is needed by both Sender Control and Filter.
Options	$\leq 1 * 40$	This field is optional and needs up to 40 octets of memory.
Sum	Max 66	The amount of octets needed to store the configuration data for the IPv4 protocol.

UDP Protocol

The UDP protocol header consists of the fields Source and Destination Port, Length, and Checksum [99] shown in Table 3.6. The Table consists of the same fields with the same meanings as Table 3.4. However, because a sender's destination port is a receiver's source port and vice versa, the amount of needed memory can be reduced by 4 octets. This leads to a memory reduction of 33%.

Table 3.6: UDP Protocol Costs

Field	# of Octets	Comment
Source Port	$2 * 2$	Needed by the sub-modules Sender Control and Filter. The Source Address of the Sender is the Destination Address of the Filter and vice versa.
Destination Port	$2 * 2$	Needed by the sub-modules Sender Control and Filter. The Destination Address of the Sender is the Source Address of the Filter and vice versa.
Length	$1 * 2$	Indicates the length of the whole UDP packet the payload.
Check-sum	$1 * 2$	Includes an error detecting code calculated over the IP-header, the UDP-header and the payload.
Sum	12	The amount of octets needed to store the configuration data for the UDP protocol.

RTP Protocol

The RTP-Protocol header consists of a fixed and a variable part like the IPv4 header. The variable part is mainly introduced by a Mixer [87]. The according header fields, as well as the memory costs for the interface of the RTP-Engine are shown in Table 3.7. These fields are not described in detail, as this has already been done in Section 2.2.2. Therefore, the comment Section is omitted in the table.

Table 3.7: RTP Protocol Costs

Field	Version & Padding & eXtenson & Contributing Source (CSRC) count	Marker & Payload Type	Sequence Number
# of Octets	$1 * 1$	$1 * 1$	$1 * 2$
Field	Timestamp	SSRC	CSRC
# of Octets	$1 * 4$	$2 * 4$	$\leq 1 * 120$
Field	Extension		
# of Octets	$\leq 1 * 65536$		

Since the SSRC has to have the same value as the sender and the receiver participating in a RTP session, it is possible to reduce the interface costs by 4 octets. Moreover, the CSRC field can be omitted because these information are provided by Mixers only. Thus no information must be provided to the module Sender Control in this case. Considering the Filter module it must be noted that only the SSRC is required to associate RTP-packets to an ongoing RTP-session. However, the CSRCs provided by incoming packet are needed by the Real-time Control Protocol (RTCP) protocol in order to transmit receiver reports to these contributing sources [87]. This issue can be solved by forwarding the first RTP packet of a session to the RTCP stack

implemented in software. The RTCP stack would have to register the contributing sources and now can transmit receiver reports to the Mixer, which forwards them to the contributing senders eventually. This reduces the interface costs by 120 octets. Moreover, the "Extension" field can be discarded because it is mostly used in experimental implementations testing new payload format independent functions [87]. This will lead to another 65536 octets freed. Therefore, the memory consumption of the RTP-Engine's interface, considering the RTP packet header, can be reduced by greater than 99%.

The examination of the RTP-protocol header concludes the discussion of the external interface to the RTP-Engine. The total interface costs are summed up in Table 3.8.

Table 3.8: RTP-Engine Interface Total Costs

Protocol	# of Octets
Ethernet	13
IPv4	20
UDP	8
RTP	12
Total	53

GPNIF Interface

In order to transmit and receive Ethernet frames with the MAC hard core, an appropriate interface has to consist of the following parts:

Receive Frame Memory This memory will be used to store a whole received Ethernet frame. The maximum packet length of an Ethernet frame leads to a memory size of 1500 octets [89]

Transmit Frame Memory The transmit buffer needs to be 1500 octets large as well, because only whole frames can be transmitted by Ethernet MACs. Thus, if a smaller Transmit Frame Memory is used, but the largest possible frame is to be sent the interception of the send process will lead to a timing violation of the Ethernet protocol, because the Ethernet standard does not support postponement of ongoing frame transmissions.

Control Data This part of the interface represents several status and functionality bits. They can be subsumed in a single 32 bit register together with the control and status bits of the RTP-Engine.

The whole GPNIF interface requires 3000 octets. The description of the NIC-Wrapper's external interfaces to the AC97 audio codec and the Ethernet MAC were already introduced in Section 3.5.2, therefore they are omitted here.

3.6.2 Internal Interfaces

In the previous Section the external interfaces of the NIC-Wrapper were introduced. This Section will present the internal interfaces of the IP-Core. In Figure 3.7 it can be seen that the internal interfaces are located between the Ethernet MAC Interface, the Arbiter, the RTP-Engine, and the GPNIF. Therefore, a common interface should be deployed for a better extensibility and maintainability. The RTP-Engine uses the Local Link interface described in Section 3.5 to interface with the MAC IP-Core. Thus, the LLI would be an appropriate interface (see Section 3.5.2). So in order to use the LLI two more components are necessary. On the one hand, an arbiter is needed to control between the access to the Ethernet MAC Controller. On the other hand, a GPNIF to LLI translator is needed. This component is used to translate the LLI read and write commands into GPNIF conform ones and vice versa. Due to the fact that the GPNIF should be independent from the rest of the NIC-Wrapper this translator is specified.

Arbiter

The Arbiter's task is to control the access to the MAC core of the RTP-Engine and the GPNIF. This is not done arbitrarily, but follows a priority scheme. The Arbiter consists of three interfaces, one exported to the RTP-Engine, one to the GPNIF, and one fed into the Ethernet MAC Interface. It's behavior can be modeled as a state machine, which is shown in Figure 3.8.

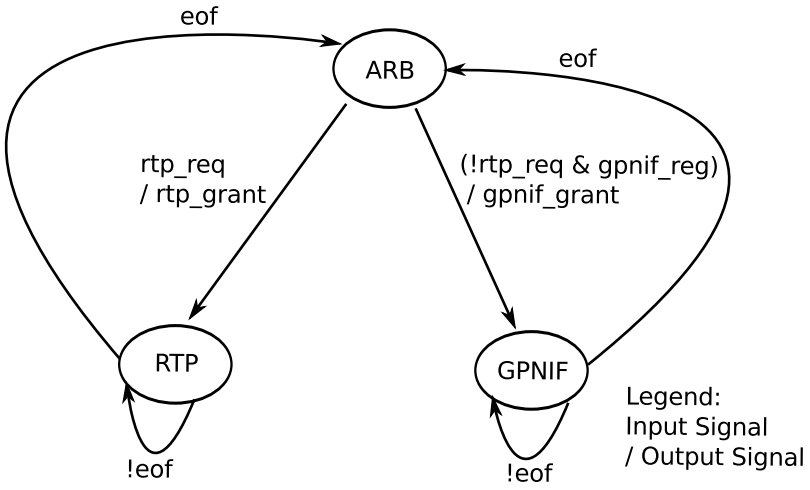


Figure 3.8: Arbiter Finite State Machine

The state-machine contains three states. In the ARB state, which is the reset state, the arbiter awaits requests from either the RTP-Engine, or the GPNIF. If the GPNIF issues its request ("gpnif_req") first, then it is granted ("gpnif_grant") and the automaton advances to the state GPNIF. In the case the RTP-Engine issues its request ("rtp_req") first, it is granted ("rtp_grant") and the next state taken is RTP. In both cases a whole frame is transmitted regardless of any further requests of either GPNIF or RTP-Engine until ("eof") is issued. After the end of the transmission the automaton returns to state ARB again. However, if the RTP-Engine and the GPNIF issue requests ("gpnif_req", "gpnif_req") at the same time, the RTP-Engine's request is granted ("rtp_grant") reflecting the need for isochronous behavior in a RTP-session. This implies that the transmission of GPNIF's Ethernet frame is postponed by the transmission time of the RTP-frame. Although it is theoretically possible that the GPNIF starves because every time a

RTP-packet is sent, the RTP-Engine issues another send request immediately, this cannot occur if the RTP-Engine is configured to follow RFC-3550. In RFC-3550 it is stated that an RTP-packet usually contains 20 milliseconds of speech, thus, this being the send interval for RTP-packets. However, it is not possible to send packets which are shorter than the highest sampling rate in the used codec. This also applies if only one encoded speech sample is sent in a packet.

Translator

This subsection introduces the Translator, a module performing a conversion between memory read/write commands and the Local Link Interface used throughout the NIC-Wrapper. Considering the Translator it can be said that the PPC405 processor chosen in Section 3.4.1 uses a peripheral bus which is either 32, 64, or 128 bits wide. However, the LLI interface on the Ethernet MAC's and on the RTP-Engine's side is specified to be 8 bits wide. Therefore, the translator is not only required to map read/write commands but also to provide a serializer/deserializer to translate between these two bit widths.

Furthermore, it is necessary that, since Ethernet frames are not 32, 64 or 128 bit aligned, the translator is able to cope with single byte aligned frames. The same is true for the converse direction. Therefore, a frame received, discarded by the Filter module of the RTP-Engine, which has to be stored in the receiver memory of the GPNIF needs to be parallized to the native word width needed in the memory. Basically, both modules, the serializer and deserializer work in the same way. First, a "start" request to start a serialization/deserialization of a packet is issued. Until a "stop" request is issued, the incoming octets are deserialized/serialzed. Therefore, they are stored in/read from words of 4, 8, or 16 octets per word. However, if "stop" is recognized the process terminates immediately regardless of how many octets are unread/unwritten in a word. In order to keep track of how many octets are actually serialized/deserialized, an appropriate counter is kept.

3.7 Partitioning of the RTP-Engine Interface

The previous Sections 3.3-3.6 introduced the bases on designing a VOIP stack in an FPGA. A series of options and decisions regarding the decisions on processors, thus, bus-systems, operating software and designing hardware interfaces were presented. This Section will take these information and conclude upon the hardware/software partitioning of the VOIP stack. It will be shown how the communication frequency of several components inside a VOIP stack influences the decision on which parts are implemented in hardware and which in software. To be more precise, which parts of the software components will be implemented in kernel space, and which in user space, due to the already done decisions upon the Base-System shown in Section 3.4. The partitioning of the VOIP stack is presented from a protocol's, and thus interface's point of view, in contrast to a hardware and software centered view. This is done because of the need for a more interface centered hardware/software co-design, as mentioned in Section 3.6. Therefore the remainder of this Section is split into the following parts: In part one, the data exchanged between the RTP stack and the RTCP stack are examined. Furthermore, the results are used to change the interfaces presented so far. In part two, possible location, kernel space, or user space for the implementation of the RTCP stack are examined. Moreover, the signaling protocols in the partitioning are defined.

3.7.1 Refining the Hardware Interface

In order to refine the interface between the RTP and the RTCP stack, the performance crucial parts have to be detected. Considering the transceiving of RTCP packets the following information is needed from a RTP stack implementation [87]:

SSRC The SSRC is needed by a receiver report of the RTCP stack. It must be retrieved every time it changes. Because the SSRC of a session changes usually only at the beginning of a session from unknown to a dedicated value, this information has no great impact on the refinement of the hardware/software interface. However, when the first SSRC is received there are generally two possibilities to feed it back to the RTP-Engine's Filter in order to configure it properly. On the one hand, the whole packet could be transmitted to user space, parsed, and the retrieved SSRC can be written back to the Filter. On the other hand the RTP-Engine may issue that an unknown SSRC has been detected and the software driver reads this information back and configures the Filter appropriately.

Sender's Packet Count The sender's packet count is to be read from the RTP-stack every time a RTCP Sender Report has to be sent. According to RFC-3550, this has to be done in a five second interval in the worst case, thus putting no significant pressure on the RTP-Engine's interface. However, it must be included into the RTP-Engine's interface, adding another 4 byte of interface costs.

Sender's Octet Count It has to be read from the RTP-stack every time a RTCP Sender Report has to be sent. However, it can be calculated from the amount of sent packets, because the amount of samples stored in a packet has to be known a priori due to RTCP-stack initialization. Therefore, it must not be included in the RTP-Engine's interface.

Jitter Because the jitter value has to be calculated as a running average value it must be updated every time an RTP packet is received. This would increase the stress on the RTP-Engine's interface because of the rising interrupt frequency and the therefore resulting increase in bus communication. Therefore, the jitter calculation may be implemented in hardware to minimize the hardware/software interaction of the RTP-Engine as much as possible.

A more detailed description of the proposed refinements is shown in Chapter 4. Moreover, the software driver controlling the RTP-Engine and the GPNIF can be implemented as a single NIC-Wrapper driver, which would lead to a better maintainability.

3.7.2 Software Partitioning

The RTCP stack can be implemented in kernel or in user space. A motivation for kernel space is, that if the RTP-stack is implemented in hardware, no mode switches would be necessary when a VOIP session is operated. The kernel module would only be accessed by the signaling protocol stack at the beginning and the end of a VOIP session. Such an approach has been proposed by Guk Sun and Jo Kim [SK05]. The authors propose a kernel level implementation of a whole RTP/RTCP stack for a Linux 2.6 kernel. Both protocols are implemented in separate modules which can be interfaced by system calls from the user space. Therefore, the authors implemented a number of new system calls into the Linux kernel API. However, since system calls can be used to notify kernel space from the user space, but not vice versa, another mechanism was proposed

by Guk Sun and Jo Kim. In their approach a kernelmodule should issue so called signals[6] from user space to kernelspace, to notify a user application on the occurrence of an event. Sun and Kim argued that it is necessary to implement special signal queues, so that no soft-interrupt is lost[7] inside the Linux Kernel. However, Kim and Sun did not realize that at the time of writing, in 2005, a feature called real-time signals has already been introduced in the Linux kernel in 2004 [65]. In the case of real-time signals also the absolute order of the signals issued is guaranteed. Moreover, real-time signals already implement a queuing mechanism, therefore obsoleting Kim's and Sun's solution. However, a major downside of this method, regardless of using real-time signals or not, is that a kernel level implementation of either protocol including a system call introduction, results in a large kernel dependency. Thus, if a new kernel version shall be used the whole code has to be ported to it.

An alternative solution would be to implement the RTCP-stack in user space and call the RTP-Engine's driver from there. This approach would be independent of the used kernel version. Considering the signaling protocol a user space implementation is the best suited way. The same reasons for implementing the RTCP protocol in user space apply here.

3.8 Design Decision Summary

This Chapter introduced the Base-System and NIC-Wrapper interface design. As it is mentioned in Table 3.8 about 53 octets of memory are needed by the RTP-Engine interface to provide a whole configuration vector to the RTP-Engine. The hardware/software partitioning of a VOIP-phone designed for the use in a PC is presented in Figure 3.1.

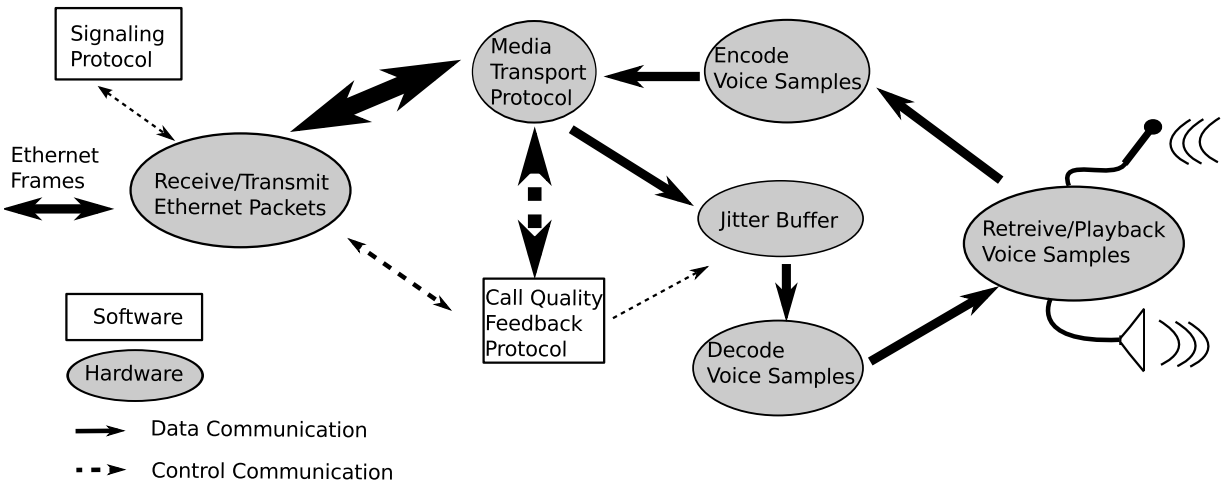


Figure 3.9: Communication Frequency & Partitioning VOIP Stack

Compared to a VOIP-phone implementing the RTP-Engine the hardware/software partitioning changed significantly as it can be seen in Figure 3.9. The arrows shown in this figure represent the communication frequency between the corresponding parts of the stack. The thicker an arrow is, the higher is the communication frequency and therefore workload for a general purpose processor

[6] Sometimes also called Soft(ware) Interrupt

[7] Actually, signals cannot get lost inside the Kernel, but can be overwritten by another signal issued before the previous signal is handled, because no buffering exists [65].

and hardware/software interfaces. An overview on the made decisions in this Chapter is shown in Figure 3.10.

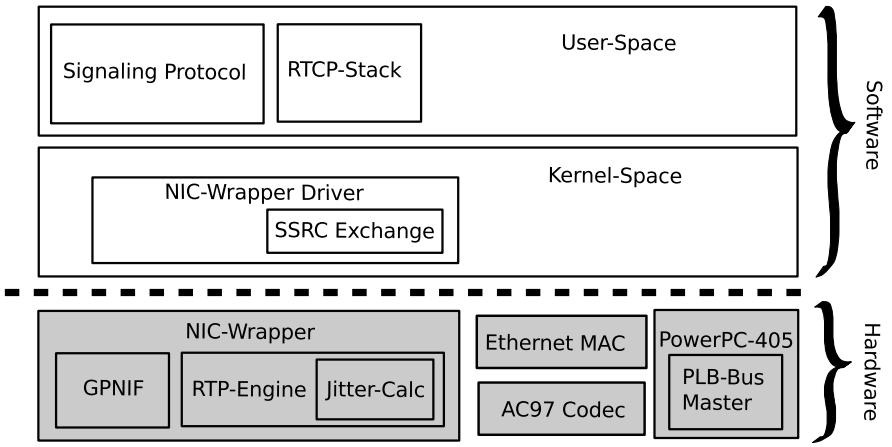


Figure 3.10: Detailed HW/SW Partitioning VOIP Stack

As it was mentioned in the previous Sections, the Base System consists of the Power-PC405 hard-macro providing a PLB bus master. Furthermore, an embedded Linux distribution with U-Boot as bootloader, and the Open Firmware Flattened Device Tree as board configuration data structure is used. Moreover, the jitter calculation is processed by the RTP-Engine in hardware. The SSRC change sequence upon detection of a new SSRC in a RTP-session is performed in kernel space by the RTP-Engine's driver. The call quality feedback protocol RTCP is implemented in user space because of portability reasons, as well es the signaling protocol stack.

4 Implementing a VOIP Stack in an FPGA

In Chapter 3, a platform design for a VOIP stack implementation in an Field Programmable Gate Array (FPGA) has been introduced. Several implementation options have been evaluated. In the end, a Base-System consisting of a Power-PC 405 hard-core providing a Processor Local Bus (PLB) interface and an Embedded Linux distribution started by the bootloader U-Boot have been chosen. Furthermore, the architecture and interface of the NIC-Wrapper providing a unified interface to the GPNIF as well as the RTP-Engine has been introduced. Moreover, Chapter 3 provided a high-level introduction on the just mentioned components. In this Chapter, the actual implementation of the whole system is described in detail. Thus the Chapter is divided into the following subsections. Section 4.1 introduces the implementation of the Base-System in greater detail. In addition the used Embedded Linux distribution is described in this Section. In Section 4.2 the NIC-Wrapper's interface is introduced. Finally, the softphone implementation, actually operating the RTP-Engine is presented in section 4.3. Figure 4.1 provides an overview on the whole VOIP stack implementation including the principle structure of the softphone. The results of Chapter 3 show that the GPNIF and RTP-Engine submodule are both located inside the NIC-Wrapper module, which provides a uniform interface to the Linux kernel. The Linux kernel houses the NIC-Wrapper device driver which consists of the GPNIF related driver functions and the RTP-Engine interface related IOCTL functions. Most of the software components, including the RTCP stack implementing the user space interface of the RTP-Engine interface, are realized in user space (see Figure 4.1). Moreover, it can be seen that Busybox is used as an embedded Linux distribution.

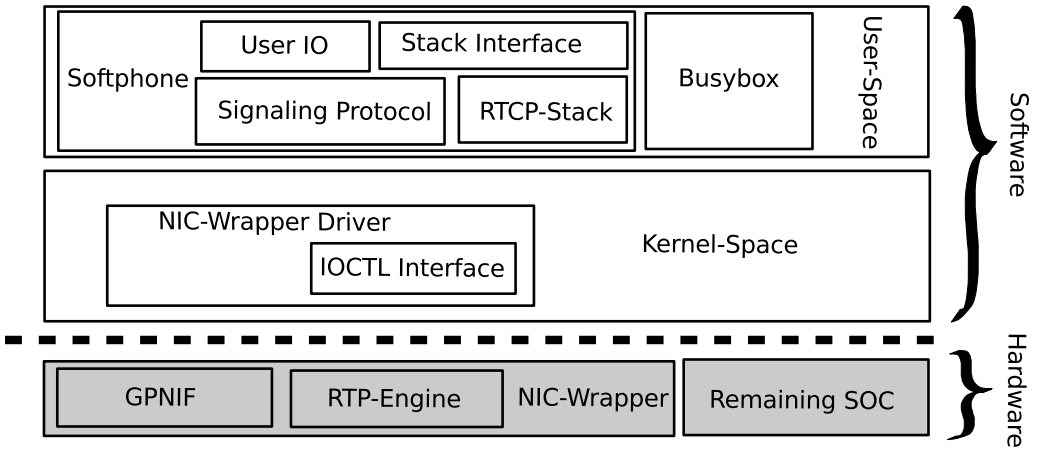


Figure 4.1: NIC-Wrapper Device Driver Architecture

4.1 Base-System Implementation

In Section 3.4.1, the Power-PC 405 general purpose processor providing a PLB bus master has been chosen as the main processor for the Base-System. Furthermore, an Embedded Linux was the operating system of choice. This section shows, how these components are integrated into the Base-System. First, the hardware components of the Base-System are described. In the next step, the configuration and operation of U-Boot and Linux on the ML405 is shown.

4.1.1 System On Chip Description

In order to provide a feasible operational environment for the RTP-Engine, the following additional components are needed:

Main Memory Controller This module provides access to the SSRAM (9 Mbit) and SDRAM (128 MByte) off-chip memories available on the ML405 development board.

