Jül-4389

Zentralinstitut für Engineering, Elektronik und Analytik (ZEA)
Systeme der Elektronik (ZEA-2)

Raspberry Pi Based IEEE 1451.1 Network for Distributed Actuation Control

Lileesha Mulpuri

JÜLICH
FORSCHUNGSZENTRUM

Raspberry Pi Based IEEE 1451.1 Network for Distributed Actuation Control

Lileesha Mulpuri

Berichte des Forschungszentrums Jülich; 4389
ISSN 0944-2952
Zentralinstitut für Engineering, Elektronik und Analytik (ZEA)
Systeme der Elektronik (ZEA-2)
Jül-4389

(Master, Hochschule Bremen, 2015)

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH
Zentralbibliothek, Verlag
52425 Jülich
Tel.: +49 2461 61-5220
Fax: +49 2461 61-6103
E-Mail: zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

ABSTRACT

The DFG funded project FOR 1779 titled “active drag reduction via transversal surface waves”, investigates robust methods to reduce the friction drag by influencing the turbulent boundary layer. The Central Institute of Engineering, Electronics and Analytics, ZEA-2: Electronic Systems, Forschungszentrum Jülich GmbH, works on the subproject “development of a real-time actuator and sensor network” for closed loop controlled transversal surface waves. For application on transportation vehicles like airplanes a large scale real-time actuator and sensor network is needed. To investigate the configuration of such a network a model based on Simulink and TrueTime is established. A Raspberry Pi based test bed is then used for parameter verification of the model.

The aim of the thesis is to steer an actuator and to record values from a sensor in a small scale distributed actuator and sensor network with the help of IEEE 1451.1 Smart Transducer Interface Standard Protocol. For this purpose Raspberry Pi network consists of three Raspberry Pi nodes is established in order to send actuation signal, to gather sensor parameters and to provide actuation parameters. The benefit for the model will be to get the propagation time through IEEE1451.1 layer out of this real world test bed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Motivation

The drag of transportation systems such as airplanes, ships and trains is investigated with respect to the friction drag. This drag has a major contribution towards the fuel consumption in modern transportation systems, which in turn increases the cost. Cost reduction became the element of concentration in the present day contest. And apparently to reduce the cost, friction drag has to be reduced. The earlier approach to reduce the drag is by stabilizing the laminar state of the boundary layer flow, as the wall shear stress in a laminar is comparatively smaller than in the turbulent boundary layer [1]. The idea of the research project FOR1779 is to reduce the friction drag by decreasing the wall shear stress of turbulent boundary layer by damping the near wall coherent structures with minimum energy input into the flow. This approach requires additional weight in the transport system. So active drag reduction in high Reynolds numbers ($>10^4$ the typical range of airplanes), by span-wise transversal surface waves is investigated to reduce the fuel consumption and also noise.

Later on flow control development based on wind tunnel and numerical studies in order to create a closed loop controlled transversal waves on a surface like an airplane wing, a large scale actuator and sensor network is needed. For the development of real-time actuator and sensor network a model based on Simulink and TrueTime has been established. To ensure the accuracy for the network development, the network parameters have to be verified in a real world test bed. For this purpose a Raspberry Pi based test bed is used in order to determine the network and transmission parameters for the distributed actuation control. With the help of this approach in later stages a link will be created between the large scale model and later microcontroller based real-time actuator and sensor network for distributed active turbulent flow control.

1.2 Project description

The FOR1779 develops robust methods to reduce the wall shear stress on the turbulent boundary layer. It is a DFG funded project, started with the aim of “Drag reduction via transversal surface waves”. It consists of seven subprojects, where each project works on a special aim. In later application for drag reduction large scale distributed actuator and sensor network is required to implement on air planes.

The Forschungszentrum Jülich GmbH, Central Institute for Engineering, Electronics and Analytics, ZEA-2: Electronic Systems is responsible to handle the sub-project TP4 titled as "Development of a Real Time Actuator and Sensor Network". For the development of real-time actuator and sensor network a model based on Simulink and TrueTime has been established. It provides interfaces to the central flow control and actuation control in a cascade control loop and can be used in the wind tunnel as model in the loop simulation [2]. In order to verify the network parameters in a real world a Raspberry Pi network based test bed is used. The Raspberry Pi based test bed is proposed to steer the actuator and to calculate the timing parameters for a large scale distributed actuator and sensor network.

1.3 Aim of the Thesis

The main focus of the thesis is to implement IEEE 1451.1 Smart Transducer Interface Standard Protocol on the Raspberry Pi test bed network. As shown in the Figure 1.1 three Raspberry Pis will be used to communicate over a network and to send and receive signals through an analog interface. One Raspberry Pi acts as a controller and sends commands to the other two Raspberry Pis. All these Raspberry Pis communicate with each other using IEEE 1451.1 protocol. The aoNCAP as shown in the Figure 1.1 is responsible to send an analog sine wave to steer the actuator and the aiNCAP is responsible to receive the sensor values from the sensor. A sound card as shown in the Figure 1.1 acts as a transducer interface module (TIM) between the Raspberry Pis and actuator and sensor.

The aoNCAP receives a command from JNCAP and starts sending the analog sine wave to the actuator. Similarly, the aiNCAP receives the command from JNCAP and starts recording the sensor values from the sensor and then they are visualised on the JNCAP application window.

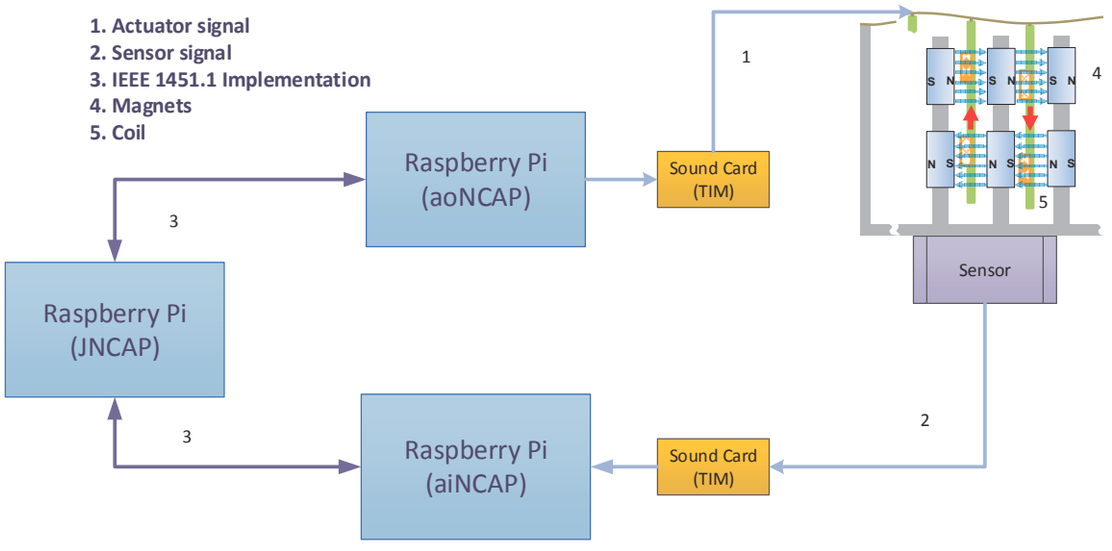

Figure 1.1 The communication process within the Raspberry Pi network test bed following the IEEE 1451.1 protocol standard. The aiNCAP is supposed to record sensor values proportional to the actuator amplitude from analog input using a sound card. The aoNCAP is supposed to send an analog sine wave to drive the current within the coil of the electromagnetic actuator, i.e. to steer the actuator. This process should be initiated and monitored by JNCAP.

After implementing the protocol, the final task is to measure the propagation time through the IEEE 1451.1 layer and to show that the loop for actuation control is closed by actuating and measuring in parallel.

2 Background

This chapter focuses on the network model which has been proposed to use in the sub-project. The reason for calculating the propagation time through the IEEE layer is shortly described.

2.1 Network model Protocol Stack

The network protocol stack referring to the ISO-OSI layer, three layer architecture consists of physical layer, transport layer and application layer is used (see Figure 2.1). The communication takes place between NCAPs and JNCAP as shown in the Figure 2.1. The IEEE 1451.1 Smart Transducer Interface Standard protocol is implemented on the application layer. The Ethernet acts as a physical layer, ACE library acts like a Transport layer and on application layer IEEE 1451.1 is implemented. The timing values will be calculated through the IEEE 1451.1 application layer.

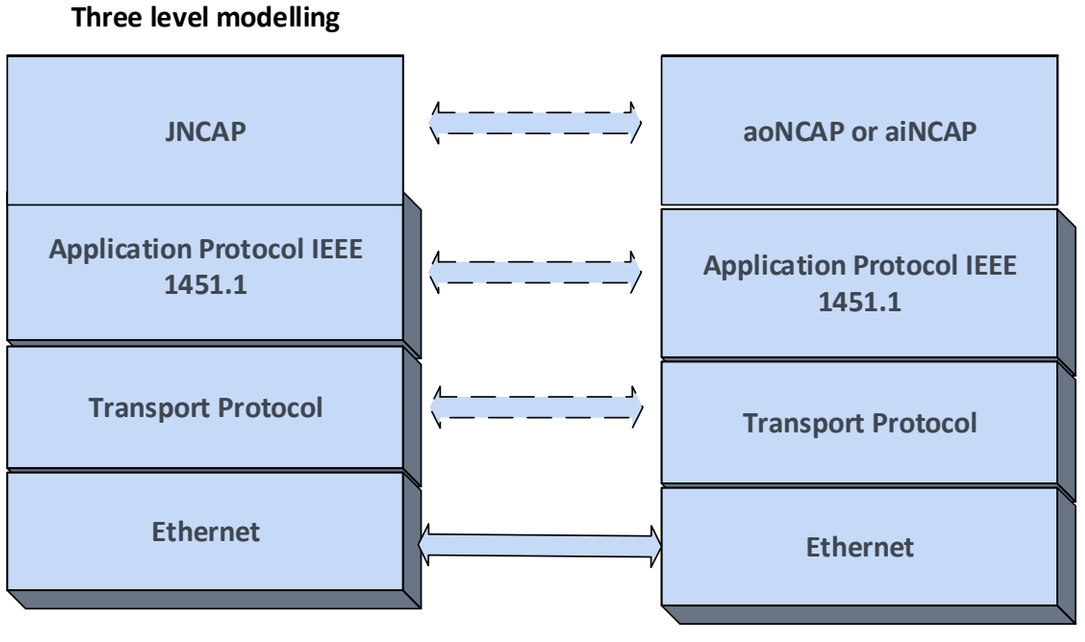

Figure 2.1 The network protocol stack explaining the communication flow. On the bottom layer a physical communication takes places, while on the upper layer a logical communication runs [3].

When JNCAP starts sending commands to NCAPs i.e. aiNCAP and aoNCAP, the application layer on which IEEE 1451.1 is implemented starts measuring the timing values. The measured timing values will be later imported to the Simulink and TrueTime model which is the overall sub project.

3 Development Environment

The details of the hardware and software used in this project are elaborated in this chapter. To start with, the technical detail of the Raspberry Pi which is the most important element of the hardware is described. Then, in later sections a brief description about the sound card which is the external hardware of the setup is given. It acts as ADC and DAC in this whole setup. The process is explained in the later stages of the chapter. In the last section a brief description about the software platforms, programming environment, example implementation and about open source platform called Adaptive Communication Environment (ACE) is given.

3.1 Raspberry Pi

Raspberry Pi is a small, single board computer with I/O connectors for peripheral interfacing. It is basically a credit card sized computer, invented by the Raspberry Pi foundation which is an educational charity group headed by the United Kingdom (UK) [4]. They invented this Raspberry Pi with a motivation to advance the education for children in the field of computer sciences. Raspberry Pi works on the software called Raspbian, which is a free operating system based on Debian Linux. The Raspberry Pi provides multipurpose utility such as programming, controlling robots, used for modeling different working modules for various applications e.g. automation applications like home theatre. Different devices can be connected to a Raspberry Pi through USB ports and a HDMI port. A typical Raspberry Pi used in this project is shown in the Figure 3.1 [4].

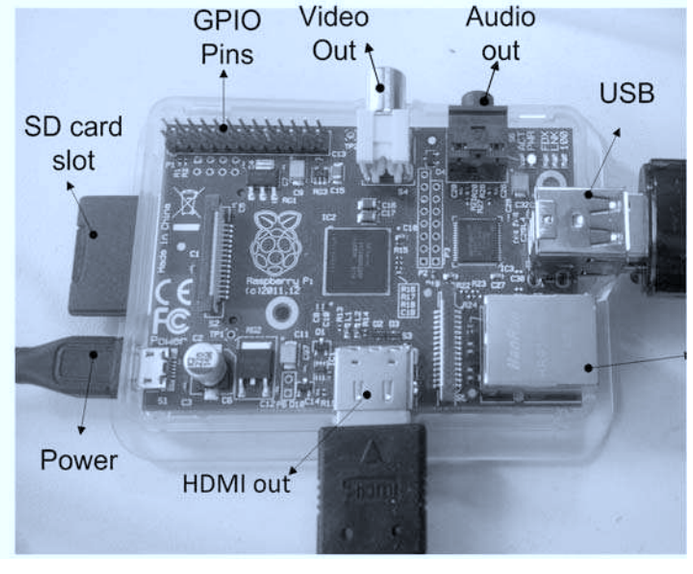

Figure 3.1 Raspberry Pi credit card sized computer with multiple I/O ports

3.2 DAQ ADC and DAC

Raspberry Pi needs some external hardware to receive and send analog data to and from the actuator and sensor. It needs an external ADC (analog to digital converter) and DAC (digital to analog converter) to receive and send analog signals. It can be done using the GPIO pins but it is time consuming and complicated. The hardware used in the project as transducer interface module (TIM) is a sound card. Transducer interface module is a part of IEEE 1451.1 standard and contains the signal conditioning, analog to digital conversion and control logic necessary to acquire a signal and convert it to data. The TIM holds the transducer electronic data sheet (TEDS) which represents the information about the sensors and actuators attached to a TIM [11]. It acts like an interfacing module between Raspberry Pis and Actuator and sensor. It uses the ALSA drivers to process the data which is explained more clearly in section 3.3.1. By using a sound card the complexity and cost reduces. It is easy to interface the sound card to Raspberry Pi as it does not need any extra effort.

3.2.1 Sound card

The sound card used in the project is Creative Sound Blaster Play [12]. It is compatible and easy to use. It can be just plugged into Raspberry Pi using the USB port. The sound card used in this project is shown in the Figure 3.2. From the figure it can be observed that there are two sockets for audio in and audio out.

Figure 3.2 Creative USB sound card with analog in (Microphone) and analog out (Headphones) ports

Specifications

The specifications of the Creative sound card are as follows [13]:

- Playback: USB 1.1 : Stereo/ Surround = 16-bit/48kHz

- Signal to noise ratio: >90dB
- Recording: 16-bit/48kHz
- Microphone : 3.5mm mini jack located at the bottom of the USB card
- Compact plug and play
- PC USB bus-powered : 400mA required
- Full scale input level : 120m Vrms
- Input impedance : 2.2 kΩ
- Line out: 0.8 Vrms at 10kΩ

3.3 Software

In this section a brief explanation about all the software applications, programming environments and software implementations is given.

3.3.1 ALSA drivers

ALSA means Advanced Linux Sound Architecture. It consists of a set of kernel drivers, an application programming interface (API) library and utility programs for supporting sound under Linux. Some of the functions of the ALSA project were automatic configuration of sound card hardware and handling of multiple sound devices in a system [14]. The library provides higher level and developer friendly programming interfaces. It also provides a logical naming of devices in order to give some information to developers about low-level details such as device files. The naming could be done using the format *hw: i, j*, where *'i'* is the sound card number and *'j'* is the device on the card.

ALSA has a capability called plugins which allows extension to new devices including virtual devices implemented entirely in software. It provides multiple number of command-line utilities, including a mixer to control input and output volume, sound file player and tools for controlling special features of specific sound cards.

ALSA Architecture

The ALSA API has interfaces like Control interface, PCM interface, Raw-Midi interface, Timer interface, Sequencer interface and mixer interface [14]. The PCM interface is used for ALSA programming in this project.

PCM interface: This is the interface used for the digital audio and capture. The PCM middle layer of ALSA is powerful. It is only necessary for each driver to implement the low-level

functions to access its hardware. In order to access the PCM layer, `#include <sound/pcm.h>` should be included in C-programming.

In addition to this, `include <sound/pcm_params.h>` might be needed if the user access to some functions related to `hw_param`. Each card device can have up to four PCM instances A PCM instance corresponds to a PCM device file. The limitation of number of instances comes only from the available bit size of the Linux device numbers, when 64bit device number is used then there will be more PCM instances. A PCM instance consists of PCM playback and capture streams and each PCM stream consists of one or more PCM sub-streams. Some sound cards support multiple playback functions.

3.4 IEEE 1451 example implementation from source forge

Theoretical, as well as practical investigations have been done to get knowledge in the IEEE 1451 Smart Transducer Interface Standard. As the standard is open the Open Gaithersburg IEEE 1451 example implementation could be used [20], (see section 7.1). For practical investigations of the actuator and sensor network implementation, the example implementation from source forge called “an open implementation of IEEE 1451.1” is proposed to use. It contains various reference implementations for users. These open implementations are invented by the NIST organization [21].

The recent reference implementation from NIST is IEEE 1451 Open Gaithersburg implementation. This is an implementation for smart transducer interfacing of sensors. It is based on NCAP (Network capable application processor) to NCAP communication. There is clear explanation about NCAP to NCAP communication in section 5.3. It can be tested on wired and wireless networks.

IEEE 1451 open Gaithersburg implementation consists of NIST C++ and Java reference implementations. The C++ reference implementation uses the open source Adaptive Communication Environment (ACE) (see section 3.5). The entity class in the IEEE 1451.1 inherits the properties from the ACE and helps in synchronization and TCP/IP communication of NCAPs.

3.5 Adaptive Communication Environment (ACE)

ACE is open source software available for object oriented framework to implement software portable real-time communication patterns. These communication software patterns are

implemented in C++. ACE provides reusable C++ wrappers and components which can be used to perform common communication tasks across many OS platforms [22].

ACE is developed with a target of providing high performance, real-time communication services and applications. It has a special feature of automating the system configuration and reconfiguration by means of dynamically linking the services into applications.

ACE provides many communication software tasks such as event de-multiplexing and event handler dispatching, signal handling, service initializations, interprocess communication, message routing, shared memory management. ACE library plays an important role in the development of IEEE 1451.1 communication patterns, concurrency and all core distribution, which can be done by NIST.

4 Concept

This chapter gives an idea of dealing the project and assumptions that have been made before implementing the task. In the first section there is a description about the hardware setup of Raspberry Pi and in the later part about the IEEE 1451 standard. Then signal generation using the analog actuator and sensor connection is described. Data processing, Time measurement and Experimental validation are explained in the last sections

4.1 Raspberry Pi Hardware setup

The Raspberry Pi hardware setup as shown in the Figure 1.1, consists of three Raspberry's. All these Raspberry Pis will be connected using a network switch [24] in order to create a closed network. This network switch can connect up to eight Raspberry Pis. When the connection is made user should be able to communicate between Raspberry Pi nodes, in order to do this a static IP address has to be created for each Raspberry Pi [25].

One Raspberry Pi acts like a control unit and two Raspberry Pis follows the instructions from the main node. The control node in the setup is JNCAP and other two nodes are aiNCAP and aoNCAP. The aiNCAP and aoNCAP receives commands from JNCAP. The aoNCAP sends analog signal to actuator and the aiNCAP receives the sensor values from the sensor. For the loop as shown in Figure 1.1 sound card acts a medium between the Raspberry Pis and actuator and sensor network. An analog signal from aoNCAP will be send to actuator using the sound card and for aiNCAP sound card acts as a medium to receive sensor values from sensor.

4.2 IEEE 1451

IEEE 1451 is a standard for communication between actuators and sensors. In particular IEEE 1451.1 standard is chosen as a protocol, as it works on NCAP communication for sensor and actuators. An example implementation of IEEE 1451 temperature program from NIST is also available, which will be used as a reference implementation. Based on the reference implementation communication will be implemented on Raspberry Pi network. The communication between NCAPs is implemented through the network interface as shown in Figure 4.1. The JNCAP and NCAP communicates through a network interface i.e. through Client/Server or Publish/Subscribe communication model as explained in section 5.3.3. The NCAP is acts like a card cage and all the blocks are plugged into the central part as shown in Figure 4.1. The function block application code is plugged into the NCAP and can be used

when ever needed. The NCAP and the JNCAP communicates over a network interface or over network ports as depicted in the Figure 4.1.

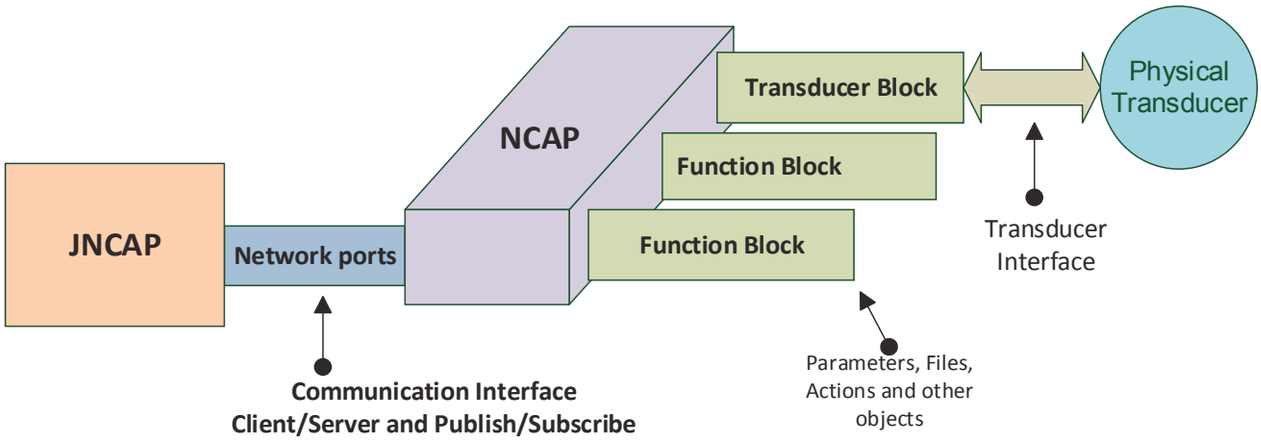

Figure 4.1 NCAP communication paradigm showing the blocks and interfacing between the transducer and function blocks. The Transducer block receives information from the physical transducer and the function block below contains the files which are needed for the NCAP communication [4]

The transducer block as shown receives the data from physical transducer or Transducer interface module (TIM).