UART Interface The Universal Asynchronous Receiver Transmitter (UART) Interface is used to export a command shell provided by the Embedded Linux distribution to the user. Thus, it serves as standard input/output interface.

Interrupt Vector Controller The Power-PC 405 processor's interrupt controller is capable of serving several internal interrupt sources, like timer, exception and debug interrupts, as well as one external interrupt source [104]. Therefore, an external interrupt controller is used to capture interrupts from peripheral devices.

Initial Boot Memory Although not needed if the primary boot and configuration media is the CF-Disk, this about 4 kByte large on-chip memory block is used to run peripheral testing software in the development phase. It is large enough to contain a whole test application for the NIC-Wrapper.

CF-Disk Controller Interface The CF-Disk serves as a primary boot and configuration media. Furthermore, it is used as hard disk by the Embedded Linux distribution. It exports an interrupt source to the Interrupt Vector Controller.

EEPROM Controller Interface The Electrically Erasable Programmable Read-Only Memory (EEPROM) is used to store persistent bootloader configuration data, which cannot be provided by the archive stored on the CF-Disk at the time of running the bootloader.

All these components are provided by the Xilinx ISE development environment [78]. This software package provides a database of several IP-Cores which can be integrated in user specified designs. ISE serves as a front-end for the Design-Flow[1] needed to generate a bit stream to configure the FPGA. In order to implement the Base-System as a System on Chip, Xilinx Embedded Development Kit (EDK), which is a part of the Xilinx ISE, is used. In contrast to ISE, EDK provides support in designing System-on-Chip platforms. It provides a graphic system representation and

[1] An FPGA's design flow covers design-elaboration (Compiling the design's Hardware Description Language (HDL) model description into an intermediate, chip independent, but vendor dependent, representation), design-synthesis (Compiling the intermediate representation into an FPGA specific library), and partition place and route (Chip specific components are mapped onto the chip, their interconnect is created and the static timing is computed).

aids in interconnecting the system's components. Furthermore, the memory map, and the open firmware flattened device tree are generated here.

The architecture of the Base-System including the NIC-Wrapper can be seen in Figure 4.2. It shows the whole system inside the Virtex4 FPGA. All off-chip components are omitted. Only the interaction of certain peripherals with FPGA IOs are shown by black arrow symbols (↔) touching the borders of the symbolic FPGA. The main components in Figure 4.2 are presented in the following:

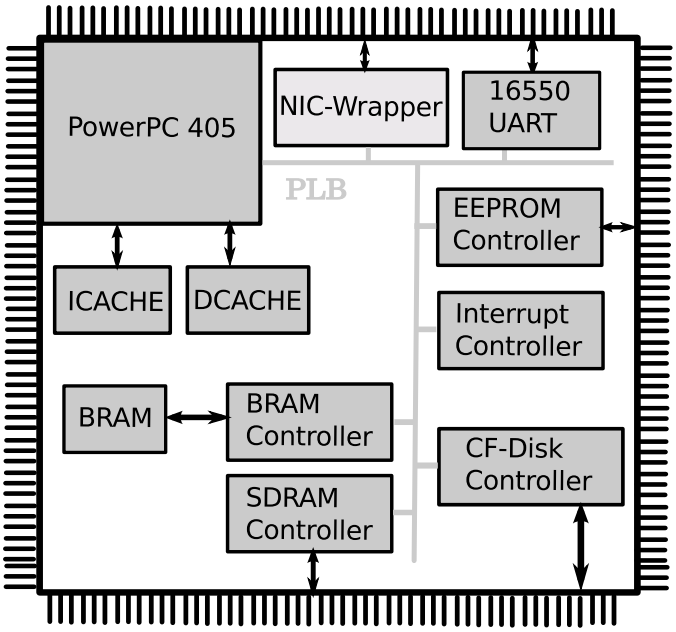


Figure 4.2: SOC Architecture

The Power-PC 405 hard core utilising on chip memory Block-Ram as instruction (ICACHE) and data (DCACHE) caches. Each cache is 16 kByte large. Although the caches are optional when the Power-PC processor wrapper is configured inside Xilinx EDK they are taken into account because of a study realized by Ben Salem et al. [AKBSS10]. In their study the authors examined, amongst others, the reaction time needed by a PI-control unit (proportional plus integral) to control a direct current (DC)-motor implemented in software. The control task was implemented as a standalone application on a Virtex4 FPGA using a Power-PC-405 processor. In one series of runs the Power-PC was configured with instruction and data caches. In another series of runs the Central Processing Unit (CPU) was configured without caches. Moreover, the PLB was used to interface with the main memory. The experiment showed, that if caches are omitted, the system's reaction time rises from 12 to 14 μs to about 85 μs for the speed controller task. And from 16 μs to about 114 μs for the current controller task. This is a factor of ~ 6 for the speed controller task, and a factor of ~ 7 for the current controller task. Although the examined tasks were hard real-time tasks in contrast to the time constrained tasks[2] in the VOIP stack, it can be seen that the use of caches decrease a system's reaction time significantly. Therefore, instruction and data caches are deployed in the Base-System as well.

[2]Setting new SSRCs, retrieving statistic data from the RTP-Engine, for example.

As it can be seen in Figure 4.2 all peripherals are attached to the PLB bus. This is done because the On-chip Peripheral Bus (OPB)[3] support will be discontinued in future releases of Xilinx ISE [78]. This is possible, because most peripherals available in ISE exist in a version providing an OPB and a PLB interface.

4.1.2 Processor Local Bus Intellectual Property Interface

A PLB bus master is provided by the Power-PC processor [107]. Nevertheless, it is possible to have more than one bus master in a PLB comprehending system. However, a bus arbiter must be utilized to control a master's bus access in this case. Considering the Base-System only one bus-master, the Power-PC 405 processor, is used. All other devices are so called PLB slaves. Each slave device is attached to the bus via the Intellectual Property Interface (IPIF). The IPIF provides an abstraction layer to the functional implementation of the peripheral. Therefore, the IPIF is the interface between the "Bus Interface" and the "Constraint Schedulable Code" (see Section 3.6 and Figure 3.6(b)). This abstraction layer provides several services which can be seen in Figure 4.3 [102]:

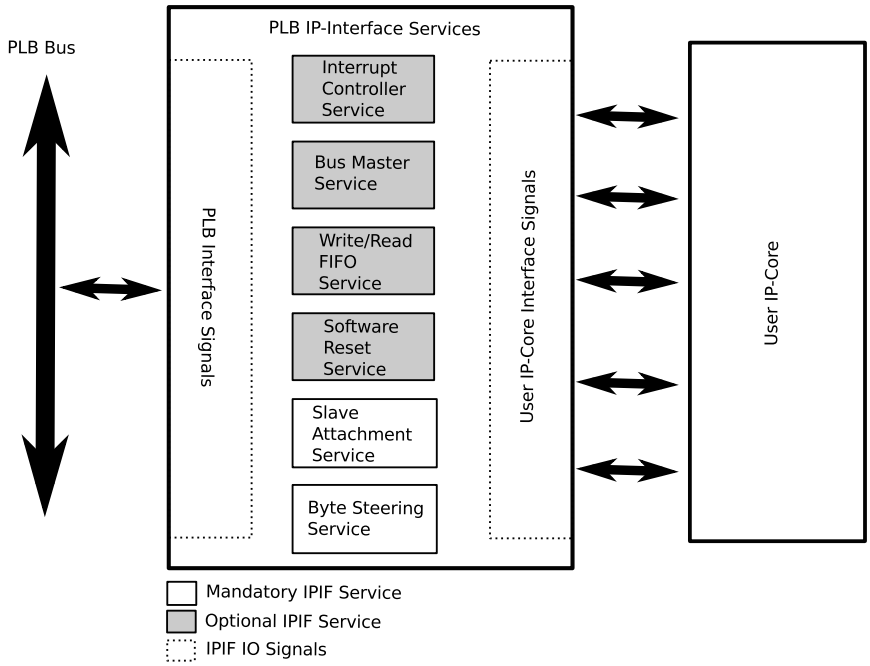


Figure 4.3: PLB - IPIF Services

Bus Master Service This function block is optional. If it is used the Intellectual Property Core (IP-Core) is able to operate as a bus master. Therefore, a bus arbiter is provided with this module.

Write/Read First In First Out (FIFO) Service The FIFO submodules provide buffer memories which can be configured in size and word width. Moreover, it can be determined if the

[3] In contrast to the PLB, the OPB is a low speed bus, connecting peripherals with infrequent CPU/peripheral demands. Therefore, it is often used to connect UART, or EEPROM modules to the processor.

memories shall be implemented using Block-Ram or FPGA Look-Up-Table (LUT)s[4]. The FIFO Service is optional.

Interrupt Controller Service The interrupt controller service provides mechanisms to capture and accommodate several interrupt sources of the peripheral. It acts as a unified interface to a systems vector interrupt controller. It is able to react upon edge and level triggered interrupts. It is optional.

Software Reset Service The SRS can trigger a synchronous reset of the peripheral if a specified value is written to it. This service is optional.

Slave Attachment Service The SAS is a mandatory service and provides address resolution between the PLB and the peripheral device.

Byte Steering Service This mandatory service provides data width conversion mechanisms between the PLB and the peripheral device.

Most peripheral devices in the Base-System facilitate at least the Interrupt Controller as additional IPIF service. Beside these services, there exist also several types of peripheral device templates. They provide example implementations to access an IPIF attached peripheral via registers, FIFOs, or Block-Ram. Furthermore, these examples influence the way a peripheral's configuration registers, memories and FIFOs are exported to the system's memory map. This means that a device using the Block-Ram template has a different offset from the device's assigned memory base-address than a device using the register or FIFO template. This is a result of the fact that a device may utilise more than one of these templates to export its "Constraint Schedulable Code" section to the IPIF. See the PLB-IPIF manual [102] for a detailed description. However, these templates are not part of the PLB specification, but are provided by Xilinx to ease the creation of an appropriate interface for a peripheral device.

The remaining peripheral devices, except for the NIC-Wrapper, which is described in Section 4.2 shown in Figure 4.2, are:

SDRAM Controller The two 16 bit wide 64 MB off-chip SDRAM memory banks which are accessed as a 32 bit wide 128 MB memory bank are used as main memory by the Base-System. The memory controller does not facilitate an interrupt controller. However, buffering FIFOs are used.

16550 compatible UART There are three types of UART modules available to serve as a primary user interface. The UartLite module which can be statically tied to a specific frame format and baudrate, the 16550, and 16450 full configurable UART modules, also featuring a modem interface. Although the UartLite uses the least resources, the 16550 compatible module has been chosen because it is natively supported by the Linux kernel. The module uses the interrupt service of the IPIF.

Interrupt Vector Controller This module is available in one version only. However, the priority of the attached interrupt service enabled peripheral devices can be configured statically.

[4]The smallest general purpose unit inside an FPGA is a Logic Block, or Logic Cell. It consists of a LUT holding a combinatorial function, an optional register, and a multiplexer. All these components are configured by the bit-stream resulting in the specified hardware implementation [FK09, p.233f].

Initial Boot Memory The initial boot memory is implemented as an IPIF module including the Block-Ram template.

CF-Disk Controller & EEPROM Controller These two peripherals are implemented as IPIF modules using the interrupt service module.

4.1.3 Operating System & Bootloader Description

This section is divided into two parts. In the first one, the utilised bootloader is introduced. In the second one, the Embedded Linux distribution and its build system is described. As already mentioned in Section 3.4.4 U-Boot [74] is used as a bootloader for the Base-System. It has been developed by Wolfgang Denx and is based upon the bootloader project PPCBoot [70], which is discontinued since 2002. U-Boot is designed to be extensible, therefore providing a core application and several optional functions. The optional functions can be configured in a board specific configuration file. This file is used as a central source of information by U-Boots build system to compile the appropriate U-Boot distribution. Furthermore, the central configuration file needs to provide information about the target platform's memory map to U-Boot. The file containing the Base-System's memory map can be retrieved from Xilinx EDK after the Base-System has been synthesized and the appropriate software support libraries have been built. However, in order to operate U-Boot on the ML405 development board utilising the CF-Disk boot method, an adapted version of U-Boot is needed. This is necessary because U-Boot is intentionally designed to run from Flash memory, but the CF-Disk boot method loads U-Boot directly into RAM. This adapted version can be retrieved from the Xilinx software repository [80]. As already mentioned in Section 4.1.1 the bootloader's configuration information consisting of the following parts is stored in the Base-System's EEPROM.

- An entry in U-Boots environment pointing to the location on the CF-Disk where the flattened open firmware device tree is located.
- A macro loading the flattened device tree from the CF-Disk to the main memory.
- An entry pointing to the location of the Linux kernel. The kernel has to be available in a special format called uImage.
- A macro copying the kernel image from the CF-Disk to the Base-System's RAM.
- A set of macros loading the device tree blob[5] and the Linux kernel as well as run the latter.

Considering the Embedded Linux distribution Busybox and its build environment Buildroot is used. Busybox is a lightweight multi-call binary Linux distribution [Wel00]. It provides size optimized re-implementations of many services, tools and software packages found in a regular Linux distribution. The size optimization is achieved by only implementing the parts of a software-package often used by users. This results in a smaller variety of available options for certain tools. The term multi-call reflects the fact that there exists only one binary that has to be called in order to execute a single program available with Busybox. This can be achieved by using symbolic links, instead of real executables, in the appropriate directories[6] of a Linux distribution. Nevertheless,

[5] The compiled form of the flattened Open-Firmware Device-Tree is called Blob [GL08].

[6] Usually, the directories */bin, /usr/bin, /usr/local/bin, /sbin*, and */usr/sbin* are used to house the installed software packages executables [54].

it is also possible to use third party software within a *Busybox* distribution. This is the task of Buildroot, which is a set of makefiles and patches providing the following functions [48]:

- Building a cross compiler toolchain.
- Building an Embedded Linux distribution.
- Handling its root file-system.
- Configuring and building an appropriate bootloader and kernel image.

The tool itself does provide services to download and build the needed software packages for a fully customized Embedded Linux distribution. Although it also facilitates services to build a bootloader and a Linux kernel, these features are not used in this thesis. Instead the software packages own build systems are utilised to build these components. However, the cross compiler toolchain provided by Buildroot is used to do so. In order to reduce the Base-Systems software memory footprint further, *uclibc* is used as a standard C library implementation [Sal08]. The advantage of *uclibc* in comparison to a *glibc* implementation is that *uclibc* is especially designed for a low memory usage. Furthermore, *uclibc* is directly supported by Buildroot.

However, there exist also other cross compiler toolchain generation tools providing similar services than Buildroot:

Bitbake Bitbake derives from the Gentoo[7] package management system portage. It consists of a set of so called recipes providing the information needed to build a software package. It can be configured to build an appropriate embedded Linux distribution [47].

Crosstool This software package provides a set of scripts and makefiles suited to build cross compiler toolchains. However, the building of whole distributions is not supported. Nevertheless, it features the creation of so called build clusters[8] in order to speed up the compilation of large software projects [50].

ELDK The Embedded Development Kit provides several pre-compiled cross-development tools targeting Power-PC processors. It is primarily focused on the development of standalone applications [52].

Openembedded It relies on Bitbake and provides configuration services to use Bitbake to generate Embedded Linux distributions in a convenient way [67].

LFS The project Linux From Scratch does not provide a ready to use build environment, but a complete set of manuals to build one [63]. This includes manuals for building cross compiler toolchains and fully customized Linux distributions.

Emdebian Emdebian is the embedded version of the Debian Linux distribution. It features a set of pre-compiled cross compiler toolchains and a variety of packages ready to be integrated into the Embdedian installer which creates the target root file-system. This makes it possible to easily upgrade an Emdebian installation on a target [53].

[7] A Linux distribution available as source code mainly. Gentoo provides its package management and build system *portage*, which compiles and installs each package from source code, giving the user a great level of flexibility when building her system.

[8] Distributed compiling on several distinct computers

The list just presented is not complete, but gives an overview on certain available tools supporting the creation of embedding Linux distributions. Because of the native support of the *uclibc* and its extensibility Buildroot in version 2009.11 has been chosen over the other mentioned approaches to build an embedded Linux distribution.

Considering the Linux kernel, a Xilinx patched version with support for the ML405 development board has been used [79]. This Linux kernel supports all vendor provided peripherals. It is used in version 2.6.34. The Kernel is compiled without module support and with a strong focus on the available hardware and required software subsystems. Thus, resulting in an application specific kernel. The module support is omitted to ease the kernel update process on the target system, as well as to reduce the update cycle times during development phase. This is useful because the kernel can be exchanged faster, if only one file, the kernel image, has to be copied to the CF-Disk.

4.2 NIC-Wrapper Interface

By now the Base-Systems hardware and software components have been introduced. This section introduces the NIC-Wrapper interface from a software and a hardware point of view. The interface itself is divided into a hardware part described in Section 4.2.1 and a software part introduced in Section 4.2.2.

4.2.1 Hardware Interface Implementation

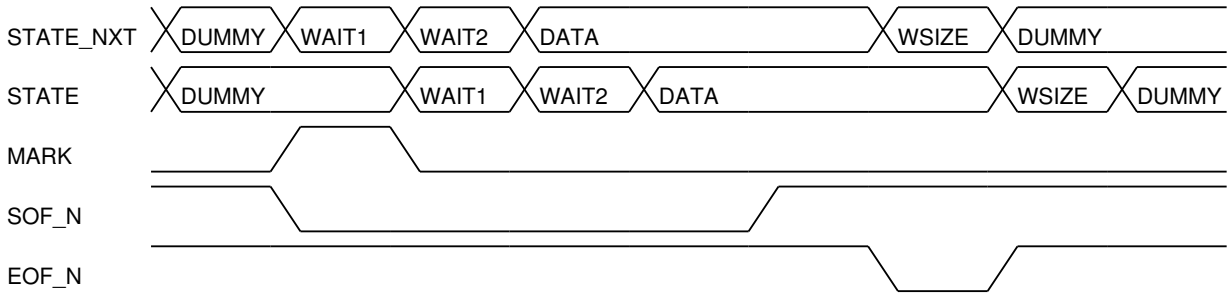
The hardware interface of the NIC-Wrapper consists of three parts: The GPNIF interface providing access to the Ethernet Media Access Control (MAC) hard macro; The RTP-Engine interface exporting the configuration interface of the RTP-Engine; And finally a common interface providing the access to certain configuration parameters not directly associated with either GPNIF or RTP-Engine. As already described in Section 4.1.1, each module attached to the Base-System's PLB is a slave device. In the case of the NIC-Wrapper the following optional services are activated in its PLB-IPIF:

The **interrupt controller service** is enabled to gather the NIC-Wrapper's interrupts and provide them to the Base-System's interrupt vector controller. However, the optionally available interrupt priority encoder and device Interrupt Service Controller (ISC) are omitted. These services have to be emulated in software. The purpose of the priority encoder in conjunction with the ISC are intended to reduce the administrative interrupt dependent traffic on IP-Cores. Because there are only two interrupt sources provided by the NIC-Wrapper, these modules are not necessary. This module is used by the GPNIF solely.

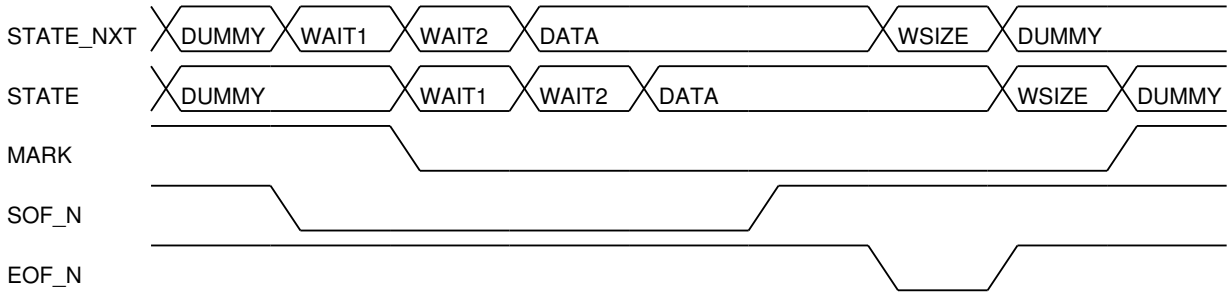
Moreover, both **FIFO services**, read and write, are utilised by the IP-Core. On the one hand, the Read-FIFO is used to store Ethernet frames which do not contain any RTP packets associated with an ongoing VOIP session. On the other hand the Write-FIFO stores the Ethernet frames to be sent. In the case of the Read-FIFO, a special *Packet-Mode* provided by the FIFO is used. Every time an Ethernet frame is received by the Filter (see Section 3.5) it is written to the Read-FIFO of the GPNIF and to the receiver module of the RTP-Engine. When the Filter decides that the frame contains a VOIP session associated RTP-packet, it is discarded from the Read-FIFO. This happens in the following way: Before a new frame is started to be written to the Read-FIFO, the current write pointer position of the FIFO is saved to a register. This step is

called MARK and performed by issuing the MARK signal. It also sets a Packet-Mode enabled FIFO into Packet-Mode. The Read-FIFO can be reset to Normal-Mode by issuing the Release signal. Now the frame can be written in the usual way. However, if a frame shall be discarded and the Read-FIFO is set to Packet-Mode, the signal Restore must be issued. Each of these three commands take two clock cycles to become effective. The Restore command loads the stored location of the FIFO's write pointer, thus deleting the so far written frame. Accordingly, the FIFO's fill indicators are set appropriately. The fill indicators show how many words can be written into the FIFO.

Although, the specification of the Read-FIFO shows that it is possible to assert MARK two cycles before the first word of a frame is written to the FIFO, and thus activating the Packet-Mode, this results in a failure of the FIFO's Packet-Mode operation. The MARK command succeeds, but the Restore command of the FIFO to discard a frame does not. However, if MARK is asserted right from the reset state of the whole system, thus saving the write-pointer in every cycle the FIFO is not demanded to store a new frame, the faulty behavior is cleared out and the FIFO operates as expected. Nevertheless, MARK must be asserted again after a frame has been written to, or discarded from the Read-FIFO to let it operate correctly. This is shown in Figure 4.4(a) and Figure 4.4(b).



(a) MARK assertion according to specification



(b) MARK assertion required to obtain specified behavior

Figure 4.4: MARK assertions of Read-FIFO supporting packet-mode

In Figure 4.4(a) the signal assertion which should lead to the behavior as described in the specification of the Read-FIFO in packet mode can be seen. However, as mentioned above, the restore process of the FIFO's write pointer does not succeed if the FIFO is controlled that way. In Figure 4.4(b) the appropriate assertion of MARK leading to the correct behavior is shown. This is obviously a bug in the FIFO implementation. Moreover, the following signals which can be seen in both figures, are necessary to be explained:

SOF_N/EOF_N These signals origin from the Local Link Interface described in Section 3.6.2.