4.3 Analog actuator and sensor connection

This is a hardware implementation part where signal generation and receiving is explained. An analog signal is used to steer the actuator. In order to do this a sine wave will be generated and sent to the actuator using a sound card. Similarly with the help of sound card Raspberry Pi will be recording the sensor values from the sensor.

The data processing will be done using the software module in order to extract the amplitude values from the signal. When aiNCAP receives sensor values from the sensor, it tries to extract the amplitude values from the sensor values.

4.4 Time measurement

Time measurement will be taken in order to measure the propagation time through IEEE layer. It is assumed to measure the time during discovering the NCAPs, before and after send and during publishing.

4.5 Experimental Validation

The experimental validation for implemented model will be done by using the signal generator and MacLab. The aoNCAP which gives an analog output signal will be measured

on MacLab. When it puts out the analog sinewave it can be visualized by connecting the sound card to the MacLab. The aiNCAP upon receiving the command from JNCAP, it displays the sensor values on JNCAP application window. It can be validated by connecting the sound card to the function generator and there by checking the values by changing the peak to peak voltage values on function generator. Then the change in the values can be visualized on the output Java application window.

Time measurement validation will be done by finding the places where the command enters and leaves the IEEE 1451 routines by executing the program line by line. Statistical analysis will be done afterwards offline.

5 IEEE 1451 standard protocol family

This chapter gives an overview of the IEEE 1451 smart transducer standard protocol family. To start with, the introduction of the standard protocol family is briefed and in later sections a detailed description of IEEE 1451.1 blocks such as NCAP, transducer block, and functional block are given. The communication path between NCAPs is also explained at the end of this chapter.

5.1 Introduction

The IEEE 1451 family of standard is developed by the Institute of Electrical and Electronics Engineers. This standard specially describes the set of common, network-independent communication interfaces for connecting sensors and actuators to microprocessors, instrumentation systems and networks [26].

There are various applications of the IEEE 1451 standard. The main applications are based on the advantages of IEEE 1451 such as plug and play capability, wide area data collection ability, multiple sensors on one network, automatic testing and many more. Having the ability to support multiple networks and transducer families in a cost effective way IEEE 1451 serves a wide range of industrial needs. It has many operating modes and also it is compatible with both wired and wireless sensor buses and networks. It simplifies the connectivity and maintenance of transducers via TEDS to device networks. It has extensive units, linearization and calibration options and also it has an efficient binary protocol which is most suitable for wireless networks. It is capable of handling multiple timing and data block size constraints [26].

There are seven different standards which are included in the IEEE 1451 family. Each standard is application specific. The standard includes [26]:

1451.0(2007) - IEEE standard for smart transducer interface for sensors and actuators which includes common functions, communication protocols, and Transducer Electronic Data Sheet (TEDS) Formats.

1451.1(1999) - IEEE standard for Smart Transducer Interface for Sensors and Actuators which includes Network Capable Application Processor Information Model.

1451.2(1997) - IEEE standard for Smart Transducer Interface for Sensors and Actuators which includes Transducer to Microprocessor Communication Protocols & TEDS formats.

1451.3(2003) - IEEE standard for Smart Transducer Interface for Sensors and Actuators which includes Digital Communications & TEDS formats for Distributed Multidrop Systems.

1451.4(2004) - IEEE standard for Smart Transducer Interface for Sensors and Actuators which includes Mixed mode Communication Protocols & TEDS formats.

1451.5(2007) - IEEE standard for Smart Transducer Interface for Sensors and Actuators which includes Wireless Communication Protocols & Transducer Electronic Data Sheet (TEDS) formats.

1451.7(2010) - IEEE standard for smart transducer interface for sensors and actuators which includes transducers to Radio Frequency Identification(RFID) Systems Communication Protocols and Transducer Electronic Data Sheet Formats.

After the intensive research work, it is decided to use IEEE 1451.1 standard, as it is usable for NCAP to NCAP communication which are represented by Raspberry Pi nodes in the network. This part of the IEEE 1451 standard is described in detail in the following chapter.

5.2 IEEE 1451.1

The IEEE 1451.1 standard provides Network Capable Application Processor (NCAP) Object Model for smart transducer interfacing of sensors and actuators in an Ethernet based actuator and sensor network [11]. IEEE 1451.1 standard is developed to provide a network-neutral application model which reduces the effort in interfacing sensors and actuators to a network. This standard defines an interface which connects the Network Capable Application Processors to control networks. This has been done with the development of a common control network information model for sensors and actuators.

There are several advantages in using this standard. The standard has a feature called *interoperability,* which makes the systems work together or to exchange information for all communication modes. It possess uniform design model for system implementation and also a network independent set of operations for system configuration. It defines network independent models for communication and for implementing application functionality. It also has portable application models. It has a network-independent layer, a uniform information model, and uniform models for managing event data, representing time, intranodes concurrency management, and for memory management.

5.3 NCAP: Network Capable Application Processor

The NCAP is defined by the hardware and software blocks. It acts like a bridge between transducers and communication network as shown in the figure 5.1.

In IEEE 1451.1 implementation, NCAP consists of three layers namely network layer, application layer and transducer layer. The network layer consists of the hardware required for the network library and also the 1451.1 API [11] and the application layer consists of software blocks which include the standard defined blocks and also application specific code. The transducer layer includes the hardware needed to communicate with the Transducer interface module (TIM), it also consists of the T-Block API.

The NCAP's hardware block should be designed in such a way that they meet the electrical and timing specifications of the network. Also the software blocks shall be designed using the formats and methods specified in the IEEE 1451.1 standard.

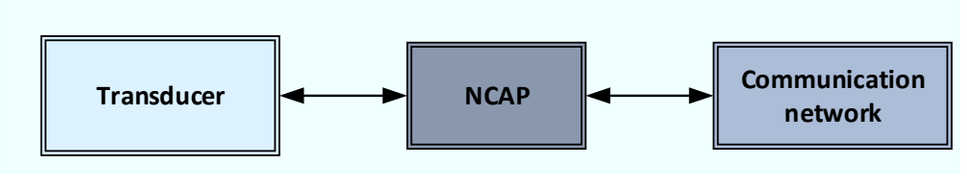

Figure 5.1 The Network Capable Application Processor (NCAP) connects the transducer module and communication network. The NCAP acts as a bridge between transducer and network [23]

The software architecture (Figure 5.2) of IEEE 1451.1 is classified into three models

- An Object model
- A Data model
- A Network communication model

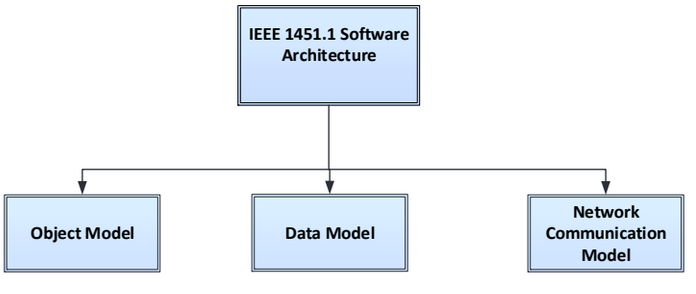

Figure 5.2 Software architecture of IEEE 1451.1 which is divided into three models such as object model, data model and network communication model concerning classes, data types and communication paradigms

5.3.1 An Object Model

The IEEE 1451.1 defines object model which defines the classes or abstract models with methods, attributes and its state behavior [27]. This is also known as Information model. The objects in the IEEE 1451.1 looks similar to other object oriented class definitions. It consists of instance data, other classes, code for implementing internal operations or methods, and state machine behavior. The Figure 5.3 shows how the object interfaces and the state behavior implemented using Object Model. It shows the interfacing between the distributed smart devices. The Object Model performs the functions like object discovery, invocation and synchronization using the attribute access and invocation operation [23].

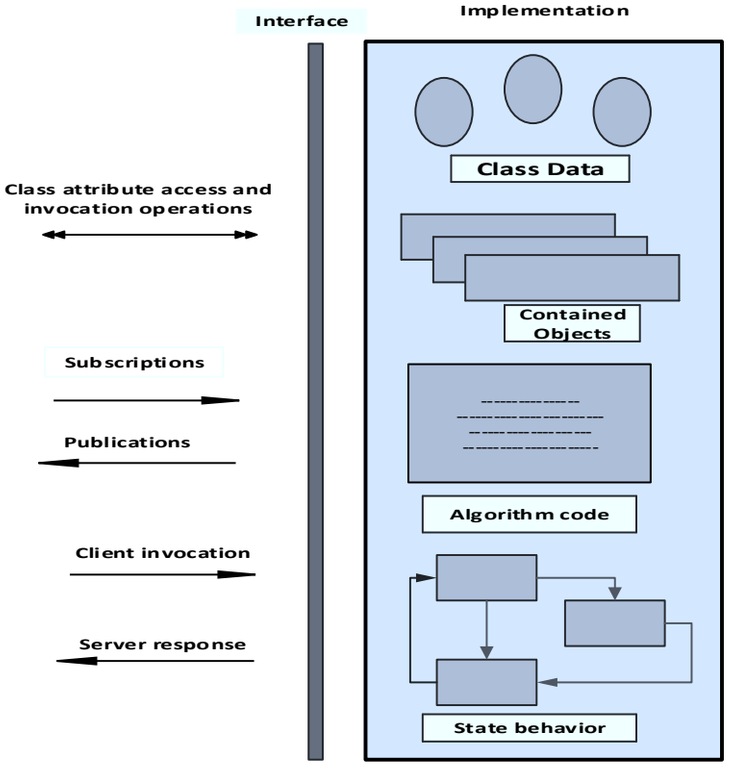

Figure 5.3 The Communication paradigm of Object class showing the functionality of the Object Model [23]

The Object Model consists of sub-classes, called block classes, component classes and service classes. As shown in the Figure 5.4, Block classes are used to perform processing, component classes are used to encapsulate data whereas service classes take care of inter-NCAP communications and system-wide synchronization. All these classes share common characteristics and entities like *object tag*, *object ID*, *object name*, *dispatch address*. Among all these classes block classes plays a key role. A brief description of the block classes can be seen in below sections.

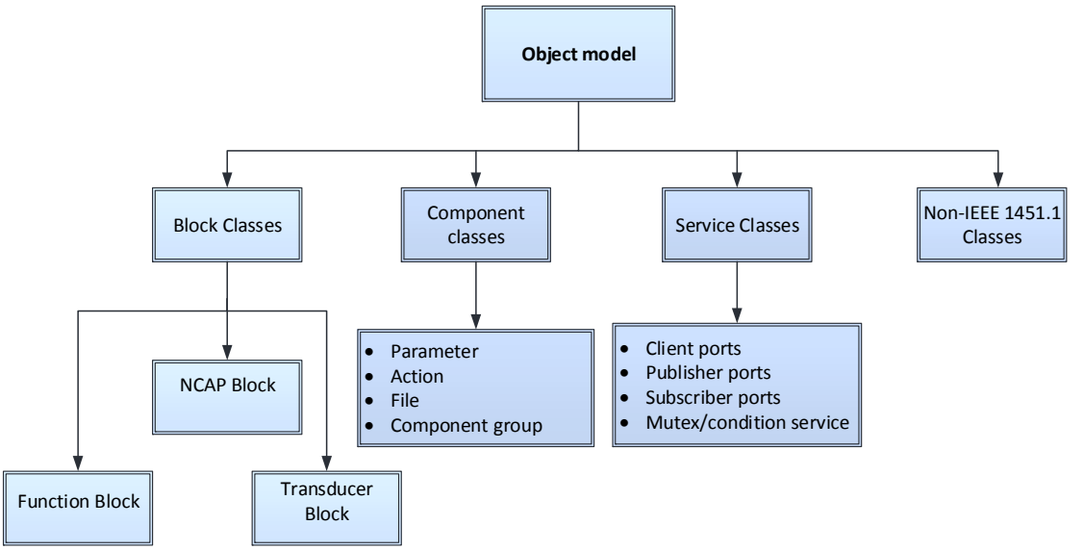

Figure 5.4 Classification of the Object Model with different classes and its subclasses in IEEE 1451.1 standard

Block classes

The block class is the root hierarchy for all the block objects and they are core for the APIs defined for the NCAP. Block classes are sub-divided into three different blocks named as NCAP block, function block and transducer block. These three blocks plays major role in NCAP communication. They work together to communicate to the physical world consisting of networks and transducers.

For network communication, these objects can be either network visible or independent. If the objects are network visible, they can be directly accessed by a network and in independent case the blocks can only be accessed by the other objects within the local NCAP. To make the object visible, it should be registered with its local NCAP block.

The behavior of the block class is controlled by a state machine with three different states namely *BL_UNITIALIZED*, *BL_INACTIVE* and *BL_ACTIVE*. The *BL_UNINITIALIZED* state is reserved for local activities such as bringing the block object into existence and performing local preparations needed for the block function [11]. The *BL_INACTIVE* state is reserved for activities such as the configuration of network communication properties of the block, for initialization, diagnosis and the maintenance of the block object. The *BL_ACTIVE* state is for the activities related to the normal application function of the block [11].

NCAP Block class

The NCAP Block has a standard software interface which supports network communication and system configuration. It keeps the information about network visible owned objects. It provides operations within an NCAP process to support block, service and component management. It also includes support for registration, deregistration, initialization, startup and shutdown.

In the Table 5.1 below a clear description of the important operations and their functionality of the NCAP block class is given [11].

Table 5.1 Table showing NCAP Block operations and its functionalities [11]

Operation	Functionality
GetNCAPBlockState	It is used to obtain the current state of the state machine
GetNCAPManufacturerID	It is used to identify the manufacturer of the NCAP
GetNCAPModelNumber	It returns an identifier which distinguishes different NCAP implementations.
GetNCAPSerialNumber	It distinguishes different instances of NCAP implementations.
GetNCAPOSVersion	It is used to specify the operating system that is in use.
GetClientPortProperties	It is used to get the ObjectTag of the NCAP and also the client port information
SetClientPortPropertiesBinidngs	It is used to initialize or modify the ObjectTag of the NCAP Block as well as the client port information.
IgnoreRequestNCAPBlockAnnouncement	It is used to ignore a publication that provides a notification of the existence of an NCAP Block object within the system.
RespondToRequestNCAPBlockAnnouncement	It is used to respond to a publication that provides a notification of the existence of an NCAP Block within the system.
RebootNCAPBlock	It is used to place the NCAP block and all

	the objects to their default power-on state.
ResetOwnedBlocks	It makes all the objects of NCAP block class to behave as if they received a rest operation.
GetBlockCookie	It is used to get the information of the Block cookie of the particular object that is being accessed.
PSK_NCAPBLOCK_GO_ACTIVE	It is used for the transitions within the state machine and the transition occurs from active to the initialized state.
RegisterObject and DeRegisterObject	These two operations are optional and will not be implemented
GoInactive	It is used for the transitions within the state machine.

The NCAP block divides the *BL_INACTIVE* state into two states called as *NB_INITILIAZED* and *NB_ERROR*. After implementing specific mechanisms in the NCAP block, the initial transition happens from the *BL_UNINITIALIZED* state to the *NB_INITIALIZED* state.

When there is any failure message in the state machine internally, then *NB_INITIALIZED* changes to *NB_ERROR* sub-state. The *GoInactive* operation causes the transition to the *NB_ERROR* sub-state if the NCAP detects any error.

Transducer Block class

The Transducer Block interfaces the transducers and application functions. It is the root for the class hierarchy of all transducer block objects in the family of transducers specified by IEEE 1451.1 standard. It establishes the mapping between the individual channels of the TIM transducers and the public transducers of the Transducer Block in the NCAP.

In the below Table 5.2 the important operations and their functionality has been described [11].

Table 5.2 Table showing the Transducer block operations and functionalities [11]

Operation	Functionality
GetCorrectionMode	It is used to obtain the current state of the state machine.
GetNumberOfTransducerChannels	It is used to access the Meta-TEDS such that it returns the number of transducers that are implemented in the TIM that is physically connected to NCAP.
GetMinimumSamplingPeriod	It is used to access the Meta-TEDS and then it returns the time in seconds of the minimum sampling rate of the TIM as a whole.
GetChannelParameterObjectChannelNumbers	It is used to return the physical interface channel numbers for the implemented physical interface channels.
GetUnrepresentedChannelNumber	This is used to return an array of numbers with each array representing a channel present at the physical interface.
UpdateAll	This causes a global trigger to be applied to the system.
EnableCorrections	This is used for the transition from the uncorrected to corrected state within the state machine
DisableCorrections	This is used for the transition from the corrected state to uncorrected state within the state machine.
GetLAstUpdateTimestamp and GetUpdateTimestampUncertainity	These are optional and will not be implemented.

The behaviour of the transducer block is controlled by the basic state machine defined for the block objects. The transducer block sub-states this state machine so that it can apply corrections to the transducer data [11]. The sub-states consist of *TB_CORRECTED* and *TB_UNCORRECTED* both in *BL_ACTIVE* and *BL_INACTIVE* states.

The Function Block class

The Function block class is root for the class hierarchy of the function block objects. It helps for the abstractions and packaging of application functionality and therefore the application specific objects are owned and controlled by the function block. Similarly, the function block allows the interaction between the application's objects and other standard defined objects.

In the Table 5.3, the operations and functionality of the functional block class has been mentioned [11].

Table 5.3 Table showing Functional block class operations and functionalities [11]

Operation	Functionality
GetFunctionBlockState	This is used to obtain the current state of the state machine for this object.
Start	This is used for transitions within the state machine for this block. It will be from the idle to the running state.
Clear	This is used for transitions within the state machine. The transition will be from the running to the idle state.
Pause	This is used for the transitions within the state machine of the block. The transition will be from the running to stopped state.
Resume	This is used for the transitions within the state machine for this block. This transition will be from the stopped to the running state.

The functional block's behavior is controlled by the inherited state machine of the block class. The *BL_ACTIVE* state is sub-stated to *FB_STOPPED, FB_RUNNING*, and *FB_IDLE.*

5.3.2 Data model

The Data model of IEEE 1451.1 defines various data types that are used for the object classes. It is a collection of primitive datatypes and structure datatypes. Whereas primitive data types are mapped to the programming language that will be used and the derived datatypes will be derived from the primitive data types [11].

Primitive data types

The primitive data types usually consist of various data types and arrays, few of them are mentioned below:

- A Boolean type
- An octet type
- Integer type
- Floating point type
- String type

Table 5.4 The simple primitive data types [11] used in IEEE 1451.1 standard

Data type	Default Value	Definition
Boolean	FALSE	TRUE or FALSE
Integer8	0	8-bit signed integer
UInteger8	0	8-bit unsigned integer
Integer16	0	16-bit integer
UInteger16	0	16-bit unsigned integer
Integer32	0	32-bit signed integer
UInteger32	0	32-bit unsigned integer
Integer64	0	64-bit signed integer
UInteger64	0	64-bit unsigned integer
Float32	+0.0	IEEE Std 754-1985 single-precision floating point number
Float64	+0.0	IEEE Std 754-1985 double-precision floating point number
Octet	All Bits set to 0	8-bit quantity not interpreted as a number

There is a brief description about the primitive data types in the Table 5.4. The string type is represented by a structure, in which it contains four fields which represent the character set, character code, language and the string data.

Derived data types

The derived data types are based on primitive data types. They are used in communication networks. An example of derived data type is *argument array* data type. Its application is

explained in section 7.2. Most of the application data in network communication theory of IEEE 1451.1 will be carried in arrays of argument [11].

5.3.3 Network communication model (NCAP – NCAP communication)

A Network Communication model is a paradigm for communicating information between NCAPs. It is an inter NCAP communication model. It supports two different models known as Client/Server network communication model and Publish/Subscribe network communication model. These models define the syntax and the semantics of the software interfaces between application objects and a communications network. The libraries consist of routines which allow the calls between the communication operations and interfaces. These libraries include marshalling and demarshaling routines for transforming the IEEE 1451.1 formats to and from their wire format [11]. Marshalling is the process of converting the data or objects into the format suitable for transmission and demarshaling is converting back to the original format. It is similar to serialization, which simplifies the complex communication.

Client/Server communication model

Client/Server communication model is a tightly coupled communication model which is used for one-to-one communication. It supports mainly two application level operations. They are

- Execute on client side, client port objects
- Perform on all Network visible, server side objects.

As mentioned, client objects `Execute()` or invoke operation on the network, whereas the server objects performs the operations based on the ID and the results to client using the function `Perform()` as shown in below Figure 5.5.

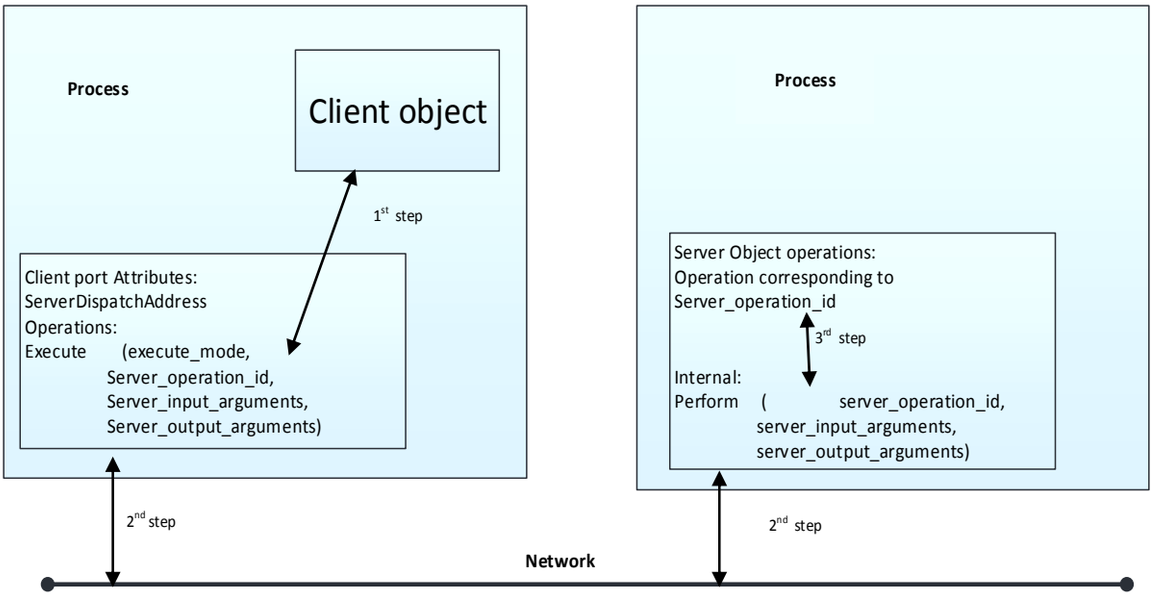

Figure 5.5 Client- Server communication model components and the process of communication in three steps [11]

As shown in the figure both `Execute()` and the `Perform`() operations work together to provide a remote object operation invocation. Invoking is basically done in a series of steps:

Step 1: The *serverDispatchAddress* attribute of a client port object in client objects process is attached to the server object dispatch address value during the system process.