They are used to indicate the beginning and the end of a frame write sequence.

STATE/STATE_NXT The signals STATE and STATE_NXT indicate the actual state and the next state to be taken in the Read-FIFO controlling state-machine. The state machine consists of the states DUMMY, SOF_S, WAIT1, WAIT2, DATA, and WSIZE. The dummy state waits until the signal SOF_N is set to low, thus starting a frame write cycle. The two wait states postpone the write operation of the first word written to the FIFO by two cycles to let the write pointer save operation take effect. The data state copies the remaining words of the frame. Eventually, the WSIZE state stores the amount of written octets in a FIFO containing only the size information of received frames.

Moreover, the **Register service** is used by the NIC-Wrapper. This service provides an efficient way to directly implement PLB-IPIF accessible configuration and data registers to an IP-Core. The ISE design tool generates an example which shows how the byte-wise addressable 32-bit registers can be accessed. In the case of the NIC-Wrapper the register service belongs to the RTP-Engine. It consists of 17 32-bit wide registers providing a configuration and feedback interface to the RTP-Engine. However, the first register is dedicated to the GPNIF as well. It serves as a common IP-Core configuration and status register, thus being the third part of the NIC-Wrapper's interface. The second register concerns the GPNIF only. For a more detailed description see Appendix A.

The implementation of the RTP-Engine's interface results in a total cost of $16 * 4 = 64$ octets. In comparison to the estimated costs of 53 octets presented in Section 3.6.1. The reason for the cost increase of about 20% is that:

- Three octets are unused, due to word alignment at the beginning of the IP-header section of the RTP-Engine interface.
- Another four octets are used by the SSRC_RX register to store the Filter's configuration data.
- The remaining four octets are occupied by the common configuration register.

Furthermore, another four octets are occupied by the register RTPE_RX_SZR, which is associated with the GPNIF.

So far the single components of the NIC-Wrapper's interface have been described. In Figure 4.5 the architecture of the NIC-Wrapper is shown.

The boxes drawn in a striped shape are implemented from scratch, where the white ones were provided by either Xilinx or Brunmayr [BHS09]. There exist light gray surroundings of certain components, showing that they belong to the GPNIF-transmitter, or receiver side of the MAC hard-core. The abbreviations RD-FIFO and WR-FIFO symbolize the Read- and Write-FIFOs provided by the IPIF. Moreover, the RD-FIFO is operated in packet-mode, as described above. It is shown that the Local Link Interface is deployed between the MAC hard macro, the Filter, the Arbiter, Local Link Interface (LLI) to FIFO interfaces, and the RTP-Engine.

Considering the LLI between the Filter, the LLI to FIFO interface, and the RTP-Engine it must be noted that the interconnection differs significantly from the one described in Section 3.5.2. Usually, a source-ready-out signal of a sender is connected to a source-ready-in signal of a receiver. The same is true for destination-ready signals. However, in the case of the interconnection of

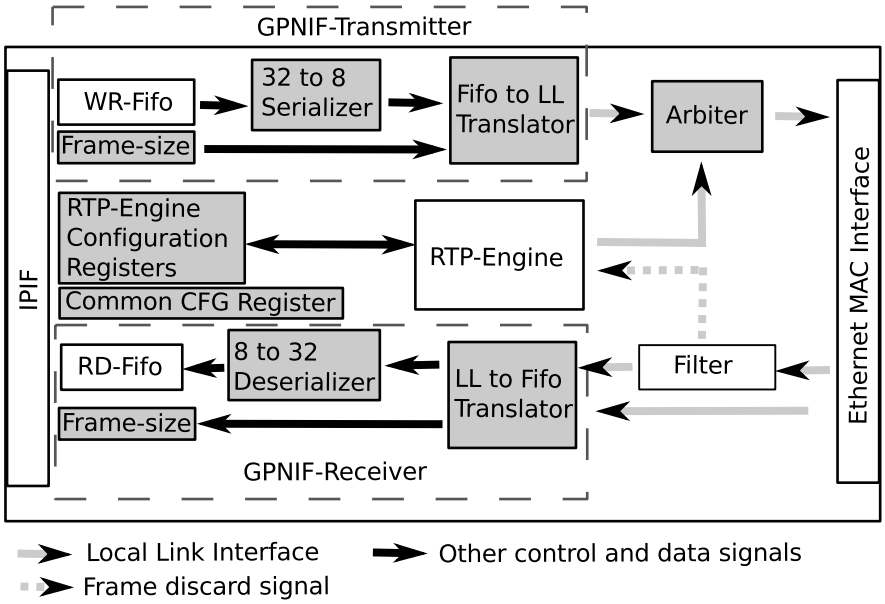


Figure 4.5: NIC-Wrapper IP-Core

Filter, the LLI to FIFO interface, and the RTP-Engine the following interconnection is done to guarantee a correct operation of the LLI. This is needed because the MAC hard macro feeds the Filter and the LLI to FIFO interface simultaneously. In the case the Filter detects a RTP-packet associated with an ongoing session it sends a discard signal to the LLI to FIFO interface. This causes the frame to be deleted from the Read-FIFO. Therefore, the following adaption to the interconnection is required:

A disjunction is performed by the destination-ready-out from the LLI to FIFO interface and destination-ready-out from the RTP-Engine. The results of this equation is fed into the source-ready-in port of the MAC hard macro interface. Moreover, the result is used in a disjunction with the source-ready-in signal of the LLI to FIFO interface. This equation's result is connected to the source-ready-in signal of the RTP-Engine. The remaining signals are connected in the same way as described in Section 3.5.2.

4.2.2 Software Interface Implementation

This section will show, how the software related interface is implemented. Since Linux is used as an operating system kernel, a corresponding kernel module has to be written to interface with the IP-Core. A kernel module consists of a number of functions possibly interfacing with another module, or a user space program through system calls. To make the development of a kernel module more convenient several types of subsystems exist in the kernel, where a module can be associated with. These subsystems provide interfaces to several system calls, like *open(), close(), read(), socket(), ioctl(),* or *write()*. Some of them, like *socket()* are dedicated to a special subsystem, others, like *open()*, can be used more generally. [JC05, p.3ff].

However, a kernel module is developed for a specific hardware component, the question of the suitable subsystem arises. In the case of the NIC-Wrapper it may be possible to divide the module into two distinct modules featuring two subsystems, because of the two distinct interfaces subsumed by the IP-Core. On the one hand, it is quite clear that the networking subsystem will

be used to implement the GPNIF interface driver, because it directly interfaces with the Ethernet MAC hard macro. On the other hand, there are two possibilities to implement the RTP-Engine interfacing modules, namely a *Character Device Driver*, or a network driver related Input/Output Control (IOCTL) implementation.

A kernel module associated with the network subsystem uses the socket interface to communicate with upper layers of the Linux network stack. In the case of the Base-System the interface communicating with the NIC-Wrapper is bound to the PLB bus. In order to interact with the peripheral, the functions *out_be()* and *in_be()* are used. *in_be()* reads a 32-bit wide word from a peripheral, and *out_be()* writes it to the IP-Core, respectively. These functions are actually implemented as macros and are provided by the Xilinx patched kernel.

The **transmission** of an Ethernet-frame using the NIC-Wrapper works in the following way, assuming that the device driver is already loaded and associated with the device: First it is checked whether the MAC hard-core is ready to transmit a new frame by reading the RTPE_CONFR register. If the MAC is not ready, the transmission of the frame is postponed by telling the kernel that the device is busy. If the MAC is ready to send, the next frame is written to the Write-FIFO in a word aligned way, thus respecting the 32-bit word boundary. Finally, the amount of octets to be sent is stored to the high half-word of the RTPE_CONFR register, thus triggering the GPNIF that the frame can be sent.

Considering the **reception** of a received Ethernet-frame the driver has to perform the following steps: When a frame is received by the MAC and is completely written to the Read-FIFO including its size information an, interrupt is raised. The interrupt service routine reads the amount of received octets in a 32-bit word aligned way and stores the actually retrieved amount of octets in a socket buffer. The size information is read from the register RTPE_RX_SZR. The interrupt is triggered by the instant that the RTPE_RX_SZR register is not empty. Furthermore, the socket buffer is checked for holding an RTP-packet with a not known SSRC. If this is the case the Synchronisation Source (SSRC) is extracted and written to the RTPE_RTP_SSRC_RX register to configure the Filter appropriately. If the received packet is not an Real-time Transport Protocol (RTP) packet is sent upstream to the Linux kernel, thus concluding the reception of a packet.

Considering the kernel driver of the RTP-Engine configuration interface, two possible solutions may be implemented as mentioned above. On the one hand, a character device driver providing a byte-wise accessible interface to a device may be used to implement the interface. On the other hand, the IOCTL system call utilising the socket interface may be used to perform the required actions. The advantage of implementing the RTP-Engine's interface as a character device driver is that the RTP-Engine can be controlled bypassing the GPNIF kernel module. This can be useful when debugging the RTP-Engine without utilising the GPNIF. However, a major disadvantage is that two device drivers would try to access a single peripheral device, thus the need to synchronize two kernel modules competing for one resource arises.

This rather complex and not efficient solution can be avoided if the IOCTL is used to let user space applications interface with the RTP-Engine interface. IOCTL provides a generic interface to bypass the regular Input/Output (IO) functions of a driver. It does so by using the drivers subsystem. Therefore, an IOCTL request on a character device driver uses the character device driver subs-system interface and a IOCTL request on a network device uses the socket interface. In the case of the RTP-Engine interface, the socket interface is used. However, in order to let a module identify an IOCTL request four command types have been defined. A command type must be unique throughout the kernel, therefore it has to be registered at the kernel maintainer's

side [JC05, p.535ff]. However, in order to let driver developers test new IOCTL commands there exist 16 user specific command types called *SIOCDEVPRIVATE + 0x0* to *SIOCDEVPRIVATE + 0xf*. In each driver implementing the IOCTL system call a central multiplexing function exists. In this function, the appropriate commands are filtered and the required actions are taken. The data exchanged by the kernel module and the user space application can be exchanged by a C structure. Upon each request data can be written to the driver and read back. The NIC-Wrapper driver provides four commands with the following functionality:

SIOCRTPSETCFG This command writes a new RTP session configuration to the RTP-Engine interface. The supplied configuration vector consists of the complete Ethernet, IP, UPD, RTP header. The header information is generated at the user space application. Furthermore, a flag provided by the structure handed over to the module indicates whether the RTP-Engine shall be started or stopped. No data is read back by *SIOCRTPSETCFG*.

SIOCRTPGETDAT The purpose of this command is to retrieve statistic and status information from the RTP-Engine. The gained data are the last sequence number of the sender, the last received sequence number of a remote peer, the actual jitter value, as well as the amount of octets sent so far by this RTP session.

SIOCRTPIDENT The only purpose of this command is to indicate whether the IOCTL is working correctly with the NIC-Wrapper kernel driver or not.

SIOCRTPSSRC This command configures the SSRC register of the Filter.

The device driver's architecture can be seen in Figure 4.6.

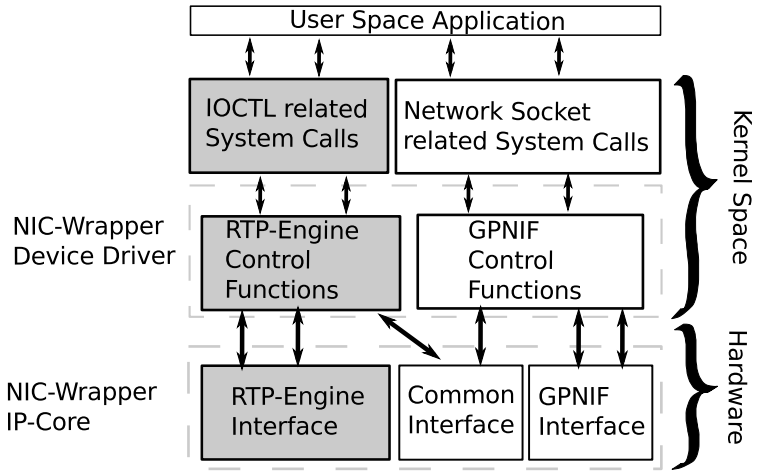


Figure 4.6: NIC-Wrapper Device Driver Architecture

It is split into two parts. The IOCTL related functions controlling the RTP-Engine and the network socket interface related functions controlling the GPNIF. However, both function blocks access the common configuration register, thus the need for synchronization arises. This is done with spinlocks. A spinlock is similar to a semaphore, except that it cannot suspend a process explicitly. Moreover, it performs busy-waiting until the lock is released. Thus, it should only be used if the amount of time spent in the spinlock has a tight upper-bound [JC05, p.116ff].

4.3 Softphone Implementation

This Section introduces the user space application controlling the RTP-Engine, thus being responsible for setting up, executing, and tearing down a Voice over Internet Protocol (VOIP) call. This application is represented by a softphone. A softphone subsumes all components necessary to operate a VOIP call. In this case, the softphone has to provide the following components:

- A user interface transforming a users request into a call establishment, or call tear down.
- A signaling protocol stack responsible for setting up a call.
- A call quality feedback protocol stack providing feedback on the health state of a VOIP session.

This section consists of four parts. In the first Subsection 4.3.1, several available signaling protocol stacks are evaluated. Subsequently, the softphone's architecture is described in Subsection 4.3.2. Afterwards, the implementation of the signaling protocol related software component is described in Subsection 4.3.3. Finally, the realisation of the call quality feedback protocol Real-time Control Protocol (RTCP) is introduced in Subsection 4.3.4.

4.3.1 Choosing a Signaling Protocol Stack

In order to implement the softphone an appropriate signaling protocol to process call administration has to be chosen. Therefore, the following open source stacks shown in the Table 4.1 and Table 4.2 have been evaluated based on the criteria presented subsequently:

- Availability of several open source softphones serving as test peers for the prototype implementation.
- The stack's build system should have cross-compilation targets, in order to integrate it into Buildroot's build system.
- The stack's build system should feature Autotools [46, 45], thus avoiding complex adaptions of its build system to integrate it into Buildroot. Autotools is a software package providing mechanisms to create makefiles and configure scripts automatically using a configuraton file. It is well suited for software projects intended to run on several platforms.

The field **Architectures** in Table 4.1 and 4.2 indicate the target architectures for which the stack is available. The field **Autotools** shows if the stack's build system supports Autoconf and Automake, or just exists as plain Makefile project. Finally, the field **Description** lists additional information about this stack implementation.

The first open source signaling stacks are associated with the H.323 protocol stack, presented in Table 4.1. Both implementations are available to x86 compatible processor only. Moreover, only ooH323c [66] supports the Autotools software package. The project H.323plus [58] offers also a media stack. The open source softphone solutions Ekiga [51], Gnomemeeting [57], and FreeSwitch [56] support H.323.

Table 4.1: H.323 Opensource Stacks

Project Name	Architecture	Autotools	Description
ooH323c	x86	No	Written in a objective style of C. Implements the signaling part of a H.323 stack only.
H.323plus	x86	Yes	Written in C++. Implements a complete media stack, including RTP/RTCP and several voice codecs.

The second variety of protocol stacks presented, implement the Session Initiation Protocol (SIP) protocol. Both SIP stack implementations are available for the used target platform. Moreover, the following softphones implement the SIP as a signaling protocol: Ekiga [51], Kphone [59], Xlite [81], and Linphone [62]. Furthermore, the eXosip [60] project supports the Autotools package in contrast to PJLIB [69] providing support for Autoconf only.

Table 4.2: SIP Opensource Stacks

Project Name	Architecture	Autotools	Description
eXosip	x86,arm,ppc32	Yes	Written in C. Is a wrapper extension to the osip library, also available from `http://www.antisip.com/as/`. Provides a SIP stack only.
PJSIP	x86,arm,ppc32	Partially	Written in C, with a focus on embedded devices. Provides a complete media stack including RTP/RTCP and several voice codecs.

Considering the signaling protocol Jingle, only one library implementing Jingle is considered here. The project is called *libjingle* [61] and provides the signaling protocol stack only. Furthermore it is written in C++ and available to x86 compatible platforms only. However, it is not licensed under the GPL, but it's source code may be used free of charge for commercial and non-commercial projects.

Based on the set up decision criteria for choosing a signaling protocol implementation, the SIP stack library *eXosip* is used. It fulfills all requirements and implements a signaling protocol stack only, thus nothing must be changed in the build system of the project, or the project itself to use the needed parts of the library. However, *eXosip* is implemented as a wrapper of the *osip2* project, hiding several implementation issues. Nevertheless, *osip2* is fulfilling the requirements as well.

4.3.2 Softphone Architecture and Functional Description

The softphone itself is implemented in plain C. It is built using the Autotools package. It consists of the Autoconf and the Automake package. These two software components provide functions to generate configure, and build scripts automatically. Since Buildroot features a corresponding hook in its build system, using Autotools was the first choice to build the softphone.

Basically, the softphone consists of two parts. One is responsible to interact with the user, and the other one communicates with the VOIP stack. Two implementation strategies have been evaluated. On the one hand, the two parts of the softphone have been realised in two distinct threads using the *pthread* library provided by most Linux distributions, including Busybox. Pthread is the Portable Operating System Interface (for Unix) (POSIX) threading library. It is added to Linux, because the kernel has initially no notion about user-level threads [DPB05, p.72ff]. The two threads were meant to communicate over a shared memory synchronized by spinlocks. This synchronization technique was chosen over semaphores because the only information exchanged by the threads would be command and status codes, as well as strings of short length. Moreover, the user input/output thread would only need to access to the critical section rarely and short. Thus, spinlocks would be the synchronization method of choice, because of their lower administrative overhead for the operating system. On the other hand, both parts were implemented in a single threaded application. Eventually, the single threaded implementation is adopted in the final release of the softphone prototype because of the following reasons:

- The Linux kernel version running on the Power-PC processor supports only an old implementation of the *pthread* library. Nevertheless, it is compatible with the new one used on the personal computer were the softphone prototype was implemented first.
- The applications response time rose significantly when running the softphone on the target platform.
- The *eXosip* library lost events when using the two threaded implementation.

The whole application is written in a non-blocking way, thus avoiding any busy-waiting statements. The softphone has the following structure as shown in Figure 4.7.

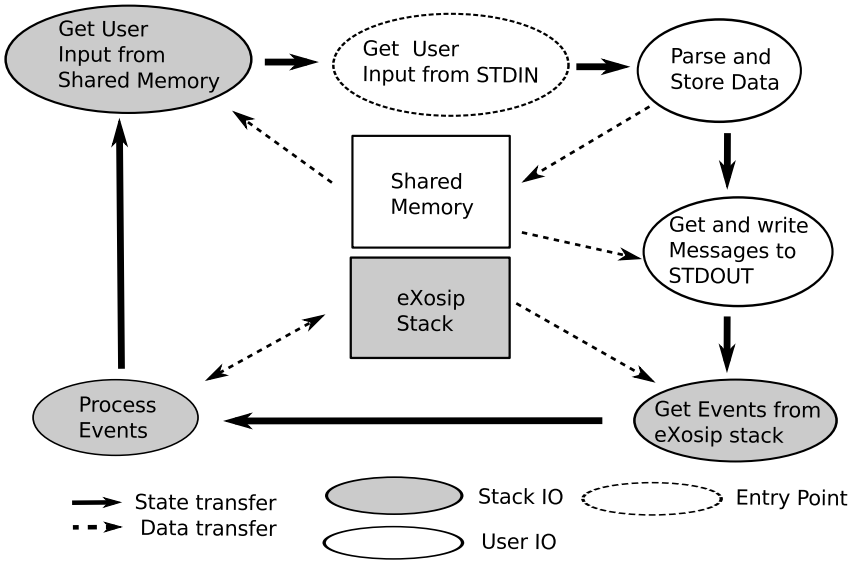


Figure 4.7: Softphone component interaction

It is presenting the components described in Section 4.3.2 and 4.3.3 interact with each other, thus omitting the RTCP stack. It can be seen that a common data structure storing messages and strings which are exchanged by the user IO and the stack IO functions is checked for new input from the user IO functions first. In the next step new user IO requests are parsed and

stored in the common structure. Subsequently, messages generated by the stack IO components are retrieved from the common data structure and are presented to the user. Afterwards, new stack events are read from the *eXosip* stack. These events are processed in a case statement controlling the further operations of the SIP stack, like setting up, executing and tearing down calls, and the generation of messages for the user.

Upon application start an initial configuration vector supplied by a command line argument consisting of the following elements is handed to a worker function implementing the softphone's features:

- An IPv4 address the application is listening on. It will be used as a configuration parameter of the RTP-Engine's source address register. Moreover, it will be used to retrieve the appropriate MAC address associated with this IPv4 address.
- The Universal Datagram Protocol (UDP) port number the SIP stack shall listen on.
- The UDP port number the RTP-Engine's Filter will be configured to listen on.
- A string representing the name of the softphone sent to a callee via SIP.

After this information is handed over to the worker function, the following tasks are performed to prepare the phone for a VOIP call: First of all, the SIP stack is initialized and a corresponding socket is opened to listen on the appropriate port provided by the configuration vector. If this does not succeed, the program terminates with an error message. If it does, the standard gateway's IPv4 address is extracted from the *Base-System's* routing table if available. Furthermore, the gateway's MAC address is retrieved from the Address Resolution Protocol (ARP) table, if available. Subsequently, the local MAC address of the first Ethernet device is retrieved from the ARP table.

The information just gained is needed to configure the RTP-Engine appropriately, because of the following three cases:

Case 1: Two communication partners are located within the same subnet and no registrar is used. This is the simplest case, thus only the local and the remote MAC address are needed to configure the RTP-Engine. The necessary remote MAC address can be read from the ARP table after a call has been established by the SIP stack (see Figuref 4.8).

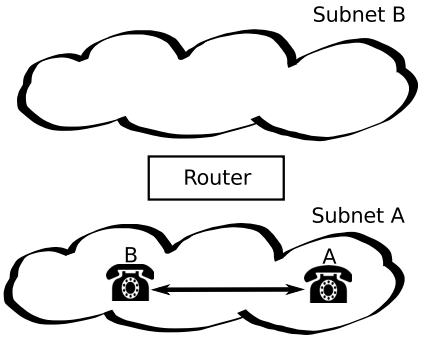


Figure 4.8: Call partner/SIP Proxy constellation: Case 1

Case 2: The two communication partners may use a SIP-proxy, but are located in distinct subnets. Therefore, the MAC address from the standard gateway is needed as a destination MAC address for the RTP-Engine's configuration vector. The peers appropriate IPv4 address can be taken from the messages exchanged by the SIP stack (see Figuref 4.9).