The client object is provided with a client port object as reference for the initialization of the system.

As shown in the Figure 5.5, the client object invokes the *execute* operation on the client port attributes. This allows the remote server object to choose the operation that has to be executed. It also provides its operation's input arguments and references to its output arguments.

Step 2: Through the network infrastructure shown in the Figure 5.5, the invocation of the `Perform()` operation on the server side by the invocation of `Execute()` operation will be done.

Step 3: The `Perform()` operation invokes the necessary operations on the server object. After the completion of server objects operation again through the network infrastructure the clients object operations will be invoked.

Finally the *Execute()* operation sends an invocation to client object with the output argument array's argument bounds [11].

Publish/Subscribe communication model

The *Publish/Subscribe* communication model is loosely coupled for many-to-many and one-to-many communication. It is mainly used for broadcasting or multicasting measurement data and configuration management information. It contains two objects called publisher and subscriber, where publisher acts as a sender and subscriber acts as a receiver. Publishing and subscription are done using the operations `Publish()` and `AddSubscribe()`. The publisher does not need to be aware of any receiving object whereas the subscriber sends a response when something sends a request of subscription.

The IEEE 1451.1 publication has following principles with it [11]:

- A publication domain defines a distribution scope for publication.
- A publication key identifies the application-independent syntax and semantics of the publication.
- A publication topic can be used to identify the application specific syntax and semantics for the publication contents.

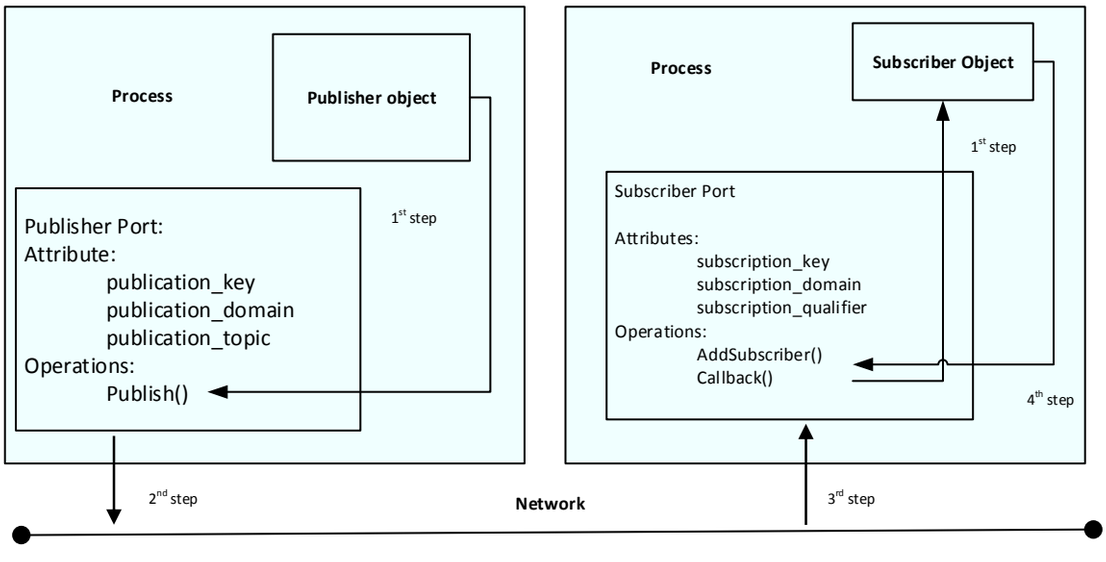

Figure 5.6 Publish/Subscribe Communication model components and the process of communication described in four steps [11]

The process of Publish/Subscribe communication model is shown below [11]:

Step 1: In the first step system initialization will be done near the publisher port and as well as at subscriber port side. The publication key attribute of a publisher port object in the publisher object's process will be bound. Similarly, the subscription key attribute of a subscriber port object in the subscriber object's process will be bound to value matching the publication key of publications of interest.

Then the publisher and subscriber objects will be provided with a local reference to their respective port objects.

Step 2: The publisher object invokes the `Publish()` operation on the publisher objects port and an input argument publication contents will be released.

Using the network infrastructure, the invocated publisher port delivers the publication to subscriber ports in the publication domain.

Step 3: The subscription qualifier is used to determine which publications will be accepted by the port. The *GetSubscriptionKey* operation will be used to obtain the current value of the port's subscription key and the *GetSusbscriptionDomain* will be used to obtain the current value of the subscription domain defining the set of candidate publications to be accepted by the port.

The subscriber object uses the subscriber port objects *AddSubscriber()* operation to register with the port to receive the publications from the selected port.

The receiving subscriber port uses the values from subscription key, subscription domain and the subscription qualifier to filter the incoming messages.

Step 4: Once the publication passes the subscription filter, then the port invokes all the registered `callback()` operations. Then the port provides an input argument the publication contents.

The Publish/Subscribe model communicates the NCAPs by following the above steps stated. The publish/Subscribe model is used in JNCAP and NCAP communication (see section 7.5).

6 Analog input and analog output using Raspberry Pi

This chapter focuses on the analog signal generation using ALSA programming to steer the actuator. It includes the working and program flow of sine wave generation and in details the capturing program to record the sensor input signals.

6.1 Sine wave Generation

As described in section 4.3, the strategy is to drive the actuators by an analog sine wave signal. For this purpose Raspberry Pi is using a sound card as ADC and DAC (see section 3.2.1). The sine wave will be generated using the ALSA drivers.

The implementation of an analog sine wave generation module for later use is described in the following section.

6.1.1 Analog Output from Raspberry Pi

A C module for sine wave generation with a specific frequency has been created in Geany.

- The access to the drivers is obtained by including the header files `#include <alsa/asoundlib.h>` and `#include <alsa/pcm.h>`. These are very essential for opening the ALSA sound drivers. They include all the definitions of ALSA functions (see section 3.3.1).
- To open a PCM device, certain functions have to be included. The function call `snd_pcm_open` opens the default PCM device and sets the access mode to *PLAYBACK*. It also sets some parameters and then displays the values of the hardware parameters as shown in Figure 6.1.

```
{
   int err;
    char *device = "default";
    snd_pcm_t *handle;

         if ((err = snd_pcm_open(&handle, device, SND_PCM_STREAM_PLAYBACK, 0)) < 0)
          {
           printf("Playback open error 1: %s\n", snd_strerror(err));
           exit(EXIT_FAILURE);
           }
         if ((err = snd_pcm_set_params(handle,
                                 SND_PCM_FORMAT_S16,
                                 SND_PCM_ACCESS_RW_INTERLEAVED,
                                 1,
                                 FS,
                                 5,
                                 500000)) <0) {   /* 0.5 sec */
           printf("Playback open error 2: %s\n", snd_strerror(err));
           exit(EXIT_FAILURE);
                  }

           return handle;

       }
```

Figure 6.1 Sample code from the sine wave generation code showing PCM device opening

- Buffer size is set to 44,100 using the macro `BUFFER_SIZE 44100`.
- The hardware parameters were activated using the function `snd_pcm_hw_params`. The desired hardware parameters can be set using API call, which includes the PCM stream handle, the hardware parameters structure and the parameter value. Then, the stream is set to interleaved mode, 2 channels and sampling rate to 44,100 bps.

The function to generate and put out the analog sine wave value is depicted in the Figure 6.2. By using the '`sin`' function (from the math library) a buffer can be filled with signal values regarding sampling rate 'FS' and at given frequency '`f`'. A sample code is attached as Figure 6.2for detailing the implementation process.

The function `snd_pcm_writei`, sends the data to PCM device to control the number of output frames is returned.

```
signed short sine(snd_pcm_t *handle, int f)
 {
     int i,j;
     signed short buffer[BUFFER_SIZE];
     snd_pcm_sframes_t frames;

    printf(" A sine wave at %d hz\n",f);

        {
           for (i = 0; i < BUFFER_SIZE; i++)

             {
                buffer[i]= (signed short)(sin(2*M_PI*f/FS*i)*32767);
             }
         printf(" buffer %u \n",buffer[2]);

        }


       for (j = 0; j<5; j++)

         {
            frames= snd_pcm_writei(handle, buffer, BUFFER_SIZE);
            printf(" FRAMES %d \n",frames);
         }
       return 0;
```

Figure 6.2 Sample code showing the sine wave generation function

The opened handle has to be closed using `close_handle` function as shown in the Figure 6.3

```
int close_handle(snd_pcm_t* handle)
{
    snd_pcm_close(handle);

}
```

Figure 6.3 Sample code to show the closing function

The program must be compiled with the linking console –lportaudio and –lasound. Then the module can be used to generate a sine wave at the given frequency which is required to drive analog signal to actuator. The module has been tested with an example program and can be reused for project purpose.

6.2 Capturing sensor values

Raspberry Pi cannot read analog signals directly. Some external hardware is required to read the data. In order to read the sensor data an external sound card is used as a medium. The signals generated from a sensor are converted to sound samples using ALSA drivers with the

help of this sound card. In order to observe the values there is a need of a software module through which the signals can be recorded.

6.2.1 Analog input to Raspberry Pi

The module has been generated using Geany. To capture analog signals.

The header files `#include <stdlib.h>` and `#include <alsa/asoundlib.h>`, which allows the program to open the important audio drives are included.