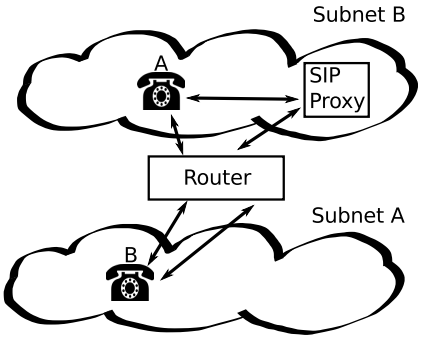


Figure 4.9: Call partner/SIP Proxy constellation: Case 2

Case 3: In this scenario, two communication partners set up a call using a SIP-proxy, but are located in the same subnet. Furthermore, it is assumed that the two peers do not know about each other in the ARP table. Therefore, the last SIP *ACK* message exchanged during a call establishment has to be awaited until the call partner's MAC addressed can be retrieved from the ARP table. The last *ACK* message is needed because it is the first packet exchanged peer-to-peer when using a SIP-proxy to establish a call (see Figuref 4.10). See Chapter 2.1.

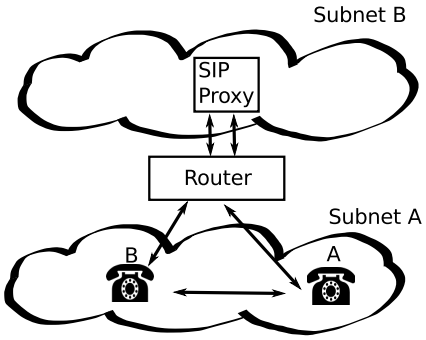


Figure 4.10: Call partner/SIP Proxy constellation: Case 3

Afterwards, the main loop is entered and the user inputs are processed. This is done in a non-blocking way using the *select()*[9] system call watching the STDIN file pointer. Subsequently, the user input is fed into the SIP stack. If a new call has been established, the RTP-Engine configuration vector is assembled. Therefore, the following information is provided to a structure handed over to the RTP-Engine with the *ioctl()* system call:

- A complete MAC header containing the source and destination MAC addresses from the ARP table.

[9]Select provides a mechanism to watch a file-pointer and perform certain tasks if its "ready", thus has new data available [DPB05, p.427ff].

- A complete IPv4 header containing the source and destination IPv4 addresses from the Session Description Protocol (SDP) message and the softphone's configuration vector from the command-line.
- A complete UDP header containing the source and destination address of the RTP session gained from the SDP message and the softphone's configuration vector from the command line. It is sufficient to use its own applications RTP-listen port as source port according to [87].
- A complete RTP header containing a random generated SSRC, initial time stamp, and sequence number offset, taken from */dev/urandom*[10], which is used to generate random numbers.

If the user issues a call tear down, the RTP-Engine is stopped via the *ioctl()* system call again. Furthermore, if the softphone is to be shut down, the SIP stack is closed down again and the main loop is left terminating the program. If a softphone termination is requested although a call is still active it is closed first, the RTP-Engine is stopped, and the program is closed cleanly after stopping the SIP stack.

4.3.3 SIP Implementation

The implementation of the SIP stack related part of the softphone bases on the *eXosip* library. It provides a front-end to the *osip2* SIP stack library. The front-end exports the incidences inside the SIP stack in the form of *events*. Each *event* provides access to the data associated with it. This may be a whole SDP packet. Furthermore, several utility functions exist to access the content of an *event* in a convenient way.

The retrieval of *events* is done by a single function, *get_event_wait()*, operating in a non-blocking way, thus utilising the *select()* system call. However, the implementation of *eXosip* requires that a function calling *get_event_wait()* has to wait at least a microsecond for a newly arriving event, thus introducing a small delay in the application. If the parameter responsible for the time-out is set to zero, the stack does not work correctly anymore. Each event received is processed by a *switch/case* statement. Every case in this statement represents a SIP message code, thus an appropriate reaction to a call partner's response is possible.

Moreover, several utility functions are used to generate and adapt SDP messages. They provide functions to access the contents of an SDP message by addressing the Identifier (ID) of a corresponding *eXosip event*. Furthermore, the functions *send_invite*, *accept_invite*, and *get_call_param* are written to encapsulate these utility functions.

The SIP stack and the user function communicate with each other by exchanging messages over a common structure. The structure consists of the following fields:

uac_cmd The uac_cmd is a character sent from the user input to a function interpreting it as a command to the SIP stack. This can be a call establishment, acceptance or decline command. Moreover, the command can also be interpreted as a softphone shut-down command.

[10] Unfortunately, *dev/urandom* generates pseudo random numbers only [DPB05, p.427ff]. However, the ML405 lacks a true random number generator device.

caller_id This is the caller-ID received from the SIP stack sent to the user IO function to display to the user.

address This is the SIP-URI sent from the user IO function to the SIP stack to establish a call.

uas_event This event is used to exchange softphone event codes and commands between the user function and the SIP stack. This includes command codes sent from the user function to the SIP stack, as well as status codes, indicating that this action is not possible because there is no call currently taking place.

4.3.4 RTCP Implementation

This section presents the implementation of the RTCP stack. Although there exist several open source media stack implementations, none of them was suitable to be adapted to the special needs of the implementation of a VOIP stack in an FPGA. The main reason is that most available stacks implement RTP and RTCP in a very tight way, so that a detachment of both is hard to achieve. This is the case considering the *ccRTP* [49] stack, written in C++, as well as the *oRTP* [68] stack. An example implementing the RTP and RTCP somewhat separately, therefore making their separation more comfortable is the *PJMEDIA* stack. However, it does only support Autoconf, and comes with a two layer design. The lower layer is designed to be hardware dependent, thus providing a generic wrapper to the upper one, implementing the RTP/RTCP stack. However, this layer makes it necessary to either adapt the RTP/RTCP stack significantly, or to uncouple the RTCP/RTP layer from the hardware dependent one.

Evaluating several RTP/RTCP implementations showed that both approaches require severe and error prone changes to the software package. Thus, the RTCP stack is implemented from scratch using RFC-3550. Usually, an RTCP stack retrieves all data necessary to build reports and to dimension the jitter buffer by accessing certain data structures in software, provided by the RTP stack. Since the RTP stack is implemented in hardware by the RTP-Engine, an appropriate configuration interface has been proposed in Section 3.7. The interface design concluded that an implementation of certain RTCP specific functions, like jitter calculation, amount of received, amount of transmitted packages and the amount of sent octets, would be best suited to be realized in hardware. This leads to the need of an appropriately implemented interface providing access to the RTP-Engines configuration register. The remainder of this Section gives an overview on the most important functions implementing the key features of the RTCP stack.

The RTCP implementation is split up into two parts. One part realises certain utility functions, like providing wrappers for the socket interface, time conversion functions, software timer configuration functions, and IOCTL interfacing functions. The other part implements the actual functionality of the RTCP stack. The interface of the stack consists of three functions:

start_rtcp() This function is called when the call parameters are exchanged by the SIP stack and the RTP-Engine has been started. It takes all necessary parameters including sampling clock frequency, RTP packet payload size, and the initial RTCP bandwidth value. The passed destination and source ports should be the RTP related destination and source ports increased by one.

process_rtcp_packet() The process RTCP packet function must be located in the main loop of a program using the stack. It processes all RTCP related tasks including reception and

transmission of RTCP packets as well as rescheduling timeouts. The function is implemented in a non-blocking way, using the *pselect()* system call where necessary. The *pselect()* system call works similar to the *select()* system call, except that it takes care of the fact that a watched function my be interrupted by a signal handler routine [DPB05, p.318ff].

stop_rtcp() This function must be called if it is decided that an ongoing VOIP call has to be terminated. It frees any resources and closes the RTCP related socket connections.

Considering the RTCP functional implementation all basic algorithms provided by Schulzrinne et al.[87] have been used. They include algorithms to calculate the next transmission interval of an RTCP packet as well as parsing and mangling received RTCP packets. Nevertheless, the structure of the stack had to be designed from the ground up resulting in the following behavior of the stack. After the function *start_rtcp()* has been called, the next occurrence of the software timer interrupt is scheduled in *start_rtcp()*. However, the timer is not yet started, because the following initialisations have to be made:

- Initialising the sessions SSRC table, where all known SSRCs of an RTP session are listed until they time out.
- Initialising the session structure with the parameters retrieved upon the call of *start_rtcp()*.
- Setting up the socket interfaces to send and receive RTCP packets. Furthermore, a socket to receive RTP packets not detected by the Filter is opened for testing purpose.

After this has been done, the software timer is started. However, the signal handler function called upon a signal[11] raised does only set a flag, indicating that the timer is expired. The real calculation process is performed by the *process_rtcp_packet()* function.

When *process_rtcp_packet()* is entered it is checked if the RTCP has already been started by checking a state variable. If the stack is already operating the flag set by the software interrupt service routine is tested. If it is set, the function *OnExpire()* is called after the current time stamp has been retrieved from the monotonic clock. *OnExpire()* determines if a RTCP report shall be sent, or if it has to be rescheduled. This function is taken from the RTCP reference implementation from RFC-3550. However, the functions called inside *OnExpire()* to assemble and send RTCP reports are written from scratch.

After reports have been sent or postponed, the timeouts of SSRCs in the SSRC-table are calculated. Furthermore, it is checked if a new RTP packet has arrived, for debugging reasons. Afterwards, a newly received RTCP packet is parsed and its statistic information are retrieved for further processing by the function *onReceive()*. The reception of RTP and RTCP packets is done in a non-blocking way. Eventually, the function is left returning 0 if no error occurred and -1 if an error occurred. Errors are defined by an unsuccessful return of the packet reception functions, although *pselect()* issued that there was new data available. This includes a detailed description of functions implementing the building and transmission of report packets, which is a central implementation detail since it requires that the RTCP stack accesses the RTP-Engine's configuration interface. The implementation of the report generation procedure is shown in Figure 4.11.

[11] Signals are sometimes called software interrupts [DPB05, p.318ff].

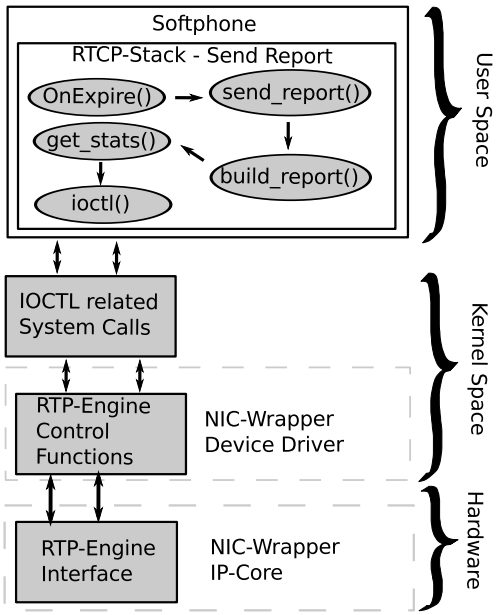


Figure 4.11: Generate RTCP reports by interfacing the RTP-Engine

Considering the function *build_report()* called within *send_report()* in the function *OnExpire()*, it must be noted that it realises the interface to the RTP-Engine via the IOCTL interface. The retrieval of the RTCP report relevant data is retrieved in the function *get_stats()*. Inside *get_stats()* *ioctl()* is called retrieving the needed data using a structure containing the following fields:

last_seq_tx An unsigned 16-bit wide integer variable containing the sequence number sent in the last RTP-packet.

last_seq_rx An unsigned 16-bit wide integer variable containing the last received RTP sequence number.

jitter The jitter value calculated by the RTP-Engine is stored in an unsigned 32-bit wide integer variable.

last_ssrc This field represents the last SSRC used to configure the *RTP-Engine's* Filter.

sent_octets Reflects the amount of sent octets by the current RTP session. it is stored in an unsigned 32-bit integer variable.

The retrieved statistic data is used to calculate the remaining information needed for a RTCP report by *build_report*. After this function returns, the report is sent.

When an RTP session is closed, an RTCP bye packet has to be sent. This request is made by calling *stop_rtcp()*. This function sets a flag which is evaluated in *process_rtcp_packet()*, thus calling *send_bye()* and afterward *teardown_rtcp()* to stop the timer and release the socket interface resources. *send_bye()* transmits a bye packet letting a callee know that the session is over, immediately.

4.4 Implementation Summary

First, the Base-System implementation consisting of a System on Chip (SOC) and the corresponding software has been described. The used components were chosen evaluating several hardware and software options leading to a SOC utilising a Power-PC 405 hard-core and an embedded Linux distribution as operating system. Furthermore, the NIC-Wrapper interface has been described from a hardware and software point of view. Two versions of implementing the kernel device driver have been evaluated, where the IOCTL based solution was chosen over the character-device driver implementation. Finally, the realization of the softphone has been described. First, the process of choosing an appropriate library implementing the signaling protocol stack, leading to the use of *eXosip* was documented. Then an overview on the RTCP stack implementation, with respect to the functions responsible for building reports was given.

5 Results

The last Chapters introduced how to design and implement a VOIP stack in an FPGA, utilising a pre-existing softcore performing the signal-process related tasks of a VOIP session. The purpose of this Chapter is to present the results provided by the prototype implementation. In Section 5.1 the synthesis and partition place and route results are presented amongst a few steps to reduce the design size further. The subsequent two sections 5.2 and 5.3 present commented simulation results regarding the Read-First In First Out (FIFO) in packet-mode and the RTP-Engine/GPNIF arbitration. Section 5.4 gives an overview on the setup used to verify Voice over Internet Protocol (VOIP) stack implementation on the Virtex4 FPGA. Afterwards, Section 5.5 shows an explained trace of a SIP- and a RTP/RTCP-session. Eventually, the results are summed up in Section 5.6.

5.1 Synthesis Results

As mentioned in Section 4.1.1, the ISE design suite together with Embedded Development Kit (EDK) from Xilinx was used to design and build the Base-System including the NIC-Wrapper. This section shows several implementation results using ISE and EDK in version 10.1. An overview on the final build results can be seen in Table 5.1. It shows the FPGAs logic utilisation level when building the Base-System including the NIC-Wrapper. As it can be seen the design occupies almost the whole Virtex4 Field Programmable Gate Array (FPGA) considering logic slices and more than a half of the available Block-RAM. Nevertheless, compared to the used logic slices only 73% of the available 4-Input LUTs are used. However, this already includes the following optimizations:

Write-FIFO The packet-mode support of the Write-FIFO as well as its vacancy calculation support were abandoned. The Write-FIFO does not need the packet-mode which can be used to retransmit a frame if the Ethernet Media Access Control (MAC) encounters an already used bus when trying to send, resulting in a back-off. However, the FIFOs included in the Ethernet MAC interface take care of these issues. Therefore, the packet-mode can be omitted. The vacancy calculation of the Write-FIFO exports a counter to the IPIF, showing the amount of words able to be written to the FIFO. It can be abandoned as well because the NIC-Wrapper design allows only one frame written to the Write-FIFO at a time.

Read-FIFO In the case of the Read-FIFO the vacancy calculation is omitted because the amount of octets written to the Read-FIFO is provided by a FIFO in the GPNIF. This FIFO is

used in favor of the vacancy calculation because vacancy reflects the amount of stored words only.

Burst Support Burst support is only suitable if all devices on the bus support it. However, the UART, the Interrupt controller, the Electrically Erasable Programmable Read-Only Memory (EEPROM) interface as well as the CF-Disk controller do not support this features, thus it can be omitted [102].

Memory Controller This device provides read and write buffers to increase its performance. Initially, these buffers were configured to be implemented as LUT-RAM. However, because of the amount of allocated resources the configuration was switched to a BRAM based implementation.

Table 5.1: Overview on the Synthesis Results of the Base-System including the NIC-Wrapper

Logic Utilisation	Used	Available	Utilization in %
Registers in Slices	9.309	17.088	54
Occupied Slices	8.542	8.544	99
Used 4 Input LUTs	12.518	17.088	73
Used Block-RAM	43	68	63
Used PPC405	1	1	100
Used JTAGPPCs	1	1	100
Used EMACs	1	2	50
Used IDELAYCTRLs	2	12	16

Table 5.2: Module specific Resource Allocation: Base-System & NIC-Wrapper

Entity Name	Slices	LUTs	Registers	LUT/BRAM
Base-System Total	10.6012	12.518	9.303	423/43
DDRAM Controller	2.178	1.923	2.491	156/7
EEPROM Controller	446	518	373	18/0
UART Controller	407	562	356	19/0
CF-Disk Controller	166	136	199	0/0
PPC405 Wrapper	396	382	370	0/0
PLB-Controller	250	317	91	0/0
NIC-Wrapper	5.322	8.014	4.970	219/18
Interrupt Controller	134	129	139	0/0

Beside the obligatory use of the EMAC and the PPC405 processor, a JTAG controller for debugging purposes (see Section 3.3) and about 16% of the available *IDELAYCTRL* elements are used. The *IDELAYCTRL* elements are required to delay the data line in comparison to the clock signal of the EMAC core when receiving an Ethernet frame from the physical, in order to assure certain timing constraints in this case [110].

In order to get a better overview on the resource usage per module, Table 5.2 and Table 5.3 show the resource allocations of a set of selected modules included in the Base-System and the NIC-

Wrapper. Moreover, the modules IPIF, GPNIF and RTP-Engine interface subsume the following sub-modules resulting in the resource allocations seen in Table 5.3.

GPNIF The GPNIF includes the Read and Write FIFO, Serialiser, Deserialiser, FIFO-to-LL interface, LL-to-FIFO interface, the common configuration and status register, and the FIFO responsible for storing the size of a received frame.

RTP-Engine Interface The RTP-Engine consists of the RTP-Engine interface, the Arbiter, the Filter, and the RTP-Engine

IPIF The IPIF subsumes the PLB-to-IPIF bridge, the Interrupt Source Controller, and the address decoder unit including a byte steering block to support byte-addressable 32-bit registers [102].

In order to meet the timing constraint of operating the system with 100 MHz the following modules had to be removed.

IPv6 Filter The IPv6 filter was a part of the Filter module. It has been removed to save FPGA resources, because already 99% of the slices were occupied by the design. Moreover, IPv6 is not supported by the RTP-Engine configuration interface and the softphone.

Clock Recovery The clock recovery module is located inside the RTP-Engine. It is used to reduce network induced jitter, which is introduced by a source and a receiver clock not being fully synchronous [RNH99]. If this module is added to the system, the design will still fit inside the FPGA. However, the timing constraint of operating the system at at least 100 MHz can not be fulfilled anymore. Thus, this module is omitted.

Jitter Calculation The jitter calculation module is responsible for calculating the jitter value in hardware. If this module is added to the system the design will still fit inside the FPGA. However, the timing constraint of operating the system at least 100 MHz can not be fulfilled anymore. Thus, this module is also omitted.

Although the modules responsible for jitter calculation and clock recovery are vital for productive use of a VOIP stack, they can be omitted for the purpose of verifying the interface and the correct behavior of the VOIP stack implementation in general.

The removed clock recovery module uses 151 slices, 218 LUTs ,105 registers and no BRAM or LUT-RAM of the available FPGA resources. The omitted jitter calculation module uses 464 slices, 808 LUTs ,361 registers and no BRAM or LUT-RAM. In order to fit these modules in the design $151 + 464 = 715$ slices, $218 + 808 = 1.026$ LUTs, and $105 + 361 = 466$ registers would be needed additionally. As mentioned above the whole design including all these components would fit inside the Virtex4 FPGA. This will utilize 99% of the slices and $> 80\%$ of the available LUTs. However, the timing requirement of operating the whole system at 100 MHz would not be fulfilled because of partition place and route issues. It would be necessary to free at least 1.026 LUTs and 466 register to integrate both components successfully. This could be achieved by removing the EEPROM-controller and redesigning the RTP-Engine interface.

According to Table 5.2, removing the EEPROM-controller would free 518 LUTs and 373 registers. As a consequence, the bootloader configuration data stored in the EEPROM would have to be entered manually every time the development platform should start the Embedded Linux

distribution. Thus, the removal of the EEPROM controller cannot be considered, as entering the bootloader configuration manually is a tedious and error prone task. For further information on the bootloader configuration, see Appendix B.

The RTP-Engine interface occupies 959 LUTs and 547 registers. Since the RTP-Engine interface is completely implemented in registers, it can be redesigned to BRAM instead. Nevertheless, the redesigned RTP-Engine interface would also require some registers and LUTs to implement the BRAM based interface, thus resulting in a resource reduction of less than the required 1.026 LUTs. Therefore, the redesign of the RTP-Engine interface in the presented way would not be a feasible approach, since the current RTP-Engine interface occupies 959 LUTs only.

Another approach to achieve the timing constraint would be the reduction of the required bus clock frequency from 100 MHz to 66.6 MHz. A frequency of 66.6 MHz is the next valid bus clock frequency, according to [109]. Moreover, a reduction of the processor clock frequency from 300 MHz to 200 MHz would be required. This results from the fact that the PowerPCs clock frequency must have a $1:1$, $2:1$, $3:1$ to $16:1$ ratio compared to the bus clock frequency [109]. Thus, the next valid processor clock and bus clock frequency configuration would be 200 MHz for the processor clock and 66.6 MHz for the bus clock respectively. This will result in the required $3:1$ ratio. However, this will lead to the following problem as the RTP-Engine is statically configured to send one RTP every 1.25 ms. In order to generate one RTP packet consisting of 10 octets every 1.25 ms, where each octet is sampled at a frequency of 8000 Hz, the RTP-Engine is required to run at 100 MHz. Thus, the recalibration of the bus and processor clock frequency cannot be considered. Therefore, an alternative would be to switch to a bigger FPGA providing more resources.

Table 5.3: Module specific Resource Allocation: NIC-Wrapper

Entity Name	Slices	LUTs	Registers	LUT-RAM/BRAM
NIC-Wrapper	5.322	8.014	4.970	219/18
PLB-IPIF	261	942	272	32/0
ISC	16	20	8	0/0
PLB-Slave	245	922	264	32/0
General Purpose Network Interface (GPNIF)	531	517	326	64/8
FIFO-to-LLI	17	28	18	0/0
LLI-to-FIFO	41	58	20	0/0
32-8 Serialiser	2	3	2	0/0
8-32 Deserialiser	45	79	36	0/0
Write-FIFO	138	88	88	0/4
Read-FIFO	206	150	105	0/4
RX-Size FIFO	82	111	57	64/0
RTP-Engine Interface	497	959	547	0/0
Arbiter	26	43	2	0/0
RTP-Engine	3614	3.405	5.796	112/6
Ethernet MAC-Interface	323	345	418	11/4

5.2 Read-FIFO Packet-Mode Simulation

This section presents the simulation results of a functional key-feature of the GPNIF. Since the Read-FIFO module provided by Xilinx does not follow its own specification, and is therefore considered erroneous it is examined here. All simulation related screen-shots were taken from the tool QuestaSim 6.5_e1 offered by Mentor Graphics [71]. In order to be printed nicely the coloured screen-shots were inverted, gray-scaled and lightened up. It consists of two parts split up into two simulations each. Figure 5.3 shows the signal conditions necessary to start the reception of an Ethernet frame. Moreover, Figure 5.4 reflects the signal conditions required to finish its reception. The signals seen have the following meaning:

bus_clk Processor Local Bus (PLB) bus clock running at 100 MHz. It provides the main clock to the NIC-Wrapper. All clocks used inside Intellectual Property Core (IP-Core) are derived from it.

ll_din An 8-bit wide bus transporting the data-octets of a received frame.

state_r This signal refers to the output of a register and indicates the actual inner state of the LL-to-FIFO interface as described in Section 4.2.1.

sof_n, eof_n These two signals are representing the low-active start and end of frame indicators of the LL-Interface. If this signal is low, it indicates the start of a frame, or the end of a frame, respectively.

trans_state Reflects a register holding the actual inner state of the Read-FIFO. Its possible states are *normal_op, mark, update, restore, release*, and *packet_op*.

mark If this signal is set to high, the actual position of the Read-FIFO's write pointer is saved in a register. The *trans_state* advances from *normal_op* or *packet_op* to *mark* and *update* and finally to *packet_op* state.

mark_address Mark_address is the register storing the address issued on a *mark* request.

restore Upon assertion of the *restore* signal the write pointer address stored in *mark_address* register is written back to the *write_address* register. This takes two cycles as well.

release If this signal is issued, the Read-FIFO is put back into *normal_op* again.

wrreq Upon assertion of this signal a word is requested to be written to the Read-FIFO.

write_address It is reflecting the actual address the next word is written to.

wr_addr This signals has the same meaning as *write_address*.

As it can be seen, the *mark* signal **(1)** (see Figure 5.3) is asserted until *sof_n* **(2)** indicates the beginning of a new frame. Now the last position of the write pointer is saved **(3)**. After the *eof_n* signal **(1)** (see Figure 5.4) indicates the end of frame, the frame remains in the Read-FIFO. Thus, the *restore* signal *(2)* is not issued. This corresponds to the implemented behavior introduced in Section 4.2.1. Considering the reception of an Ethernet frame containing a Real-time Transport Protocol (RTP) packet it can be seen in Figure 5.1 that the behavior is exactly the same as in Figure 5.3 showing the beginning reception of an Ethernet frame. However, considering the end of an Ethernet encapsulated RTP packet reception Figure 5.2 shows that the *restore* signal **(2)** is asserted after a complete reception indicated by the *eof_n* signal **(1)** being low. Thus, the received frame is discarded from the Read-FIFO by setting the write pointer back **(3)**.

5.3 RTP-Engine/GPNIF Arbitration Simulation

This section shows the correct behavior of the Arbiter as designed in Section 3.6.2. The simulation of the Arbiter showing two test-cases can be seen in Figure 5.5. The signals seen have the following meaning:

clk, reset The Arbiter is clocked with a 100 MHz clock from the NIC-Wrapper interface. The reset signal is high-active.

cpu, rtp_ll_sender_src_ready_n These signals indicate the wish of a source to gain access to the MAC transmitter interface.

state_r This component is implemented as a register reflecting the inner state of the arbiter as designed in Section 3.6.2.

The first test case is the sole attempt of the GPNIF to grant access to the MAC transmitter interface. Because the RTP-Engine does not have issued the same request, the GPNIF wins the arbitration and can send its frame. In the second case, GPNIF and RTP-Engine issue their resource request at the same time, thus, resulting in the RTP-Engine getting accessing the MAC core interface. The next section presents the setup of the experiment showing the correct behavior of the VOIP stack implementation.

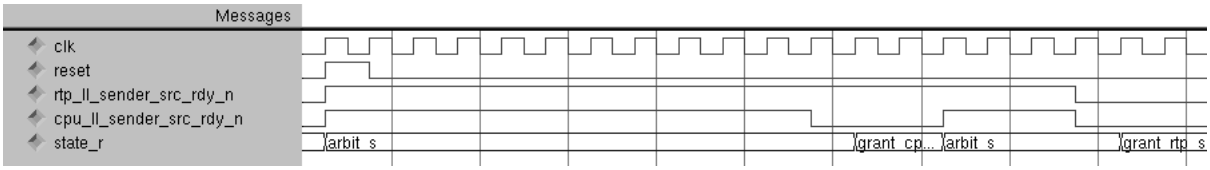


Figure 5.5: Simulation Result: Arbitration between GPNIF and RTP-Engine

5.4 Experimental Setup

The former sections showed the correct behavior of key-elements of the NIC-Wrapper implementation by presenting simulation traces. In order to demonstrate the correctness of the whole approach a VOIP session is monitored in Section 5.5. Therefore, this section introduces the experimental setup used to perform the functional test on the ML405 development board. The test-setup can be seen in Figure 5.6.

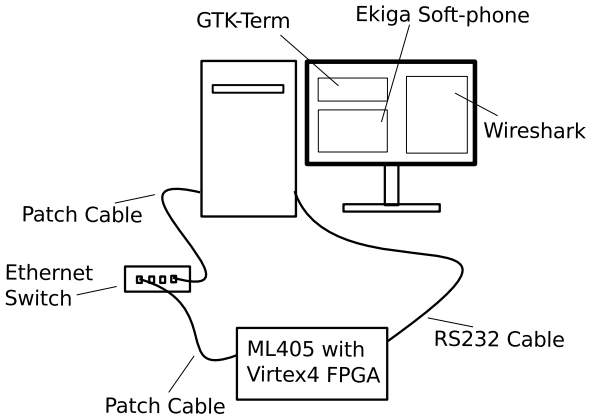


Figure 5.6: Experimental Setup

It consists of a personal computer running a Debian Linux distribution version. The Ekiga VOIP application being the test-peer for the softphone running on the ML405 development board. In addition, the packet tracing software Wireshark is monitoring a USB Ethernet device connected to the PC. The USB Ethernet device is connected to the ML405 via a patch cable using an network switch. Moreover, a USB-to-serial converter dongle is attached to the PC. It is connected to the RS232 serial port of the target-platform and used as a Standard Input Outpu (STDIO) console. The console is interfaced by GtkTerm[1] running on the host PC. In order to offer an appropriate IPv4 address to the ML405, the udhcp[2] Dynamic Host Configuration Protocol (DHCP) server is operated on the PC. Moreover, the retrieval of an IPv4 address by the target-board is used as a test-case for the implemented kernel driver and the GPNIF, showing its correct operation.

5.5 Wireshark Trace of a VOIP session

This section presents a SIP-Session trace to demonstrate the correct usage of the *eXosip2* library in the implementation of the softphone. All screen-shots were taken from the Wireshark packet sniffer and analyzer in the version 1.0.2 [77]. In order to be printed nicely the pictures were gray-scaled. The adjacent screen-shots show different aspects of a SIP-session. The retrieval of an initial Internet Protocol (IP) address by the ML405 development board in conjunction with a MAC/IP address association exchange can be seen in Figure 5.7.

[1] GtkTerm is serial terminal featuring a GTK-base GUI. See `http://gtkterm.feige.net/`. - Visited on 25/02/2011

[2] A small resource conserving implementation of the DHC-Protocol. `http://freshmeat.net/projects/udhcp/`. - Visited on 25/02/2011

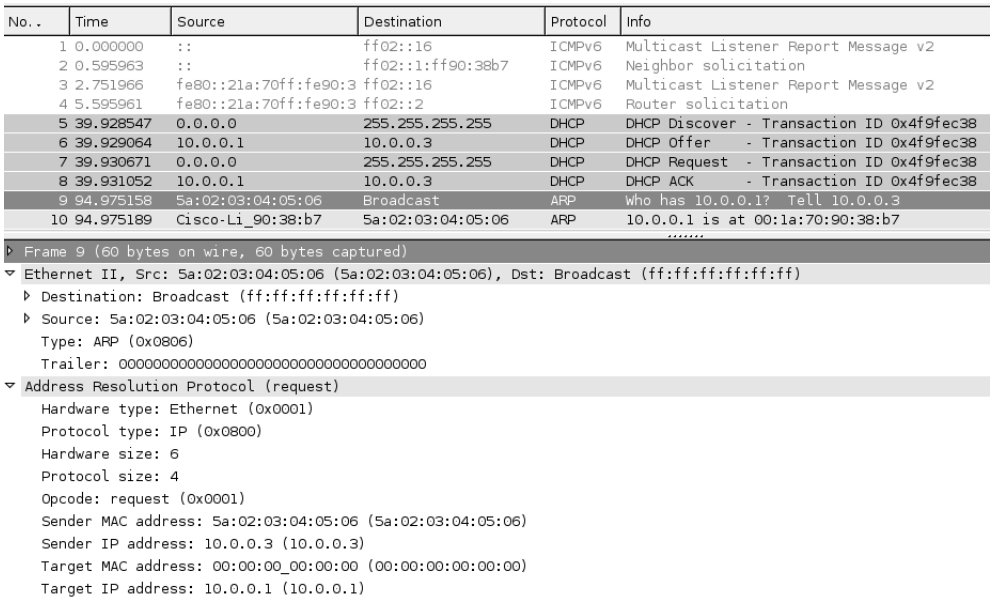


Figure 5.7: Packet Trace: ARP handshake and IP Address configuration via DHCP

The Ethernet frame seen in detail is the ARP request sent by Busybox to receive a MAC-address to IP-address association from the host computer.

Request Call An initial INVITE sent from the prototyping board to Ekiga is presented in Figure 5.8. As it can be seen, the softphone's id is *sip_ua* and it is using the *eXosip* library. Moreover, Wireshark is able to decode the packet correctly, therefore it is transmitted in a valid way showing that the kernel driver, as well as the GPNIF is working correctly. Because the Session Description Protocol (SDP) payload of the Session Initiation Protocol (SIP) INVITE message cannot be seen in Figure 5.8, Figure 5.9 provides a detailed view on the initial SDP payload indicating the use of the Pulse Code Modulation A-Law (PCMA) codec with a sampling rate of 8000 Hz. This corresponds to the media type's definition found in [86]. Furthermore, the RTP-session's intended audio port (8000) is transmitted.

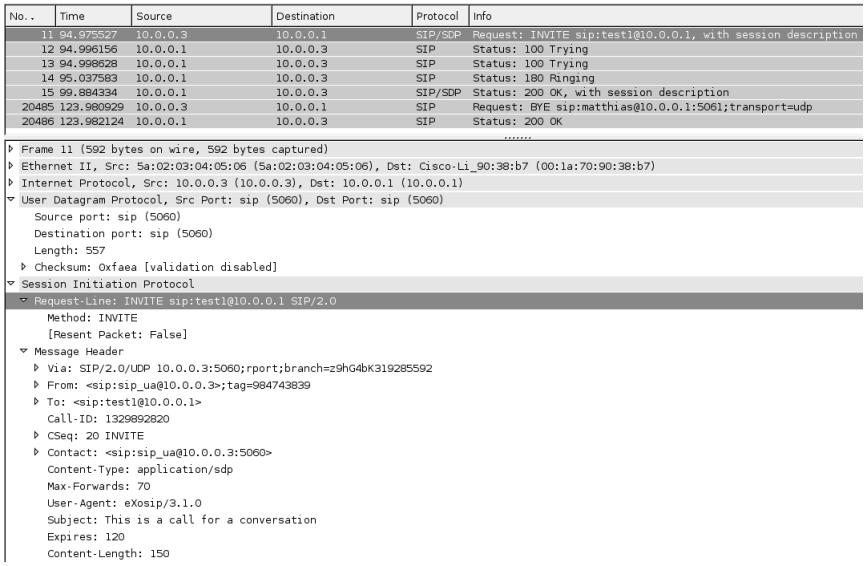


Figure 5.8: Packet Trace: Softphone sends SIP INVITE

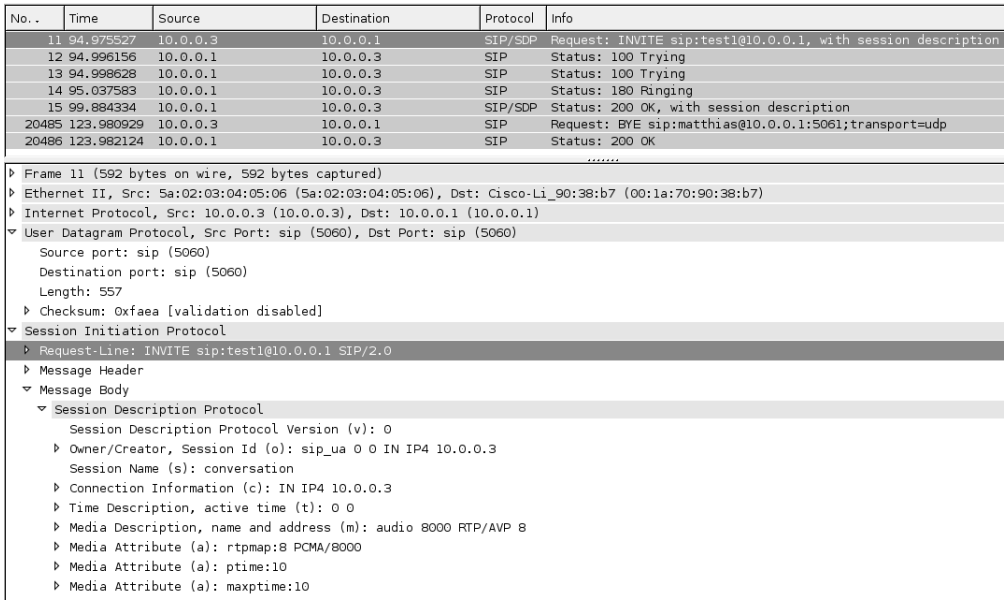


Figure 5.9: Packet Trace: SIP's INVITE SDP message

Remote Peer Accepts Eventually, Ekiga running on the PC answers with a 200 OK SIP message, indicating that the requested codec in the initial INVITE is supported and be used in the VOIP session. This can be seen in Figure 5.10. Moreover, it can be seen that this message also includes the RTP listening port on Ekiga used to configure the RTP-Engine.

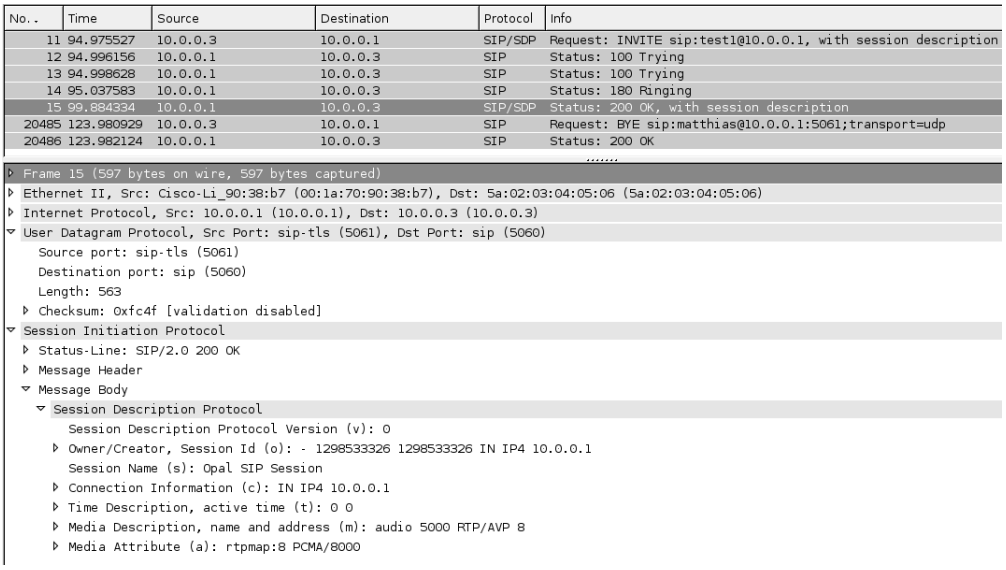


Figure 5.10: Packet Trace: Ekiga answers with SIP 200 OK

RTP/RTCP Packet Exchange By now, all required call parameters are exchanged. Therefore, voice communication related packets can be exchanged. The RTP-packet presented in Figure 5.11 is sent from the ML405 development board. It shows that each RTP-packet contains 10 samples one octet long of PCMA encoded voice. Moreover, its complete header including SSRC, timestamp, and sequence number can be seen. The packet is obviously assembled correctly since wireshark recognized the packet and its fields. A corresponding RTCP-packet is presented in Figure 5.12. A Sender-report sent by the RTP-Engine is shown. As it can be seen, there have been 9.072 packets sent by the RTP-Engine at the report creation time. This number seems quite large, however, only 10 samples are compound in a RTP-packet, thus requiring a high packet transfer rate. Furthermore, the packet length is approved by wireshark showing a correct packet assembly by the RTCP-stack.

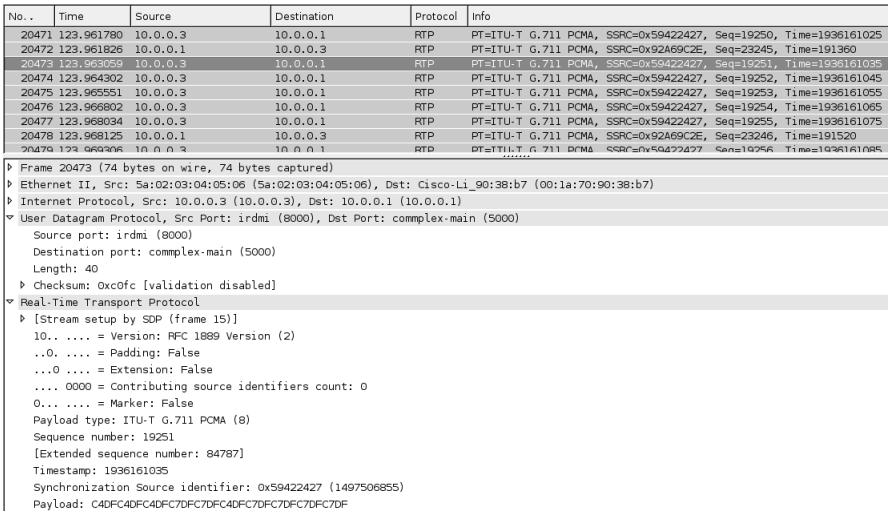


Figure 5.11: Packet Trace: An RTP Packet sent by the Softphone

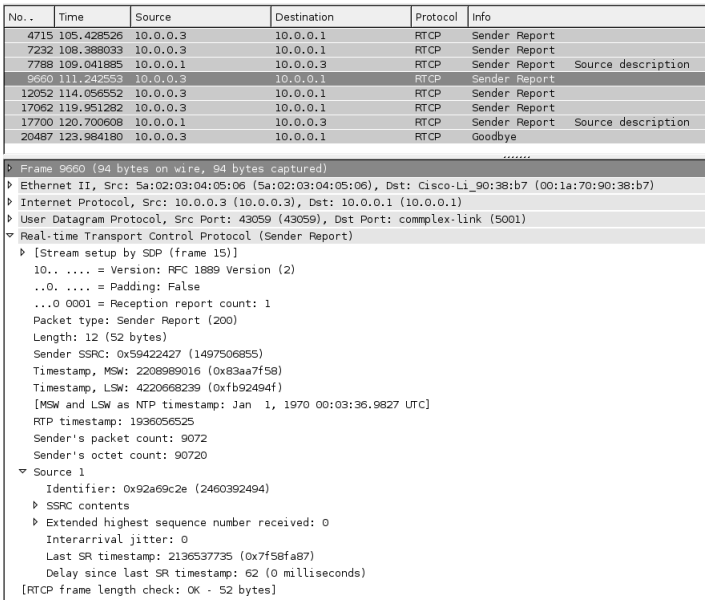


Figure 5.12: Packet Trace: An RTCP Packet sent by the Softphone

Cease Call At the end of the VOIP session the ML405 sends a BYE packet seen in Figure 5.13, which is answered with an ACK message by Ekiga, thus ceasing the connection.

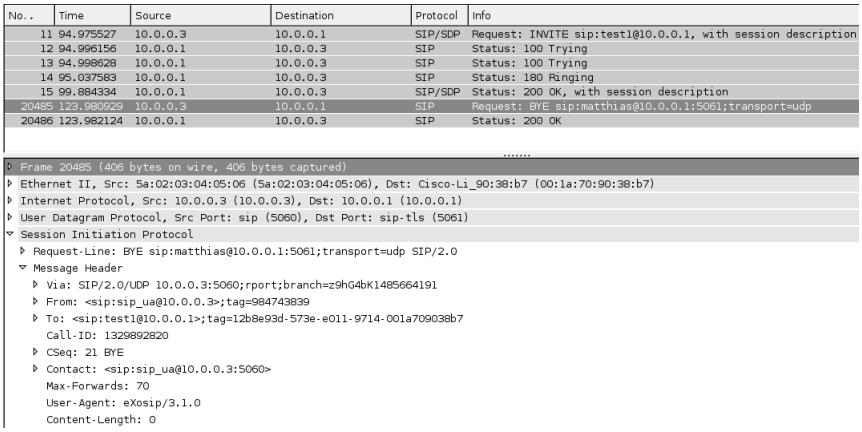


Figure 5.13: Packet Trace: Softphone issues a SIP BYE packet

5.6 Result Summary

This Chapter presented the results achieved by designing and implementing a VOIP stack for high packet rates in a Virtex4 FPGA. First of all, the synthesis results have been shown. Since the whole system consisting of the Base-System, the RTP-Engine and the RTP-Engine interface could not meet its timing requirements, occupies 99% of the FPGAs slices and $>$ 80% of the available LUTs the following components have been removed.

- The IPv6 Filter component has been removed from the RTP-Engine since IPv6 packets are not supported by the RTP-Engine interface anyway.

- The clock recovery module has been omitted because it is not a vital component for the verification of the RTP-Engine interface.
- Although, the jitter calculation module is a vital submodule is is not integrated in the RTP-Engine since it occupies more reconfigurable FPGA resource than the two components named above. Thus omitting the precedingly named components, but keeping the jitter calculation module did not result in a successful implementation of the system inside the FPGA.
- Finally, the reduction of the bus clock frequency from currently 100 MHz to 66.6 MHz was considered. This would reduce the PowerPC core clock from 300 MHz to 200 MHz to respect the specified CPU to bus clock frequency ratio of 1 : 1 to 1 : 16. However, this measurement would lead to an incorrect clock frequency for the RTP-Engine which requires a frequency of 100 MHz.