In order to open a PCM device several functions have to be executed in the function routine. The function call `snd_pcm_open` opens the PCM stream and sets the access mode to `CAPTURE`. A sample code showing the open handle function is depicted in the Figure 6.4.

```
if ((err = snd_pcm_open (&capture_handle,pcm_name, SND_PCM_STREAM_CAPTURE, 0)) < 0) {
  fprintf (stderr, "cannot open audio device %s (%s)\n",
           pcm_name,
           snd_strerror (err));
  exit (1);
}
```

Figure 6.4 Sample code to open the handle function

After initialising the access mode, the other required parameters for the capture module have to be initialised. The hardware parameters, channels, rate, format, access etc. needed for capturing are set. The Figure 6.5 shows the function sequence in detail

```
if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
  fprintf (stderr, "cannot allocate hardware parameter structure (%s)\n",
           snd_strerror (err));
  exit (1);
}

//fprintf(stdout, "hw_params allocated\n");

if ((err = snd_pcm_hw_params_any (capture_handle, hw_params)) < 0) {
  fprintf (stderr, "cannot initialize hardware parameter structure (%s)\n",
           snd_strerror (err));
  exit (1);
}

//fprintf(stdout, "hw_params initialized\n");

if ((err = snd_pcm_hw_params_set_access (capture_handle, hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
  fprintf (stderr, "cannot set access type (%s)\n",
           snd_strerror (err));
  exit (1);
}

//fprintf(stdout, "hw_params access setted\n");

if ((err = snd_pcm_hw_params_set_format (capture_handle, hw_params, format)) < 0) {
  fprintf (stderr, "cannot set sample format (%s)\n",
           snd_strerror (err));
  exit (1);
}

//fprintf(stdout, "hw_params format setted\n");

if ((err = snd_pcm_hw_params_set_rate_near (capture_handle, hw_params, &rate, 0)) < 0) {
  fprintf (stderr, "cannot set sample rate (%s)\n",
           snd_strerror (err));
  exit (1);
}
```

Figure 6.5 Setting parameters required to capture the sensor readings

A function called `Capture()` is implemented to capture the analog signal as samples. All the values will be stored in a buffer as shown in the Figure 6.6.

```
int capture(snd_pcm_t *capture_handle, char **buffer)
{
  int i,k,err;
  //char* buffer;
  int buffer_frames = 44100;

snd_pcm_format_t format = SND_PCM_FORMAT_S16_LE;

  *buffer = (char*) malloc(buffer_frames * snd_pcm_format_width(format) / 8 * 2);

  //fprintf(stdout, "buffer allocated\n");

    if ((err = snd_pcm_readi (capture_handle, *buffer, buffer_frames)) != buffer_frames) {
      fprintf (stderr, "read from audio interface failed (%s)\n",snd_strerror (err));
      exit (1);
    }
```

Figure 6.6 Function showing the buffer capturing the values

Similar to sine generation, a `close_capture` function is used to close the handle.The module must be compiled with the linking console with either –lportaudio or –lasound.

This interfacing can be included in an arbitrary program. The returned buffer contains the set of analog input samples over approximately one second.

7 IEEE 1451.1 Implementation

This chapter provides a detailed description about the IEEE 1451.1 NIST open Gaithersburg implementation in this project. A brief description about the example implementation of temperature program is also given in this chapter which is downloaded from the source forge [20]. In the later sections, the exact implementation of aoNCAP and aiNCAP is discussed. This chapter also covers the detail description of program flow of IEEE 1451.1 and control mechanism of aiNCAP and aoNCAP by JNCAP.

7.1 Example implementation of a temperature measurement using IEEE 1451.1

An example implementation from IEEE 1451.1 is used as mentioned in the chapter 3. Temperature NCAP (tempNCAP) and Java NCAP (JNCAP) are two different programs through which communication takes place in the reference model. The tempNCAP uses IEEE 1451 library to communicate with JNCAP. The IEEE 1451 library consists of several files which acts like a base for all functions in the tempNCAP. It provides all the functionalities required for the tempNCAP, which inherits the properties from the IEEE 1451 library as mentioned in the IEEE 1451.1 (see chapter 5) standard [11].

NCAP consists of transducer, functional and NCAP block as mentioned in the chapter 5, Figure 5.4. It uses the data model for the data formats like *Float32* which is required for the communication paradigm and an object model for discovering, invoking synchronizing and many such. It uses the communication models like publish/subscribe and client/server models for the communication between JNCAP and tempNCAP. All these blocks deduce their functionality from the IEEE 1451 library while communicating with JNCAP.

As mentioned in the section IEEE 1451 example implementation from source forge, this temperature program consists of C++ and JAVA files, in which the C++ program uses the ACE wrappers. The tempNCAP is controlled by the JNCAP which is a Java application. Initially the reference program sends some temperature values of a sensor through the network. The simulated temp TIM get values from the transducer block. The *Float32* data array is used in the temperature program and the argument array consists of all the temperature data. The used data architecture was described in the section 5.3.2.

The output from the temperature program consists of the temperature values. When JNCAP starts, various options are displayed as shown in the Figure 7.1. In order to discover the

tempNCAP, user have to push discover button and the IP address of the remote tempNCAP is displayed on the screen.

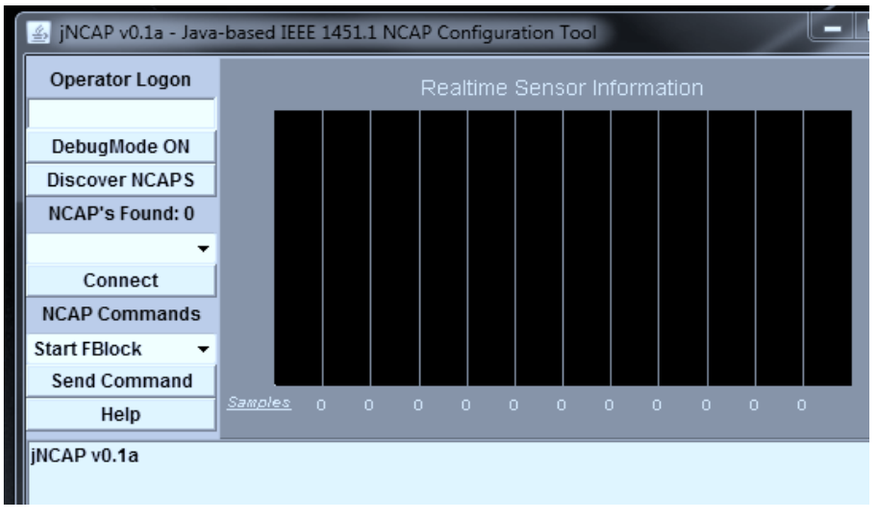

Figure 7.1 JAVA JNCAP screen when it starts initially showing the options like *discover, connect* and *send command* on the window

In the next step, in order to connect JNCAP and tempNCAP, "connect" option has to be selected as shown in the Figure 7.1. If the tempNCAP receives the command it triggers all the blocks in tempNCAP. Then tempNCAP sends an acknowledgment to JNCAP and connects with JNCAP. When the user wants to receive some temperature values from tempNCAP, "Send" command has to be selected.

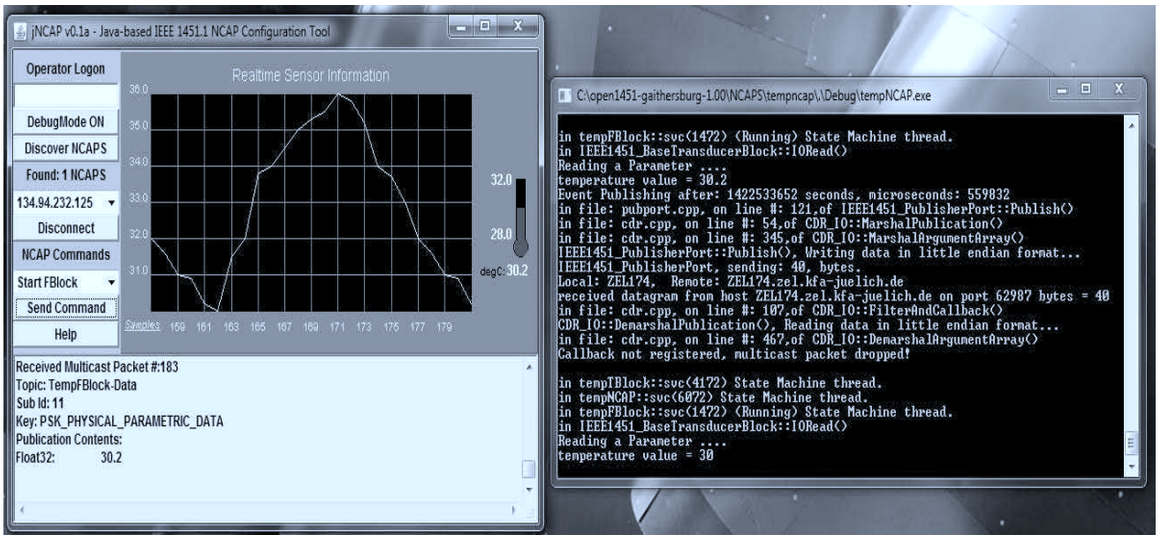

Figure 7.2 The output windows of JAVA NCAP (JNCAP) and tempNCAP showing the fictional temperature values when JNCAP sends the *send command.*

Once the command is received by the tempNCAP, it sends a request to the respective blocks inside the tempNCAP. The blocks inside the tempNCAP i.e. transducer and function blocks

which are responsible for the data processing are pushed into active state. As mentioned in the section 5.3.1, the *BL_ACTIVE* goes to active state. When *BL_ACTIVE* state goes to active state the transducer block is activated. This changes the state of the function block from *FB_IDLE* to *FB_RUNNING*. When the function block comes to running mode, it starts sending the temperature data to JNCAP as shown in the Figure 7.2. The temperature values are stored in *Float32* array and are extracted from the transducer block using the *IORead* function in the functional block. The function block then publishes the data using the publish/subscribe mode of communication and JNCAP subscribes the data. And apparently, the temperature data can be visualized on the JNCAP screen.

7.2 The aoNCAP Implementation

As the objective of this project work is to send some amplitude values via analog output to steer the actuator and to receive analog signal values from the sensor in a distributed actuation control, interfacing has been done in such a way that it receives and sends data from raspberry pi from and to actuator and sensor. One NCAP is used for sending the analog signal to the actuator and the other NCAP receives the sensor reading from sensor. One JNCAP controls both the NCAPs using the NCAP to NCAP communication path. Therefore they are named as analog output NCAP (aoNCAP) and analog input NCAP (aiNCAP).

7.2.1 Purpose of aoNCAP

The aoNCAP is responsible for sending analog sine wave to actuator. It uses sound card as a DAC (digital to analog converter). This can be observed from the Figure 7.3. The aoNCAP is controlled by JNCAP which is Java NCAP, receiving commands from JNCAP, what it has to be performing.

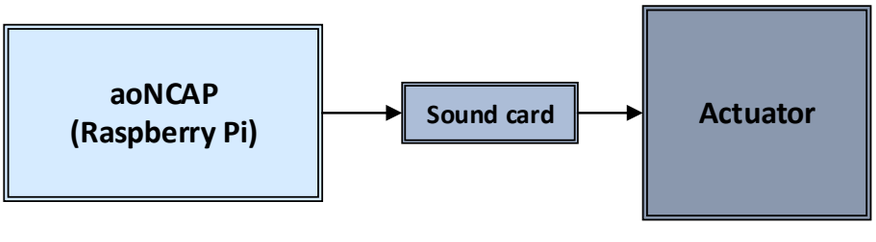

Figure 7.3 Diagram showing connection between AONCAP and actuator using sound card as a transducer interface module (TIM)

The aoNCAP communicates with JNCAP with the help of IEEE 1451 library.

7.2.2 AONCAP description

The aoNCAP communicates with JNCAP through network interface. The major blocks in aoNCAP are transducer block, function block, aoNCAP block and aoMain block as shown in the Figure 7.4. Every block has an exclusive functionality in the NCAP communication. A brief description of each block can be seen in the following sections.