In order to fit the design inside the FPGA obeying all constraints the move to a bigger FPGA was desired.

The subsequent sections introduced the test environment setup on the one hand. On the other hand, the correct behavior of the arbiter was shown by simulation. Moreover, a complete VOIP session has been recorded and presented as a proof of concept.

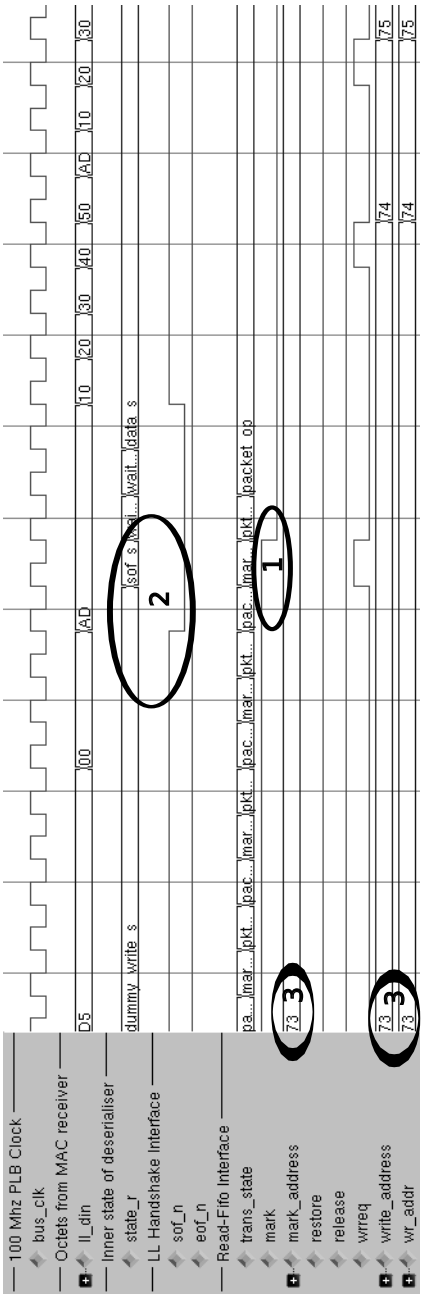


Figure 5.1: Simulation Result: Begin reception of Ethernet Frame without RTP Payload

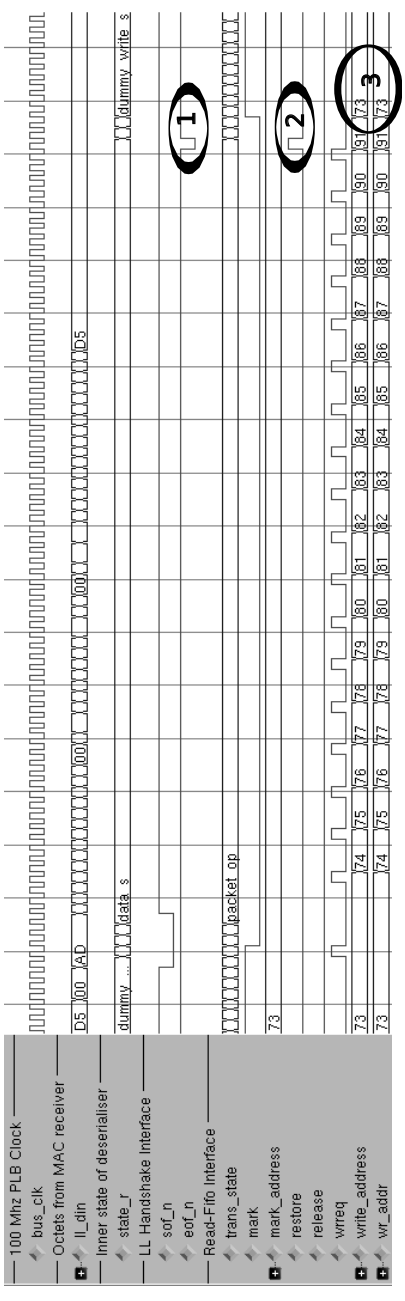


Figure 5.2: Simulation Result: End reception of Ethernet Frame without RTP Payload

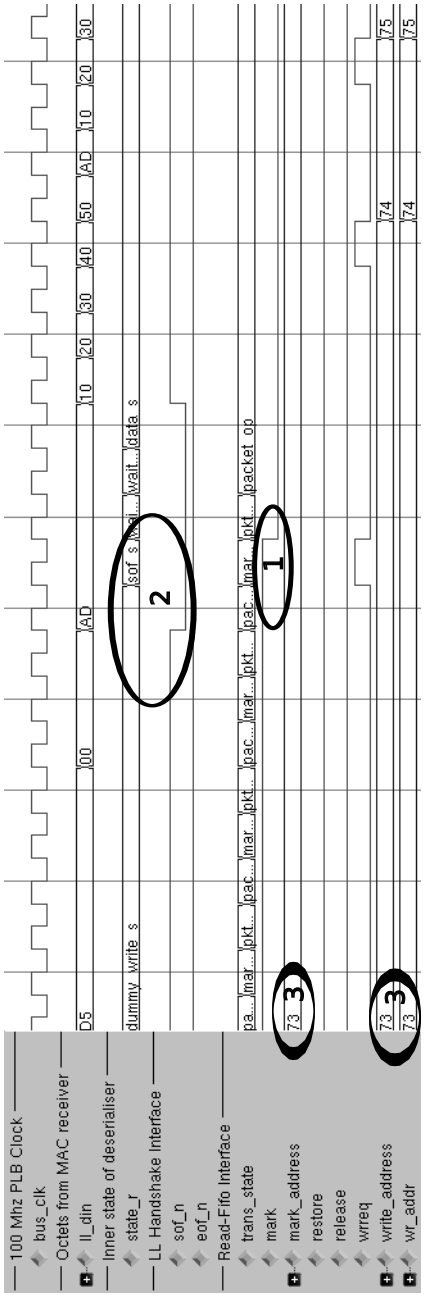


Figure 5.3: Simulation Result: Begin reception of Ethernet Frame including RTP Payload

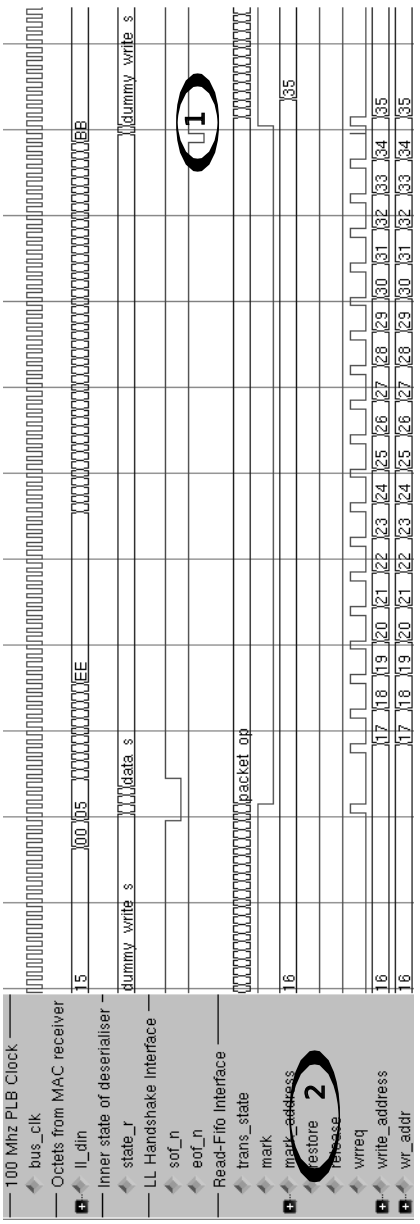


Figure 5.4: Simulation Result: End reception of Ethernet Frame including RTP Payload

6 Conclusion and Outlook

In this thesis, the design and implementation of a VOIP stack in a Xilinx Virtex4 FPGA operated on a ML405 development board is presented. The starting point was a hardware-module performing all computations necessary to basically exchange voice packets between a source and a sink. This IP-Core is called RTP-Engine. It's capabilities are to retrieve a series of voice samples from a microphone, encode and pack them to a RTP-packet and transmit these packets over an Ethernet MAC interface. Moreover, it is able to receive RTP-packets, extract the voice sample, put them into a jitter buffer to regain isochronous behavior, and play the voice samples back via speaker. However, in order to perform a complete VOIP call including a signaling protocol stack, an interface to the RTP-Engine, a media transport protocol stack, and a call-quality feedback protocol stack operating on a SOC have been developed. Where the SOC is terminologically divided into the RTP-Engine, its interface and the Base-System.

The SOC uses a Power-PC 405 core incorporating instruction and data caches to improve the system's reaction time. Moreover, it comprises 128 MB main memory, a persistent storage controller and IO components. As an operating system, Busybox, an Embedded Linux distribution was chosen.

Furthermore, an appropriate hardware/software partitioning of the interface between the Base-System and the RTP-Engine has been realised. This interface implements an interconnection between the RTCP stack and the SIP stack of a softphone. The examination of various design possibilities revealed the following partitioning schema:

- The hardware modules responsible for transceiving the VOIP related RTP comprising Ethernet frames (RTP-Engine) and the other Ethernet frames (GPNIF) are subsumed by a single module called NIC-Wrapper. The NIC-Wrapper offers a unified interface to the Linux kernel.
- The Linux kernel offers a device driver located in the network-subsystem to interact with the GPNIF. Considering the configuration of the RTP-Engine a IOCTL based approach has been favored over the implementation of a distinct character device driver to interact with the RTP-Engine. Moreover, the automatic reconfiguration of the Filter of the RTP-Engine is done by the kernel module, rather than at the user space application.
- Considering the part of the RTP-Engine configuration interface realised in hardware it can be said that it's octet costs, resulting in the usage of FPGA resources could be reduced significantly by carefully examining the RTP-Engine's requirements to achieve an operational prototype.

- Moreover, the design decision to implement the module responsible to calculate a running jitter value as a hardware component reduced the pressure imposed to the kernel driver. This is done by reducing it's interrupt frequency from one interrupt at most every 1.25 ms to one interrupt at most every 5 seconds, where 5 seconds is the smallest intervall to send RTCP packets.

Nevertheless, in order to complete the VOIP stack in an FPGA implementation a SIP stack used for signaling, as well as a RTCP stack have been used to realize a softphone. The softphone uses the RTCP stack to interface with the Linux kernel driver controlling the RTP-Engine through the RTP-Engine interface.

Unfortunately, the timing requirements of the whole design could not be met first. However, in order to solve this issue, three RTP-Engine related components were removed from the design, namely, the IPv6 Filter, the jitter calculation module, and the clock recovery component. The only component actively reducing the functionality of the whole system is the jitter calculation module. Nevertheless, the clock recovery component is vital for a productive use of the VOIP stack, as it reduces network induced jitter. Thereore, it can be omitted for the RTP-Engine interface's functional verification.

Due to the fact that the jitter calculation module is omitted, the need for an alternative inclusion of the jitter calculation functions is required. One possibility is to redesign the RTP-Engine in such a way that each RTP packet normally discarded from the Read-FIFO is handed to the operating system for jitter calculation purposes. However, this would lead to a higher pressure on the utilized operating system and the CPU, if each of them would have to handle an interrupt every 1.25 ms at the current configuration setting of the RTP-Engine. Thus, making the current RTP/RTCP partitioning useless, because the RTP-Engine interface is designed to unload the Base-System with RTP related communication tasks as much as possible.

Another option would be to implement a small buffer memory holding an amount of timestamps from arrived RTP packets. This memory must only store the transit time of a RTP packet, since the jitter value can be calculated from it. The transit time can be calculated by subtracting the current local time maintained in the RTP-Engine from the timestamp in the received RTP packet. Thus, deploying the buffer memory could result in reducing the pressure of issuing interrupts in a high frequency from the RTP-Engine. The interrupt interval regarding the jitter calculation could be reduced from at most 1.25 ms to at most $(1.024/4) * 1.25 = 320$ ms. This reduction of the jitter calculation associated interrupt interval can be achieved if the smallest available BRAM size of 1 kByte is used for the buffer memory, since a timestamp is 32 Bits wide. By enlarging the buffer memory the interrupt interval times can be increased further. Deploying the 1 kByte larger buffer would result in the need of an operating system change due to the introduction of firmer real-time constraints caused by the formerly mentioned change of the jitter calculation module. Therefore, the Embedded Linux operating system should be exchanged by a real time operating system like Xilkernel. However, this would lead to a thorough redesign of the software components of the VOIP stack in an FPGA.

In addition to the change possibilities just described of the implemented functions, the following possibilities for enhancements exist:

- Add domain-name resolution to the softphone capabilities.
- Add support for SIP registrars to the softphone.

- Add conference call support to the softphone as well as to the RTP-Engine
- The RTP-Engine configuration interface may be changed from a register model to a Block-RAM using design. However, the resource retrenchments would be too low to use this solution.

Where the first two enhancements are straight-forward to implement in general, the last enhancement possibility requires a more thoughtfully design. First of all, the appropriate administrative data structure must be implemented when using the eXosip library. Moreover, the RTCP stack has to be patched to support more than one concurrently taking place VOIP calls. Furthermore, the RTP-Engine must be extended by some sort of scheduling mechanism in order to serve each VOIP session taking place in the conference call equally. Nevertheless, the extension of the RTP-Engine to support conference-calls may require a migration of the whole system to a larger FPGA, since by now already more than two-third of the chip's resources are occupied. The possibility to change the configuration interface of the RTP-Engine might provide the FPGA resource needed to implement at least a conference-call support for three participants. However, the whole design must be ported to a development platform housing a bigger FPGA, as mentioned in Chapter 5.

Although there is still room to enhance the presented solutions, it is shown how an appropriate hardware/software interface-design for a hardware-module performing the isochronous related tasks of a VOIP stack can be done. Moreover, it is shown that most available RTCP stacks can hardly be adapted to be used with a dedicated IP-core.

A NIC-Wrapper Specification

Overview & Features

The NIC-Wrapper is a soft-core implementing an interface to the RTP-Engine and an Ethernet MAC hard-core. The part of the NIC-Wrapper realising the Ethernet MAC interface is called *GPNIF*. The part realising the RTP-Engine interface is called *RTP-Engine Interface*. However, both interfaces can be accessed over a unique interface provided by the (Intellectual Property Inter-Face) IPIF. The IPIF is dedicated to the (On-chip Peripheral Bus) OPB/(Processor Local Bus) PLB on chip buses developed by IBM and provided by Xilinx to run inside its (Field Programmable Gate Arrays) FPGAs and development platforms. Thus the implementation of the NIC-Wrapper is bound to FPGAs of this vendor by now. The purpose of this manual is to show how to operate the NIC-Wrapper. It contains the following sections:

- Section A introduces the NIC-Wrapper's external interface.
- Section A gives an overview on the available registers grouped in functional blocks. Moreover, their memory offsets from a base address is given.

The IP-Core provides the following features:

- 100Mbit Ethernet controller interface with receiver-ready and transmitter-empty interrupt support
- Polling mode
- Separate sender and receiver FIFOs, where receiver FIFO is capable of dismissing received frames upon request from the RTP-Engine
- Fully configurable sender and filter for RTP-packets
- All registers expect big-endian formatted data.
- Ethernet network controller and RTP-Engine can be controlled separately
- Hardware Jitter-calculation
- Hardware controlled arbitration between GPNIF and RTP-Engine

- Configurable RTP packet header (from Ethernet to RTP protocol header)
- Supports SSRC change during an ongoing RTP-session while operating
- Xilinx Virtex4 Power-PC compliant PLB IP-Core interface

Interface

This section describes the top-level interface of the NIC-Wrapper Core which is needed to connect the IP-Core to a given Host-System. The hardware interface between the NIC-Wrapper Core and the CPU is realized as a PLB-IPIF slave device where the following features are configured.

- 32 bit bus width
- Interrupt source controller
- 2 KByte 32 bit write data FIFO
- 2 KByte 32 bit read data FIFO with packet-mode support
- 17 User accessible 32bit big-endian expecting registers (Used as device configuration and status registers)

As a consequence of this configuration the following output signals and their description are available and can be seen in Table A.1. The corresponding input signals are listed in Table A.2. Moreover, the ethernet controllers physical interface, a MII interface is implemented with a constant speed of 100Mbit supporting full duplex mode only. Table A.3 shows its IO signals.

Table A.1: PLB-Slave Output Signals

Signal name	Direction	Description
Sl_addrAck	O	Slave address acknowledge
Sl_SSize	O	Slave data bus size
Sl_wait	O	Slave wait indicator
Sl_rearbitrate	O	Slave re-arbitrate bus indicator
Sl_wrDAck	O	Slave write data acknowledge
Sl_wrComp	O	Slave write transfer complete indicator
Sl_wrBTerm	O	Slave terminate write burst transfer
Sl_rdDBus	O	Slave read data bus
Sl_rdWdAddr	O	Slave read word address
Sl_rdDAck	O	Slave read data acknowledge
Sl_rdComp	O	Slave read transfer complete indicator
Sl_rdBTerm	O	Slave terminate read burst transfer
Sl_MBusy	O	Slave busy indicator
Sl_MWrErr	O	Slave write error indicator
Sl_MRdErr	O	Slave read error indicator
Sl_MIRQ	O	Slave interrupt indicator
IP2INTC_Irpt	O	Interrupt output to processor

Table A.2: PLB-Slave Input Signals

Signal name	Direction	Description
SPLB_Clk	I	PLB main bus clock
SPLB_Rst	I	PLB main bus reset
PLB_ABus	I	PLB address bus
PLB_UABus	I	PLB upper address bus
PLB_PAValid	I	PLB primary address valid indicator
PLB_SAValid	I	PLB secondary address valid indicator
PLB_rdPrim	I	PLB secondary to primary read request indicator
PLB_wrPrim	I	PLB secondary to primary write request indicator
PLB_masterID	I	PLB current master identifier
PLB_abort	I	PLB abort request indicator
PLB_busLock	I	PLB bus lock
PLB_RNW	I	PLB read/not write
PLB_BE	I	PLB byte enables
PLB_MSize	I	PLB master data bus size
PLB_size	I	PLB transfer size
PLB_type	I	PLB transfer type
PLB_lockErr	I	PLB lock error indicator
PLB_wrDBus	I	PLB write data bus
PLB_wrBurst	I	PLB burst write transfer indicator
PLB_rdBurst	I	PLB burst read transfer indicator
PLB_wrPendReq	I	PLB write pending bus request indicator
PLB_rdPendReq	I	PLB read pending bus request indicator
PLB_wrPendPri	I	PLB write pending request priority
PLB_rdPendPri	I	PLB read pending request priority
PLB_reqPri	I	PLB current request priority
PLB_TAttribute	I	PLB transfer attribute

Table A.3: MII IO Signals

Signal name	Direction	Description
MII_COL_0	I	Collision detection
MII_CRS_0	I	Clear sender indicator
MII_TXD_0	O	Transmitter data pin
MII_TX_EN_0	O	Transmitter enable indicator
MII_TX_ER_0	O	Transmission error
MII_TX_CLK_0	O	Transmitter clock
MII_RXD_0	I	Receiver data pin
MII_RX_DV_0	I	Data valid indicator
MII_RX_ER_0	I	Receiver error indicator
MII_RX_CLK_0	I	Receiver clock
PHY_RESET_N	O	Physical reset

Register Description

The NIC-Wrapper Core consists of 17 configuration and status registers, controlling both the RTP-Engine and the GPNIF. Furthermore, there are 4 configuration and data registers for each of the two data FIFOs and 3 interrupt source controller registers. Every register is 32 bit wide and synchronous to the PLB clock. Moreover, the registers expect big-endian formatted data. The IP-Core wide base address is called BASE_ADDRESS. It is used by the IPIF to attach the IP-Core to a system's memory map.

Read Data FIFO Registers

The Read Data FIFO is associated with the GPNIF. The base address of the registers used to control the Read Data FIFO is BASE_ADDRESS + $0x200$. However, this does not apply for the Read-FIFO Data Port. In the tables A.4 and A.6 the Read Data FIFO Register are described.

Table A.4: Read Data FIFO Registers

Name	Address offset	Read/Write
RDFIFO Reset register	0x00	W
RDFIFO Module Identification register	0x00	R
RDFIFO Status register	0x04	RW

Write Data FIFO Registers

The Write Data FIFO is associated with the GPNIF. The base address of the registers used to control the Write Data FIFO is BASE_ADDRESS + $0x400$. However, this does not apply for the WRFIFO Data Port. In the tables A.5 and A.6 the Write Data FIFO Registers are defined.

Table A.5: Write Data FIFO Registers

Name	Address offset	Read/Write
WRFIFO Reset register	0x00	W
WRFIFO Module Identification register	0x00	R
WRFIFO Status register	0x04	RW

Table A.6: Read FIFO and Write FIFO Data Port

Name	Address	Read/Write
RDFIFO Data Port	BASE_ADDRESS + 0x300	R
WRFIFO Data Port	BASE_ADDRESS + 0x500	W

NIC-Wrapper Configuration/Status Registers

The base address of the registers used to control the NIC-Wrapper Configuration/Status registers is BASE_ADDRESS. Table A.7 gives an overview of the NIC-Wrapper Configuration/Status registers. However, all registers except the RTPE_CONFR register are associated with the RTP-Engine Interface.

Table A.7: NIC-Wrapper Configuration/Status Registers

Name	Address offset	Read/Write
RTPE_CONFR Configuration register	0x00	RW
RTPE_RX_SZR Received frame size register	0x04	RW
RTPE_ETHDST	0x08	RW
RTPE_ETHDST_SRC	0x0c	RW
RTPE_ETHSRC	0x10	RW
RTPE_ETHTYPE	0x14	RW
RTPE_IPIHL_TO_TOTLEN	0x18	RW
RTPE_IPID_FRAGOFF	0x1c	RW
RTPE_IPTTL_TO_CHECK	0x20	RW
RTPE_IPSADDR	0x24	RW
RTPE_IPDADDR	0x28	RW
RTPE_UDPSRC_DST_PORT	0x2c	RW
RTPE_UDPLEN_CHECK	0x30	RW
RTPE_RTPVER_TO_SEQ	0x34	RW
RTPE_RTP_TIMESTAMP	0x38	RW
RTPE_RTP_SSRC_TX	0x3c	RW
RTPE_RTP_SSRC_RX	0x40	RW

Interrupt Source Registers

The interrupt source registers are associated with the GPNIF, since the RTP-Engine does not raise any interrupts. The base address of the registers used to control the Interrupt source controller is BASE_ADDRESS + $0x100$. In table A.8 the Interrupt Source registers are subsumed.

Table A.8: Interrupt Source Registers

Name	Address offset	Read/Write
RTPE_INTR_GIER Device enable interrupt register	0x1C	RW
RTPE_INTR_IPSR User IP interrupt status register	0x20	RW
RTPE_INTR_IPIER User IP interrupt enable register	0x28	RW

Detailed register description

This section offers a detailed register description describing the meaning of each bit used in the available registers. The NIC-Wrapper Core consists of 17 configuration and status registers controlling both the RTP-Engine and the GPNIF. All registers expect big-endian formatted data. Furthermore, there are 4 configuration and data registers for each of the two data FIFOs and 3 interrupt source controller registers. Every register is 32 bit wide and synchronous to the PLB clock. The IP-Core wide base address is called BASE_ADDRESS.

Read Data FIFO Registers

The Read Data FIFO(RDFIFO) is used to store incoming and accepted ethernet frames. The tables A.9, A.10, A.11 and A.12 specify the registers of the RDFIFO.

RDFIFO Reset Register

Table A.9: RDFIFO Reset Register

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	Reset Data Key	W	n/a	Write $0x0000000A$ to the register to reset the RDFIFO. All other values have no effect.

RDFIFO Module Identification Register

Table A.10: RDFIFO Module Identification Register

Bit(s)	Name	Read/Write	Reset Value	Description
0-3	Major Version Number	R	n/a	Module Major Version Number
4-10	Minor Version Number	R	n/a	Module Minor Version Number
11-15	Minor Version Letter	R	n/a	Module Minor Version Letter. Letters a-z are encoded as 00000 - 11001
16-23	Block ID	R	n/a	Module Block ID
24-31	Block Type	R	n/a	Module Block Type

RDFIFO Status Register

Table A.11: RDFIFO Status Register

Bit(s)	Name	Read/Write	Reset Value	Description
7-31	Occupancy	R	0	RdFIFO Occupancy This is an unsigned value reflecting a current snapshot of the number of locations available for data retrieval from the RdFIFO memory core.
4-6	Width	R	$0x00$	Encoded FIFO Data Port Width. This field reflects the parameterized width of the FIFO data port. In this implementation the FIFO is 32 bits wide. So the value is 000
1-3	Unused	R	0	Unused
0	Empty	R	0	'0' = FIFO is not empty. '1' = FIFO is empty.

RDFIFO Data Port Register and WRFIFO Data Port Register

Table A.12: Read Data FIFO Data Port

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	RdFIFO Data	R	n/a	RdFIFO Data
0-31	Data	W	n/a	WRFIFO Data

Write Data FIFO Registers

The Write Data FIFO(WRFIFO) is used to store to be sent ethernet frames. The tables A.9, A.14, A.15 and A.12 specify the registers of the RDFIFO.

WRFIFO Reset Register

Table A.13: WRFIFO Reset Register

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	Reset Key	W	n/a	Write $0x0000000A$ to the register to reset the WRFIFO. All other values have no effect.

WRFIFO Module Identification Register

Table A.14: WRFIFO Module Identification Register

Bit(s)	Name	Read/Write	Reset Value	Description
0-3	Major Version Number	R	n/a	Module Major Version Number
4-10	Minor Version Number	R	n/a	Module Minor Version Number
11-15	Minor Version Letter	R	n/a	Module Minor Version Letter. Letters a-z are encoded as 00000 - 11001
16-23	Block ID	R	n/a	Module Block ID
24-31	Block Type	R	n/a	Module Block Type

WRFIFO Status Register

Table A.15: WRFIFO Status Register

Bit(s)	Name	Read/Write	Reset Value	Description
8-31	Occupancy	R	512	WRFIFO Occupancy This is an unsigned value reflecting a current snapshot of the number of locations available for data to be written to the WRFIFO memory core.
7	Unused	R	0	Unused
4-6	Width	R	0	Encoded FIFO Data Port Width. This field reflects the parameterized width of the FIFO data port. In this implementation the FIFO is 32 bits wide. So the value is 000
3	Unused	R	0	Unused
2	DL	R	0	'0' = FIFO is not in DeadLock. '1' = FIFO is in DeadLock (Simultaneously Full and Empty due to Packet Mode Operations).
1	Unused	R	0	Unused
0	Empty	R	0	'0' = FIFO is not full. '1' = FIFO is full.

NIC-Wrapper Configuration/Status Registers

A detailed description of the NIC-Wrapper Configuration/Status registers is presented in the tables Table A.16 -Table A.36.

RTPE_CONFR Configuration register

Table A.16: RTPE_CONFR Part A

Bit(s)	Name	Read/Write	Reset Value	Description
25-31	Unused	RW	n/a	Unused
24	RTPE TX Busy	R	0	'1' = RTP Engine transmitter is busy sending a frame. '0' = RTP Engine transmitter is silent, new configuration data can be added. However, be sure to disable the sender while doing so.
23	RTPE RX Busy	R	0	'1' = RTP Engine filter is checking a frame. '0' = RTP Engine filter can be reconfigured. However, be sure to disable the sender while doing so. If a new SSRC is to be written to the Filter configuration this can also be done without disabling the receiver since the new SSRC will only be valid after the filter has finished its checking on the frame.
22	RTPE TX Enable	RW	0	'1' = RTP Engine sender is enabled. '0' = RTP Engine sender is disabled. The RTP Engine sender is NOT reset upon this bit is set to '0'. A reset of the configuration registers must be done in software.
21	RTPE RX Enable	RW	0	'1' = RTP Engine receiver is enabled. '0' = RTP Engine receiver is disabled. The RTP Engine receiver is NOT reset upon this bit is set to '0'. A reset of the configuration registers must be done in software.

Table A.17: RTPE_CONFR Part B

Bit(s)	Name	Read/Write	Reset Value	Description
20	RTPE SSRC changed	R	0	'1' = SSRC in the received packet is different from the others so far received in the stream. '0' = SSRC did not change.
19	TX Enable	RW	0	'1' = MAC Transmitter unit is activated. '0' = MAC Transmitter unit is deactivated, no frames can be sent.
18	RX Enable	RW	0	'1' = MAC Receiver unit is activated. '0' = MAC Receiver unit is deactivated, no frames can be received.
17	TX Frame Done	R	0	'1' = If the WRFIFO is empty. '0' = Otherwise. Writes take no effect.
16	RX Frame Ready	R	0	'1' = If there is a whole new accepted frame in the RDFIFO. '0' = No new frame has been received. Writes take no effect
0-15	TX Frame size	RW	0	The size of the next to be transmitted frame. The transmission starts if the TX enable bits are set and the register is written at the bits 8-15 of this half word.

RTPE_RX_SZR Received frame size register

Table A.18: RTPE_RX_SZR

Bit(s)	Name	Read/Write	Reset Value	Description
0-15	RX Frame Size	R	n/a	Returns the size of a completely received frame from the RDFIFO. The register is coupled to a FIFO which is storing the frame sizes according to their appearance in the RDFIFO.
16-31	Unused	RW	n/a	Unused

RTPE_ETHDST Register

Table A.19: RTPE_ETHDST

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	DSTADDR	RW	0	Stores the first 4 bytes of an ethernet frame destination address.

RTPE_ETHDST_SRC Register

Table A.20: RTPE_ETHDST_SRC

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	DSTSRCADDR	RW	0	Stores the last 2 bytes of an ethernet frame destination address and the first two bytes of ethernet frame source address.

RTPE_ETHSRC Register

Table A.21: RTPE_ETHSRC

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	SRCADDR	W	0	Stores the last 4 bytes of an ethernet frame source address.
0-31	JITTER	R	0	Returns the actual jitter value at the time of register access.

RTPE_ETHTYPE Register

Table A.22: RTPE_ETHTYPE

Bit(s)	Name	Read/Write	Reset Value	Description
0-15	ETHTYPE	W	0	Stores the ethernet frame type.
0-15	RXSEQ	R	0	Returns the last received sequential number from an RTP packet.
16-31	Unused	WW	0	Unused

RTPE_IPIHL_TO_TOTLEN Register

Table A.23: RTPE_IPIHL_TO_TOTLEN

Bit(s)	Name	Read/Write	Reset Value	Description
0-3	Version	RW	0	IP header field 'Version'
4-7	IHL	RW	0	IP header field 'Internet header length'
8-15	TOS	RW	0	IP header field 'Type of service'
16-31	TOTLEN	RW	0	IP header field 'Total length'

RTPE_IPID_FRAGOFF Register

Table A.24: RTPE_IPID_FRAGOFF

Bit(s)	Name	Read/Write	Reset Value	Description
0-15	Identification	RW	0	IP header field 'Fragment identification'
16-18	Flags	RW	0	IP header field 'Fragmentation flags'
18-32	Offset	RW	0	IP header field 'Fragmentation offset'

RTPE_IPTTL_TO_CHECK Register

Table A.25: RTPE_IPTTL_TO_CHECK

Bit(s)	Name	Read/Write	Reset Value	Description
0-7	TTL	RW	0	IP header field 'Time to live'
8-15	Protocol	RW	0	IP header field 'Protocol type of ascending layer'
16-32	Checksum	RW	0	IP header field 'IP header checksum'

RTPE_IPSADDR Register

This register configures the source address of the RTP-Encoder an the destination address of the RTP-Filter.

Table A.26: RTPE_IPSADDR

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	SADDR	RW	0	IP header field 'Source address'

RTPE_IPDADDR Register

This register configures the destination address of the RTP-Encoder an the source address of the RTP-Filter.

Table A.27: RTPE_IPDADDR

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	DADDR	RW	0	IP header field 'Destination address'.

RTPE_UDPSRC_DST_PORT Register

This register configures the module RTP-Encoder and the module RTP-Filter. In the case of the RTP-Encoder the bit-field SRCPORT points to the SRCPPORT of the RTP-Encoder and the bit-field DSTPORT to the DSTPORT of the RTP-Encoder. In the case of the RTP-Filter SRCPORT points to DSTPORT and vice versa. So only one register is needed to configure both RTP-Encoder and RTP-Filter.

Table A.28: RTPE_UDPSRC_DST_PORT

Bit(s)	Name	Read/Write	Reset Value	Description
0-15	SRCPORT	RW	0	UDP header field 'Source port'
16-31	DSTPORT	RW	0	UDP header field 'Destination port'

RTPE_UDPLEN_CHECK Register

Table A.29: RTPE_UDPLEN_CHECK

Bit(s)	Name	Read/Write	Reset Value	Description
0-15	Length	RW	0	UDP header field 'Payload and Header length'
16-31	Checksum	RW	0	UDP header field 'Checksum over the whole packet (IP-header, UDP-header, payload)'

RTPE_RTPVER_TO_SEQ Register

The field SEQ in this register is written by the RTP-Engine as well.

Table A.30: RTPE_RTPVER_TO_SEQ

Bit(s)	Name	Read/Write	Reset Value	Description
0-1	Version	RW	0	RTP header field 'Version'
2	P	RW	0	RTP header field 'Padding'
3	X	RW	0	RTP header field 'Xtension'
4-7	CC	RW	0	RTP header field 'Contributing sources'
8	M	RW	0	RTP header field 'Marker'
9-15	PT	RW	0	RTP header field 'Payload Type'
16-31	SEQ	RW	0	RTP header field 'Sequence Number'

RTPE_RTP_TIMESTAMP Register

This register is written by the RTP-Engine as well.

Table A.31: RTPE_RTP_TIMESTAMP

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	Timestamp	RW	0	RTP header field 'Timestamp'.

RTPE_RTP_SSRC_TX Register

Table A.32: RTPE_RTP_SSRC_TX

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	SSRCTX	RW	0	RTP header field 'Synchronization Source' configuration for the RTP-Sender

RTPE_RTP_SSRC_RX Register

Table A.33: RTPE_RTP_SSRC_RX

Bit(s)	Name	Read/Write	Reset Value	Description
0-31	SSRCRX	RW	0	RTP header field 'Synchronization Source' configuration for the RTP-Filter

NIC-Wrapper Interrupt Source Registers

In the tables A.34, A.35 and A.36 the NIC-Wrapper Interrupt Source registers are illustrated.

RTPE_INTR_GIER Register

Table A.34: RTPE_INTR_GIER

Bit(s)	Name	Read/Write	Reset Value	Description
0	GIE	RW	0	Global Interrupt Enable. '0' = Interrupts disabled '1' = Device Interrupt enabled.
1-31	Unused	R	0	Unused

RTPE_INTR_IPSR Register

The interrupt status register has a toggle on write behavior. That means if '1' is written to bit position where a '1' is already stored the bit inside the register at the corresponding bit position will be toggled to a '0'.

Table A.35: RTPE_INTR_IPSR

Bit(s)	Name	Read/Write	Reset Value	Description
0-29	Unused	RW	0	Unused
30	TXDONE	RW	0	Transmission FIFO empty interrupt. '1' = active, '0' = not active
31	RXDONE	RW	0	At least one completely received ethernet frame is in the receiver FIFO. '1' = active, '0' = not active

RTPE_INTR_IPIER Register

The interrupt enable register has a toggle on write behavior. That means if '1' is written to bit position where a '1' is already stored the bit inside the register at the corresponding bit position will be toggled to a '0'.

Table A.36: RTPE_INTR_IPIER

Bit(s)	Name	Read/Write	Reset Value	Description
0-29	Unused	RW	0	Unused
30	TXDONE	RW	0	Trigger an interrupt if the transmission FIFO is empty. '1' = enable, '0' = disable
31	RXDONE	RW	0	Trigger an interrupt if at least one completely received ethernet frame is in the reciter FIFO. '1' = enable, '0' = disable

Architecture

This section describes the architecture as well as the internal interfaces of the NIC-Wrapper Core. Figure A.1 shows a data-flow oriented picture of the NIC-Wrapper. Figure A.2 shows the general architecture and the sub-module interconnection of the IP-Core. The IP-Core consists of the GPNIF subsuming Write-FIFO,Read-FIFO, serialiser,de-serialiser, Local-Link to FIFO Interface and a FIFO to Local Link Interface (LLI). The LLI is a handshaking protocol for inter-module communication. It is capable of postponing ongoing transmissions if either sender or receiver side cannot cope with the peers data-rate. The RTP-Engine's configuration interface, the Common Configuration and Status Register as well as the GPNIF are interfaced through the Intellectual Property Interface (IPIF). Moreover, Filter, Arbiter, and MAC-Interface communicate through the LLI as well.

Figure A.1: Interface and data flow description NIC-Wrapper Core

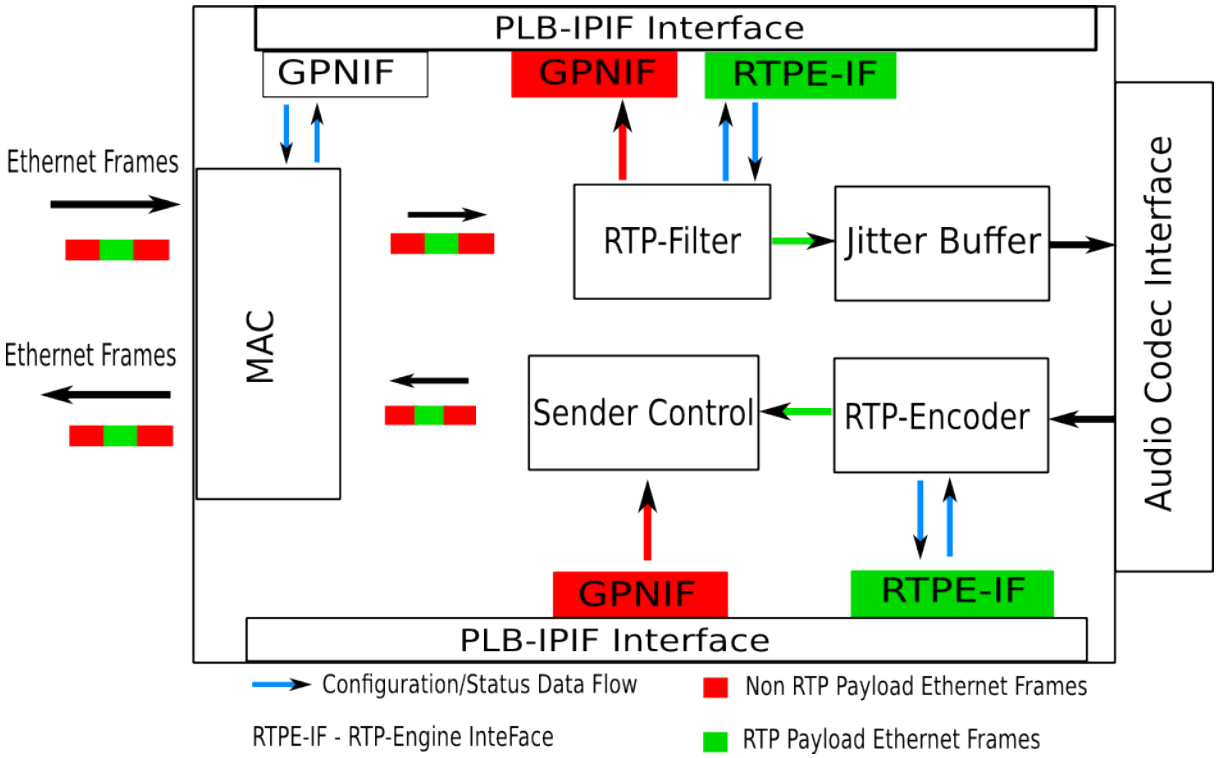
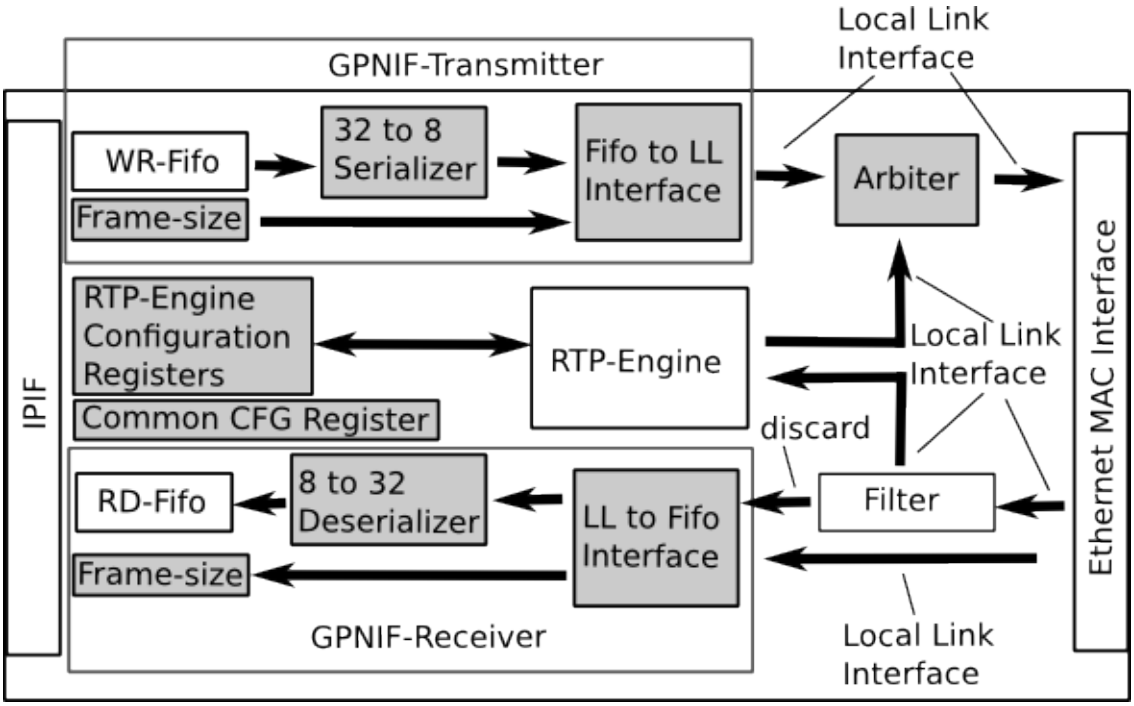


Figure A.2: Architectural description NIC-Wrapper Core



Modes of operation

There are two modes operation available to the NIC-Wrapper Core. Polling and Interrupt-mode. However, apart from these two modes subsection A describes how the RTP-Engine can be operated.

Polling Mode

In Polling-mode all interrupts are deactivated so the bits 30 and 31 in register *RTPE_INTR_IPIER* must be set to 0. Moreover, bit 0 in the *RTPE_INTR_GIER* register has to be 0 as well. To check whether the transmitter is ready to get a new frame written to its transmission FIFO, or the receiver has received a new frame the bits 17 (*TX Frame Done*) and 16 (*RX Ready*) have to read.

Interrupt Mode

To activate the receiver or the transmitter interrupt of the MAC module of the NIC-Wrapper Core the following procedure must be adhered:

- Set the corresponding bit in the *RTPE_INTR_IPIER* register to enable a specific interrupt.
- Set bit 0 in the *RTPE_INTR_GIER* to activate the device interrupt controller.

To acknowledge any occurred interrupt read the *RTPE_INTR_IPSR* register. Mask out the bit of the triggering interrupt and write the word back to the *RTPE_INTR_IPSR* register. Due to the toggle on write behavior of the *RTPE_INTR_IPSR* the interrupt will be acknowledged. To deactivate an interrupt simply apply the interrupt activation procedure in reverse order. By now the RTP-Engine does not generate any interrupts since its time of sampling is determined by the RTCP protocol.

Running the RTP-Engine

Initially the RTP-Engine Configuration registers are set to zero and the RTP-Engine enable bit in the RTPE_CONFR(Table A.16 and Table A.17) register is set to zero as well. The RTP-Engine can be configure by writing a corresponding configuration vector into the configuration registers. After an initial configuration, set bits RTPE TX Enable and RTPE RX Enable in the RTPE_CONFR register (see Table A.16 and Table A.17). This has to be done when the RTP-Engine is not operating, to prevent it from undefined behavior. However, if a new SSRC has to be written to the Filter configuration, which is the case upon the reception of a first RTP packet associated with a VOIP-session, the SSRC can also be written when the RTP-Engine is currently operating. Nevertheless, on has to wait until bit RTPE RX Busy in the RTPE_CONFR register (see Table A.16 and Table A.17) is low, to prevent undefined behavior. To stop the RTP-Engine simple wait until both RTPE RX Busy and RTPE TX Busy are low and unset the bits RTPE TX Enable and RTPE RX Enable in the RTPE_CONFR register (see Table A.16 and Table A.17).