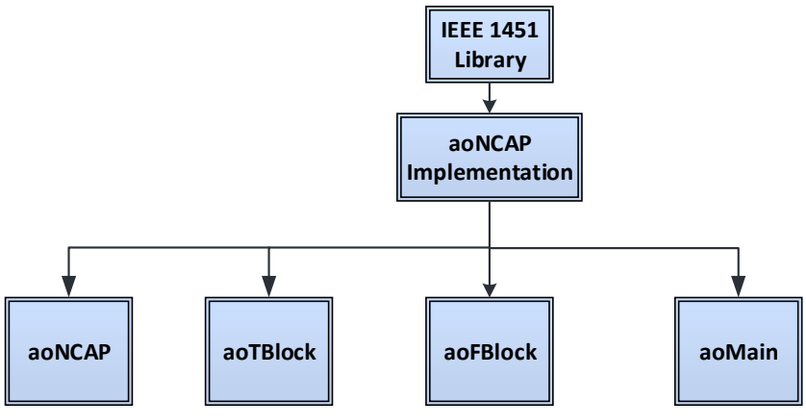

Figure 7.4 Diagram showing the blocks inside AONCAP module through which communication occurs

The aoNCAP block

The aoNCAP block brings all the blocks together and is responsible for communication housekeeping. The discover and publish operations are done by this aoNCAP block. The functionality of the NCAP block has been explained clearly in the section 5.3.1. In this section the main functionality of the NCAP in aoNCAP implementation has been depicted.

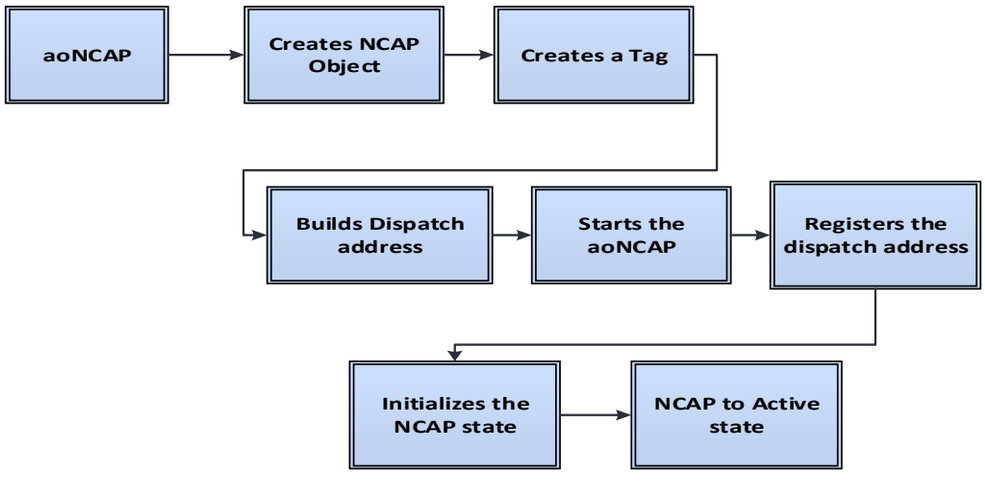

Figure 7.5 Functions performed by the aoNCAP block in AONCAP implementation in step wise

The NCAP block creates an NCAP object which defines the TCP server port and its assignment. Then it creates a tag, which can be used to identify the NCAP for others on

different networks. It builds a dispatch address for clients to connect to the JNCAP. There after it registers the dispatch address with the NCAP and tells NCAP to go active or to start running. The communication flow can be observed from the Figure 7.5 above.

The aoTBlock

This is responsible for interfacing the transducers and the application functions using software programming. The operations and their functionalities such as correction mode, sampling mode and others are discussed in the section 5.3.1in detail.

The functionality of the TBlock is clearly depicted in the Figure 7.6. In the first step the transducer block starts defining a tag for the object, named as "aoNCAP-TBlock" in order to be recognised by the clients. The object tag defines a logical endpoint for the server side of client-server communication with a datatype '*Object Tag'*. It is usually assigned by the end user and is unique within a given system [11]. Then it creates a dispatch address additional to the one created in the NCAP block.

As illustrated in the Figure 7.6, it defines a set of object properties which gives a special identity to the block. Then it registers the transducer block with the NCAP, such that it makes itself visible for the operations. In the next step it initiates the transducer block, followed by the registration of the transducer block with the NCAP. Then on receiving the appropriate trigger form the JNCAP it goes to active state. Then the state machine of the transducer block goes to *TB_CORRECTED* i.e. *BL_ACTIVE* as mentioned in section 5.3.1.

The transducer block is also responsible for sending data to the function block on receiving the command from the JNCAP. As seen in the example implementation, the transducer block is responsible for holding the temperature values, and in the similar way, aoTBlock is responsible for sending analog signal to actuator. This is done by implementing the *IOWrite* function as shown in the Figure 7.7.

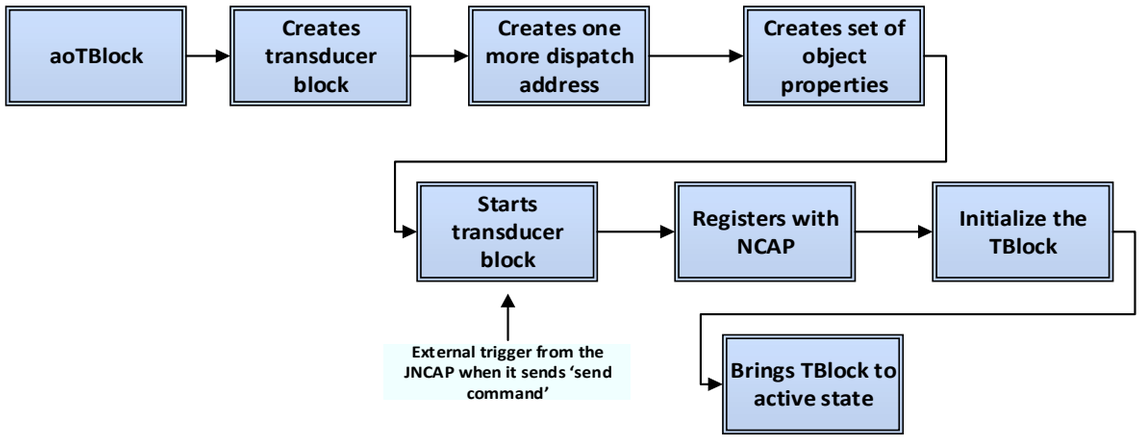

Figure 7.6 Functions performed by TBlock in aoNCAP Block in pictorial representation

The *IOWrite* function extracts the data from the sinewave generation interface described in the section 6.1.1. The functionality of the sine wave generation has been implemented in the transducer block of aoNCAP, so that it generates a sine wave at particular frequency as given in the program.

```
OpReturnCode aoTBlock::IOWrite(const ArgumentArray& io_input_arguments,
                               ArgumentArray& io_output_arguments)

{
    ArgumentArray aa(io_input_arguments);

    Float32 frequency;
    aa >> frequency;

    snd_pcm_t *handle;
    handle=create_handle();
    sine(handle,(int)frequency);
    close_handle(handle);
    std::cout<< "write data \n" <<frequency<< std::endl;
    io_output_arguments << frequency;
    return MJ_COMPLETED;
}
```

Figure 7.7 The sample code of the transducer block showing sine wave generating function

As can be observed from the Figure 7.7 transducer block uses the argument array to store the frequency value through the `io_input_arguments`. It gets the functionality of the sine wave from sine wave generation interfacing i.e. `sine(handle,(int) frequency)` as shown in Figure 7.7, which is explained in chapter 6. The *`IOWrite`* operation sends the data to the functional block through which actuator gets the frequency upon JNCAP sending the *'send command'*.

The aoFBlock

This block is responsible for the abstraction and packaging of application functionality in the IEEE 1451.1 based program. All the application specific functionalities are owned and controlled by the functional block. The theoretical functionality of this block is described in section 5.3.1

The program flow of the functional block in aoNCAP is clearly mentioned in the Figure 7.8below. Firstly, a function block defines tag for the object as "*aoNCAP-FBlock*". Then optional dispatch address will be created which contains the local host address of the function block. It creates a set of object properties including the ncap multicast address which gives a special identity to the block. It instantiates the function block with a new parameter to represent by an instance. After that, it registers the functional block with NCAP to make it visible to the network operations. Once the registration is done, it initializes the functional block and makes the transducer block active. Then finally it starts the application or invocation with the help of *start ()* function.

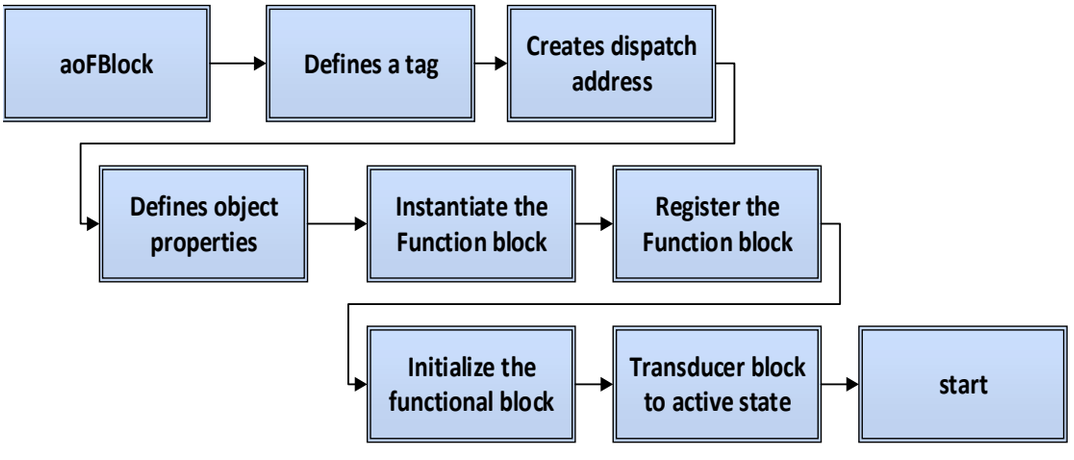

Figure 7.8 Functions performed by FBlock in aoNCAP block in pictorial representation

As explained in the example implementation, functional block communicates with the JNCAP in order to send and receive data. In aoNCAP, it publishes the data to JNCAP upon receiving the *send command* from the JNCAP. This block is responsible to process the data which it obtains from the transducer block using the *IOWrite* function. The *IOWrite* function has been explained in the above section in aoTblock.

A sample code from the FBlock where the sine wave function has been implemented is shown in the Figure 7.9. As observed from the Figure 7.9, it is clear that function block is calling the IOWrite function on tblock using input and output arguments called `ao_arg_p_read` and

`ao_arg_p_write`. These input and output arguments have been implemented in the IEEE 1451 library, and whenever needed the functionality of these arguments will be called to aoNCAP block.

```
mx->Unlock( x_ );
// send it
*ao_arg_p_write << aofloat;


/* call IOWrite on TBlock */
ret = t_block_p_->IOWrite(*ao_arg_p_write, *ao_arg_p_read);

 // reset >>
*ao_arg_p_write >> aofloat;
// read frequency
*ao_arg_p_read >> aofloat;

 std::cout << " by tempTblock #2 " << aofloat << " ***" <<endl;
```

Figure 7.9 Sample code of the Fblock showing the use of the analog output interface IOWrite

When the functional block receives message it changes its state from *FB_IDLE* to *FB_ACTIVE* and it starts the functional block then it activates the transducer block as shown in the Figure 7.8.