B System Build Howto

This appendix represents a small guide on how to build and run the VOIP-stack prototype implementation on the ML405 development board. It mainly focusses on the build steps of the software components and how to prepare the Buildroot environment to generate appropriate binaries for the target platform.

How-to build a Base System

This small how-to is arranged into 4 sections.

- Section 1 explains a few details which should be concerned when generating a new base system with the Base System Builder Wizard in Xilinx EDK 10.3.
- Section 2 shows how the used software can be built
- Section 3 talks about how to integrate the generated files into the SystemACE disk layout
- Section 4 shows how to set up an U-Boot boot environment to boot automatically into Busybox

If a line starts with "#" its command is to be typed at a shell prompt.

Generate Hardware

Generate new hardware with the BSB wizard (Base System Builder). Include the EEPROM Ip-Core to store the U-Boot environment on it. In the project tab, file: system.ucf change

```
Net fpga_0_clk_1_sys_clk_pin LOC=AB12 | IOSTANDARD = LVCMOS33;
to
Net fpga_0_clk_1_sys_clk_pin LOC=AB14 | IOSTANDARD = LVCMOS33;
```

AB12 is the user clock pin. AB14 is the system clock pin. In EDK Version 11.3 this assignment was faulty. Set the boot-loop to be initialized in BRAM.

Generate Board support package

Generate a device tree with the Xilinx provided device-tree generator [1]. After that export the hardware design and generate a Linux 2.6 compatible board support package with the SDK. Following parameters are needed:

- Memory size: $0x8000000$ -¿ 128MB RAM
- Console: RS232_Uart must be the name of the Uart IP-Core instance
- Additional boot arguments: console=ttyS0,9600 ip=off root=/dev/xsa2 rw
- Finally, add the hardware peripherals to the BSP by clicking "ok" on each item in the view

The last step is to copy the xparamters.h file and the device-tree sources to their corresponding locations Copy Board support file.

```
#cp -v <ROOT_DIR_OF_EDK_PROJECT>/ppc405_0/libsrc/linux_2_6_v1_02_a/
  linux/arch/ppc/platforms/4xx/xparameters/xparameters_ml40x.h <
  ROOT_DIR_KERNEL_SRC>/arch/powerpc/platforms/40x/xparameters_ml405.h
```

Copy Board support file for U-Boot.

```
#cp -v <ROOT_DIR_OF_EDK_PROJECT>/ppc405_0/libsrc/linux_2_6_v1_02_a/
  linux/arch/ppc/platforms/4xx/xparameters/xparameters_ml40x.h <
  ROOT_DIR_UBOOT>/board/xilinx/ml405/xparameters.h
```

Copy device tree sources to kernel directory.

```
#cp -v <ROOT_DIR_OF_EDK_PROJECT>/ppc405_0/libsrc/device-tree_v0_00_x/
  xilinx.dts <ROOT_DIR_KERNEL_SRC>/arch/powerpc/boot/dts/virtex405-
  ml405.dts
```

Build the software

Call

```
#make ml405_config
#make
```

to build u-boot in u-boot source root directory

Call

```
#make uImage
```

to build u-boot favored kernel image in kernel source directory

Call

[1] This tool can be found under http://git.xilinx.com/cgi-bin/gitweb.cgi

```
#scripts/dtc/dtc -b 0 -V 17 -R 4 -S 0x3000 -I dts -O dtb -o ml405.dtb
  -f arch/powerpc/boot/dts/virtex405-ml405.dts
```

in kernel source root directory to build device tree blob

After configuring the tool-chain with

```
#make menuconfig
```

configuring uclibc with

```
#make uclibc-menuconfig
```

and configuring Busybox with

```
#make Busybox-menuconfig
```

alter device-table.txt in the Buildroot directory and call

```
#make
```

in Buildroot root directory to build Busybox

Generate SystemACE disk

Copy the following files to the FAT partition of the systemACE disk

Linux kernel

Copy linux/arch/powerpc/boot/uImage to ml405/linux

Device-tree blob

Copy linux/ml405.dtb to ml405/dtb

Generate system ace file

Call from base_system root directory

```
#xmd -tcl genace.tcl -jprog -target ppc_hw -board ml405 -hw system_top
  .bit -elf ../../../sw/u-boot-xlnx.git/u-boot -ace uboot.ace -
  start_address 0x02002100
```

Copy base_system/uboot.ace to ml405/myace

Setup root partition

Copy the following file to the second systemACE partition

```
#sudo dd if=buildroot2009.08/binaries/build_pc405/rootfs.powerpc_nofpu
  .ext2 of=/dev/sd[abcd]2
```

Finish root file-system partition

```
#sudo /sbin/e2fsck -f /dev/sdc2
#sudo /sbin/resize2fs /dev/sdc2
```

Configure the U-Boot environment

When U-Boot started into interactive mode one can see its prompt "=¿". Execute the following commands:

```
setenv dtb ml405/dtb/ml405.dtb
setenv getdtb fatload ace 0 0x1000000 ml405/dtb/ml405.dtb
setenv linux ml405/linux/uimage
setenv getlinux fatload ace 0 0x400000 ml405/linux/uimage
setenv prepboot run getdtb getlinux
setenv myboot bootm 0x400000 - 0x1000000
setenv bootcmd run prepboot myboot
savenv
```

Now the environment is written to the EEPROM. Upon power-up U-Boot is started, retrieves the device-tree blob, loads the kernel and the system is started automatically. The standard login-mane is "root". No password is required.

Literature

[AGSS09] Asodi, S. ; Ganesh, S.V. ; Seshadri, E. ; Singh, P.K.: Evaluation of transport layer protocols for voice transmission in various network scenarios. In: *Applications of Digital Information and Web Technologies, 2009. ICADIWT '09. Second International Conference on the*, 2009, S. 238–242

[AKBSS10] Ahmed Karim Ben Salem, Slim Ben O. ; Saoud, Slim B.: Field Programmable Gate Array-Based System-on-Chip for Real-Time Power Process Control. In: *American Journal of Applied Sciences* 7 (2010), Nr. 1, S. 127 – 139

[AML+10] Apostolakos, Spyros ; Meliones, Apostolos ; Lykakis, George ; Touloupis, Emmanuel ; Vlagoulis, Vassilis: Design, Implementation and Validation of an Open Source IP-PBX/VoIP Gateway Multi-Core SoC. In: *International Journal of Parallel Programming* 38 (2010), S. 288–302

[APSL05] Abbasi, T. ; Prasad, S. ; Seddigh, N. ; Lambadaris, I.: A comparative study of the SIP and IAX VoIP protocols. In: *Electrical and Computer Engineering, 2005. Canadian Conference on*, 2005, S. 179 –183

[Bad07] Badach, Anatol. *Voice over IP DIE TECHNIK*. 2007

[BHS09] Brunmayr, P. ; Haase, J. ; Schupfer, F.: Late Hardware/Software Partitioning by Using SystemC Functional Models. In: *Modelling Simulation, 2009. AMS '09. Third Asia International Conference on*, 2009, S. 194 –199

[BMMR09] Bonfiglio, D. ; Mellia, M. ; Meo, M. ; Rossi, D.: Detailed Analysis of Skype Traffic. In: *Multimedia, IEEE Transactions on* 11 (2009), January, Nr. 1, S. 117 –127

[BW05] den Braak, Michel V. ; Wong, Stephan: FPGA Implementation of Voice-over IP. In: *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, 2005, S. 338 –342

[CCA07] Chung, Kyung H. ; Choi, Myung S. ; Ahn, Kwang S.: A Study on the Packaging for Fast Boot-up Time in the Embedded Linux. In: *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, 2007, S. 89 –94

[CWX+03] Chen, Xiuzhong ; Wang, Chunfeng ; Xuan, Dong ; Li, Zhongcheng ; Min, Yinghua ; Zhao, Wei: Survey on QoS management of VoIP. In: *Computer Networks and Mobile Computing, 2003. ICCNMC 2003. 2003 International Conference on*, 2003, S. 69 – 77

[DB06] Dai, S. ; Bozorgzadeh, E.: CAD Tool for FPGAs with Embedded Hard Cores for Design Space Exploration of Future Architectures. In: *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, 2006, S.

329 –330

[DPB05] Daniel P. Bovet, Marco C. *Understanding the Linux Kernel, Third Edition.* 2005

[FCP09] Fathi, Hanane ; Chakraborty, Shyam S. ; Prasad, Ramjee: Voice over Internet Protocol. In: *Voice over IP in Wireless Heterogeneous Networks.* Springer Netherlands, 2009, S. 37–48

[FIT+09] Faroudja, A. ; Izeboudjen, N. ; Titri, S. ; Sahli, L. ; Louiz, F. ; Lazib, D.: Hardware/Software development of a System on Chip platform for VoIP application. In: *Microelectronics (ICM), 2009 International Conference on*, 2009, S. 62 –65

[FK09] Frank Kesel, Ruben B. *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs.* 2009

[G.09] G., Gajski D.D. Abdi S. Gerstlauer A. S. *Embedded System Design: Modeling, Synthesis and Verification.* 2009

[Gin03] Ginosar, R.: Fourteen ways to fool your synchronizer. In: *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, 2003, S. 89 – 96

[GL08] Grant Likely, Josh B.: A Symphony of Flavours: Using the device tree to describe embedded hardware. In: *Linux Symposium 2008. Proceedings on*, 2008, S. 27 –37

[GR05] Gantz, John ; Rochester, Jack B. *Pirates of the Digital Millennium.* 2005

[Gra04] Gracely., Davidson Jonathan James Peters Jim Peters B. *H.323. Voice over IP fundamentals.* 2004

[Gra05] Gray, R.M.: The 1974 origins of VoIP. In: *Signal Processing Magazine, IEEE* 22 (2005), July, Nr. 4, S. 87–90

[Gra10] Grant, Ian A. Glover Peter M. *Digital Communications - 3rd ed.* 2010

[HTL+03] Ho, C.C. ; Tang, Tzi-Chiang ; Lee, Chin-Ho ; Chen, Chih-Ming ; Tu, Hsin-Yang ; Wu, Chin-Sung ; Chang, Chao-Hsi ; Huang, Chin-Meng: H.323 VoIP telephone implementation embedding a low-power SOC processor. In: *Electron Devices and Solid-State Circuits, 2003 IEEE Conference on*, 2003, S. 163 – 166

[ISO] ISO. *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, ISO/IEC 7498-1:1994(E)*

[JC05] Jonathan Corbet, Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* 2005

[JF10] Jijiang, Yu ; Fiyu, Lian: Design and implementation of an embedded VoIP system using Bluetooth technique. In: *Future Computer and Communication (ICFCC), 2010 2nd International Conference on* Bd. 3, 2010, S. V3–344 –V3–347

[JW05] Jerraya, A.A. ; Wolf, W.: Hardware/software interface codesign for embedded systems. In: *Computer* 38 (2005), February, Nr. 2, S. 63 – 69

[KT01] Karam, M.J. ; Tobagi, F.A.: Analysis of the delay and jitter of voice traffic over the Internet. In: *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* Bd. 2, 2001, S. 824 –833

[LM00] Liu, Hong ; Mouchtaris, P.: Voice over IP signaling: H.323 and beyond. In: *Communications Magazine, IEEE* 38 (2000), October, Nr. 10, S. 142 –148

[OS00] Olshefski, Weibin Zhao D. ; Schulzrinne, Henning. *Internet Quality of Service: an Overview. Columbia Technical.* February 2000

[RNH99] Raffaele Noro, Maher H. ; Hubaux, Jean-Pierre. *CIRCUIT EMULATION OVER IP NETWORKS - Technical Report.* 1999

[SA05] Saint-Andre, Peter: Streaming XML with Jabber/XMPP. In: *IEEE Internet Computing* 9 (2005), S. 82–89. – ISSN 1089–7801

[SA07] Saint-Andre, P.: Jingle: Jabber Does Multimedia. In: *Multimedia, IEEE* 14 (2007), January - March, Nr. 1, S. 90 –94

[Sal08] Sally, Gene: System Minimization. (2008), February, Nr. 166

[SHR99] Seo, Gun ; Hwang, Woo-Chang ; Rhee, Youngok: An implementation of VoIP cable modem. In: *TENCON 99. Proceedings of the IEEE Region 10 Conference* Bd. 2, 1999, S. 1532 –1535

[SK05] Sun, Dong G. ; Kim, Sung J.: A Kernel-Level RTP for Efficient Support of Multimedia Service on Embedded Systems. In: *Computational Science and Its Applications - ICCSA 2005* Bd. 3482. Springer Berlin / Heidelberg, 2005, S. 79–88

[SR00] Schulzrinne, H. ; Rosenberg, J.: The Session Initiation Protocol: Internet-centric signaling. In: *Communications Magazine, IEEE* 38 (2000), October, Nr. 10, S. 134 –141

[TAK06] Tong, J.G. ; Anderson, I.D.L. ; Khalid, M.A.S.: Soft-Core Processors for Embedded Systems. In: *Microelectronics, 2006. ICM '06. International Conference on*, 2006, S. 170 –173

[TCTR05] Toral-Cruz, H. ; Torres-Roman, D.: Traffic analysis for IP telephony. In: *Electrical and Electronics Engineering, 2005 2nd International Conference on*, 2005, S. 136 – 139

[VB01] Vlaovic, B. ; Brezocnik, Z.: Packet based telephony. In: *EUROCON'2001, Trends in Communications, International Conference on.* Bd. 1, 2001, S. 210 –213 vol.1

[VSMH02] Varshney, Upkar ; Snow, Andy ; McGivern, Matt ; Howard, Christi: Voice over IP. In: *Commun. ACM* 45 (2002), January, S. 89–96

[Wel00] Wells, Nicholas: BusyBox: A Swiss Army Knife for Linux. (2000), October, Nr. 78

[XJX10] Xuejing, Zhang ; Jinping, Li ; Xiulan, Hao: Scheduling and Control of VOIP Video Telephone under DSP/BIOS. In: *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on* Bd. 2, 2010, S. 726 –729

[ZW10] Zhai, Wenbo ; Wang, Jian: An application of VoIP communication on embedded system. In: *Computer Application and System Modeling (ICCASM), 2010 International Conference on* Bd. 4, 2010, S. V4–619 –V4–623

Internet References

[45] Autotools - Autoconf. http://www.gnu.org/software/autoconf/ - 04/11/2011.
[46] Autotools - Automake. http://www.gnu.org/software/automake/ - 04/11/2011.
[47] Bitbake. http://developer.berlios.de/projects/bitbake - 02/13/2011.
[48] Buildroot. http://buildroot.uclibc.org/ - 02/13/2011.
[49] ccRTP. http://www.gnu.org/software/ccrtp/ - 02/17/2011.
[50] Crosstool. http://kegel.com/crosstool/ - 02/13/2011.
[51] Ekiga - Gnome softphone. http://www.ekiga.org - 01/14/2011.
[52] Embedded Development Kit. http://www.denx.de/wiki/DULG/ELDK - 02/13/2011.
[53] Emdebian. http://www.emdebian.org/ - 02/13/2011.
[54] FHS - Linux Filesystem Hierarchy. http://www.pathname.com/fhs/ - 02/28/2011.
[55] fork, Linux Manpages, Section 2. http://linux.die.net/man/2/fork - 02/27/2011.
[56] FreeSwitch. http://www.freeswitch.org - 02/17/2011.
[57] Gnomemeeting. http://www.gnomemeeting.org - 02/17/2011.
[58] H323plus. http://sourceforge.net/projects/h323plus/ - 02/17/2011.
[59] Kphone - KDE softphone. http://sourceforge.net/projects/kphone/ - 01/14/2011.
[60] libeXosip. http://www.antisip.com/as/ - 02/17/2011.
[61] libJingle. http://code.google.com/apis/talk/libjingle/index.html - 02/17/2011.
[62] Linphone - softphone development kit. http://www.linphone.org/ - 01/14/2011.
[63] Linux From Srcatch. http://www.linuxfromscratch.org/ - 02/13/2011.
[64] Linux Kernel Archives. http://www.kernel.org/pub/linux/kernel/ - 01/31/2011.
[65] Linux Signal Manpage 7. http://linux.die.net/man/7/signal - 01/28/2011.
[66] ooH323c. http://www.obj-sys.com/telephony-objective.shtml - 02/17/2011.
[67] Openembedded. http://www.openembedded.org/index.php/Main_Page - 02/13/2011.
[68] oRTP. http://www.linphone.org/eng/download/git.html - 02/17/2011.
[69] PJLIB. http://www.pjsip.org - 02/17/2011.
[70] PPCBoot Bootloader. http://ppcboot.sourceforge.net/ - 02/13/2011.
[71] Questasim by Mentor Graphics. http://www.mentor.com - 02/24/2011.
[72] SIP parameter list. http://www.iana.org/assignments/sip-parameters - 12/13/2010.
[73] Sykpe. http://www.skype.com/intl/de/home/ - 03/27/2011.
[74] U-Boot Bootloader. http://www.denx.de/wiki/U-Boot/WebHome - 02/13/2011.
[75] uCLinux. http://www.uclinux.org/description/ - 01/21/2011.
[76] UEFI. http://www.uefi.org/home/ - 01/21/2011.
[77] Wireshark Packet Sniffer. http://www.wireshark.org - 02/24/2011.
[78] Xilinx ISE. http://www.xilinx.com/tools/designtools.htm - 02/01/2011.

[79] XILINX Linux repository. http://git.xilinx.com/?p=linux-2.6-xlnx.git;a=summary - 02/14/2011.

[80] XILINX U-Boot repository. http://git.xilinx.com/?p=u-boot-xlnx.git;a=summary - 02/13/2011.

[81] Xlite - non-free softphone. http://counterpath.com/x-lite.html&active=4 - 01/14/2011.

[82] Danny Cohen. SPECIFICATIONS FOR THE NETWORK VOICE PROTOCOL (NVP). RFC 741 (Standard Track) http://tools.ietf.org/html/rfc741 - 03/27/2011, January 1976.

[83] DARPA INTERNET PROGRAM. INTERNET PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION. RFC 791 (Standard Track) http://tools.ietf.org/html/rfc791 - 01/23/2011, September 1981.

[84] DARPA Internet Program. TRANSMISSION CONTROL PROTOCOL. RFC 793 (Standard Track) http://tools.ietf.org/html/rfc793 - 12/03/2010, September 1981.

[85] H. Schulzrinne A. Rao R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Standard Track) http://tools.ietf.org/html/rfc2326 - 12/13/2010, April 1998.

[86] H. Schulzrinne S. Casner R. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551 (Standard Track) http://tools.ietf.org/html/rfc3551 - 12/03/2010, July 2003.

[87] H. Schulzrinne S. Casner R. Frederick V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard Track) http://tools.ietf.org/html/rfc3550 - 12/03/2010, July 2003.

[88] IEEE. Virtual Bridged Local Area Network. http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf - 01/23/2011, 2006.

[89] IEEE. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. http://standards.ieee.org/about/get/802/802.3.html - 01/23/2011, 2008.

[90] Intel. Audio Codec '97. http://download.intel.com/support/motherboards/desktop/sb/ac97_r23.pdf - 01/24/2011, April 2002.

[91] ITU-T. G.711.0 : Lossless compression of g.711 pulse code modulation. http://www.itu.int/rec/T-REC-G.711-198811-I/en - 12/03/2010, November 1988.

[92] ITU-T. Series q: Switching and signalling digital subscriber signalling system no. 1 - network layer. http://www.itu.int/rec/T-REC-Q.931-199805-I - 12/13/2010, May 1988.

[93] J. Rosenberg H. Schulzrinne G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3261 (Standard Track) http://tools.ietf.org/html/rfc3261 - 12/06/2010, June 2002.

[94] M. Handley C. Perkins E. Whelan. Session Announcement Protocol. RFC 2974 (Standard Track) http://tools.ietf.org/html/rfc2974 - 12/13/2010, March 2000.

[95] M. Handley H. Schulzrinne E. Schooler J. Rosenberg. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 2543 (Standard Track) http://tools.ietf.org/html/rfc2543 - 12/06/2010, March 1999.

[96] M. Handley V. Jacobson. SDP: Session Description Protocol. RFC 2327 (Standard Track) http://tools.ietf.org/html/rfc2327 - 12/13/2010, March 1998.

[97] M. Handley V. Jacobson C. Perkins. SDP: Session Description Protocol. RFC 4566 (Standard Track) http://tools.ietf.org/html/rfc2543 - 12/13/2010, March 2006.

[98] M. Spencer B. Capouch E. Guy, Ed. F. Miller K. Shumard. IAX: Inter-Asterisk eXchange Version 2. RFC 5456 (Standard Track) http://tools.ietf.org/html/rfc5456 - 04/03/2011, February 2010.

[99] J. Postel. User Datagram Protocol. RFC 768 (Standard Track) `http://tools.ietf.org/html/rfc768` - 12/03/2010, August 1980.

[100] R. Stewart Q. Xie K. Morneault C. Sharp H. Schwarzbauer T. Taylor I. Rytina M. Kalla L. Zhang V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Standard Track) `http://tools.ietf.org/html/rfc2960` - 12/03/2010, October 2000.

[101] S. Deering R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Standard Track) `http://tools.ietf.org/html/rfc2460` - 01/23/2011, December 1998.

[102] Xilinx Inc. PLB IPIF. `http://www.xilinx.com/support/documentation/ip_documentation/plb_ipif.pdf` - 02/04/2011, April 2005.

[103] Xilinx Inc. Xilkernel. `http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf` - 01/21/2011, December 2006.

[104] Xilinx Inc. Power PC Processor Reference Guide. `http://www.xilinx.com/support/documentation/user_guides/ug011.pdf` - 01/21/2011, January 2007.

[105] Xilinx Inc. Microblaze Processor Reference Guide. `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf` - 01/21/2011, January 2008.

[106] Xilinx Inc. ML405 Evaluation Platform. `http://www.xilinx.com/support/documentation/boards_and_kits/ug210.pdf` - 01/18/2011, March 2008.

[107] Xilinx Inc. PowerPC 405 Block Reference Guide. `http://www.xilinx.com/support/documentation/user_guides/ug018.pdf` - 02/03/2011, May 2008.

[108] Xilinx Inc. Virtex-4 FPGA User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug070.pdf` - 01/18/2011, December 2008.

[109] Xilinx Inc. PowerPC 405 Processor Block Reference Guide. `http://www.xilinx.com/support/documentation/user_guides/ug018.pdf` - 01/18/2011, January 2010.

[110] Xilinx Inc. Virtex-4 FPGA Embedded Tri-Mode Ethernet MAC. `http://www.xilinx.com/support/documentation/user_guides/ug074.pdf` - 01/24/2011, February 2010.