The aoMain block

The aoMain block holds all the data functionality of the transducer and functional block. The functions implemented in the functional and transducer block are accessed from the main block.

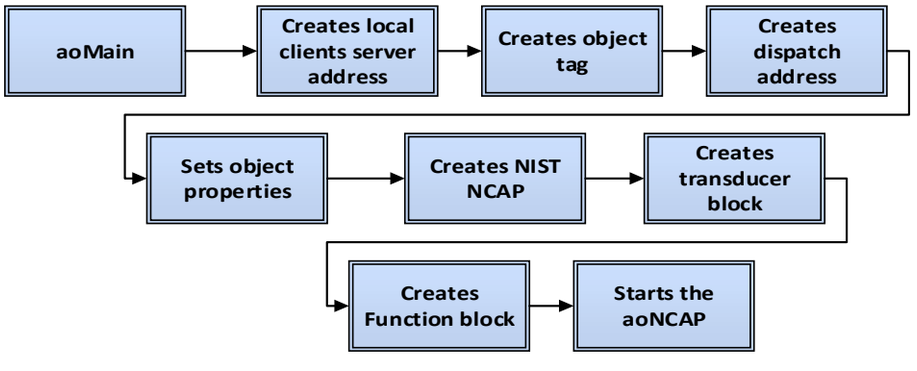

Figure 7.10 Functions performed by the main block in aoNCAP in pictorial representation

The program flow of the main block has been depicted in the Figure 7.10. It creates the local client server address and port in order to be recognised by the JNCAP. Then it creates an object tag as mentioned in the TBlock and FBlock in previous sections, it also creates the

dispatch address for the NCAP. Then it sets the object properties as similar to TBlock and FBlock and then it creates the NIST NCAP, through which the JNCAP can discover the aoNCAP. Similarly it creates the transducer and functional block, all the properties of transducer, functional and NCAP blocks are available by the main block. It publishes the data using the functional and transducer block functions.

7.3 The aiNCAP Implementation

The aiNCAP is responsible for receiving the sensor readings from the sensor in distributed actuation control. The operation of aiNCAP is similar to aoNCAP, except the configuration in FBlock. It is connected to the JNCAP using the IP address and it will be controlled by JNCAP using the send command. When JNCAP sends command the output from the sensor will be displayed on the JNCAP screen. A clear description of the aiNCAP functionality will be explained in below sections.

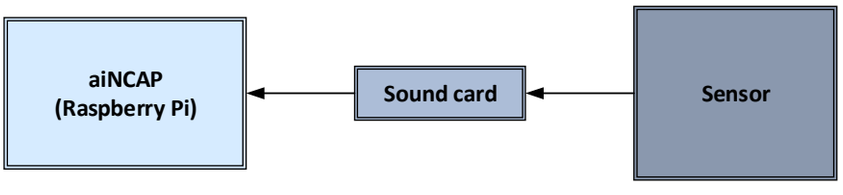

Figure 7.11 Diagram showing the connection between the aiNCAP and the Sensor with a sound card as a transducer interface module (TIM)

Figure 7.11 shows the connection between the aiNCAP and sensor and how the aiNCAP communicates the sensor using sound card.

7.3.1 The aiNCAP description

The aiNCAP communicates JNCAP through network interface. The blocks inside aiNCAP are transducer block, functional block, aiNCAP block and aiMain block as shown in the Figure 7.12. The functionality of the NCAP block and Main block in aiNCAP is similar to aoNCAP and aoMain block of aoNCAP implementation. For this reason, there will be clear description regarding the transducer and functional block in detail instead of concentrating on NCAP and main blocks in this section.

The NCAP block is responsible to create the TCP server port which supports network communication. It helps for registration, deregistration, initialization, startup and many functions like this as mentioned in the aoNCAP block in 7.2.2.

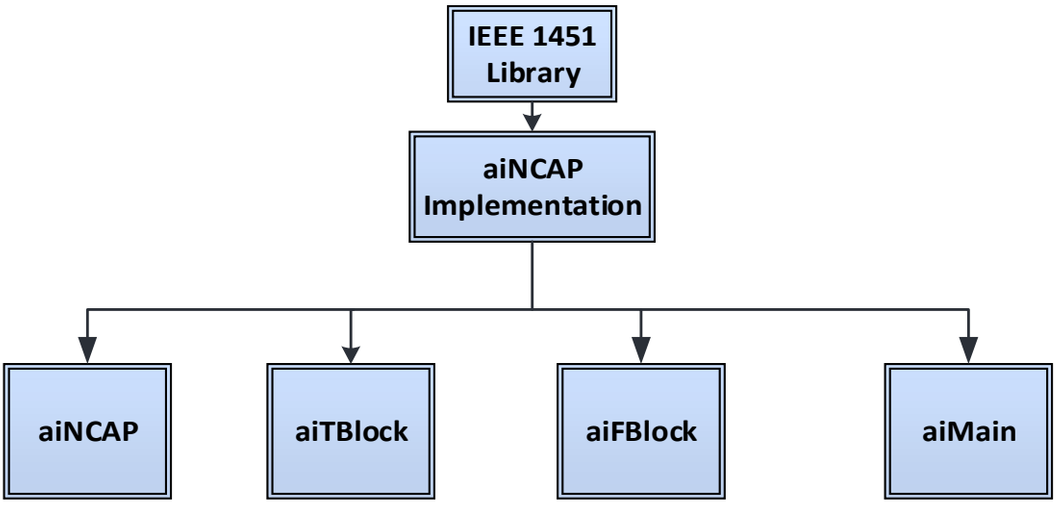

Figure 7.12 Figure showing the blocks inside aiNCAP through which communication occurs

The aiMain block is responsible to hold the data functionality of the transducer block and functional block. There is a clear description about this block in aoMain block in 7.3.1.

The aiTBlock

The transducer block is responsible for interfacing the transducer and the application functions using software programming as mentioned in the aoTBlock section 7.2.2 above. The program flow is same as explained in the aoTBlock section except the functionality. The aiTBlock is designed to receive the sensor values using the *IORead* function. In aoTBlock *IOWrite* function has been used to generate analog signal.

As shown in the Figure 7.13, the transducer block creates transducer block tag and then creates the set of object properties which gives special identity to the aiTBlock. In the next step it registers the transducer block with the NCAP as explained in the aoTBlock above.

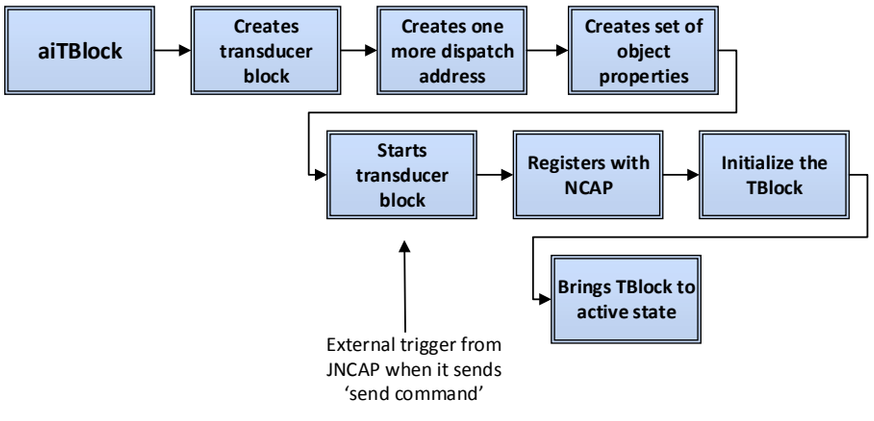

Figure 7.13 The figure showing the program flow of the transducer block in aiNCAP implementation

The aiTBlock is initialized and brought to active state upon receiving the external command from the JNCAP.

For data receiving it uses the capture interfacing program which is explained in the section 6.2. The *IORead* function as shown in the Figure 7.14 consists of input and output argument arrays where the data will be stored. When JNCAP sends a command, it starts the TBlock and receives the sensor values using the sound card and ALSA programming interface. The values obtained from the sound card will be extracted by the *IORead* function. From the Figure 7.14 it can be observed that a capture function has been initialized to receive the sensor readings using `capture(capture_handle, &buffer).`

```
/*
 Analog input function through which NCAP reads the sensor values
 */
OpReturnCode aiTBlock::IORead(const ArgumentArray& io_input_arguments,
                              ArgumentArray& io_output_arguments)

{
    ACE_DEBUG((LM_INFO, "BLA::IORead()\n"));

    char *buffer;
    snd_pcm_t* capture_handle;
    capture_handle=open_handle();
    int a=capture(capture_handle,&buffer);
    close_handle(capture_handle);

    sensorarray = Float32Array(a,buffer);

    io_output_arguments << sensorarray;

    return MJ_COMPLETED;
}
```

Figure 7.14 A sample code showing the capture function which reads the sensor readings from the sensor

As shown in the Figure 7.14, the `sensorarray` extracts the data from the `buffer` and then it hands it over to `io_output_arguments.` After receiving the sensor values, the io_output_arguments sends the data to functional block for data processing, which will be explained in the aiFBlock.

The aiFBlock

This block is responsible for data abstraction and controlling the application specific functionalities. The program flow of the analog input functional block is much similar to aoFBlock mentioned in the section 7.2.2. A brief description of the function block is depicted in the Figure 7.15. It starts defining a tag and then creates the dispatch address which contains the local host address of the function block.

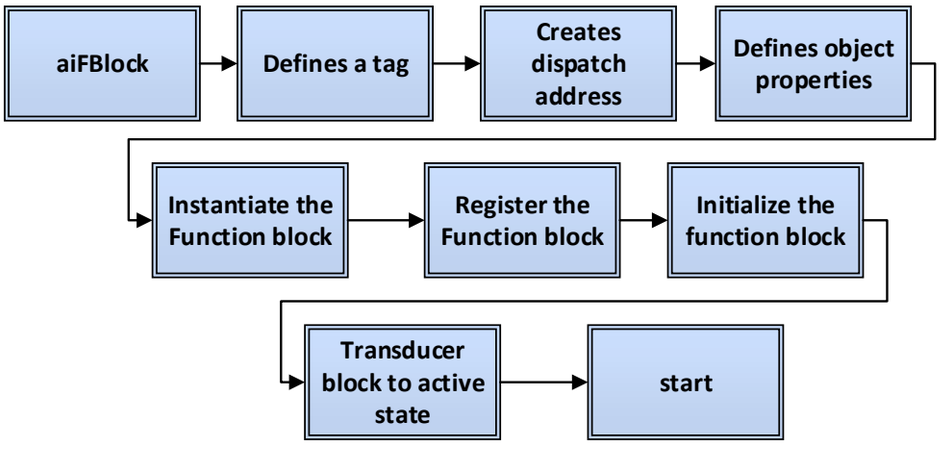

Figure 7.15 The figure showing the program flow of the function block in aiNCAP implementation

In the later stage it defines the object properties which give a special identity to the block and then it registers the function block and initializes the function block. When the function block receives a command from the JNCAP, it changes its state from *FB_IDLE* to *FB_RUNNING*. It activates the transducer block once it goes to running state and will be able to receive the data from the transducer block.

```
// convert the argarray from read to the analog input
*ai_arg_array_p >> ai;

dataArray = ai.Get();

size=ai.length();
// extract amplitude value
maximumval= 0;
for(int i=1; i<size; ++i)
{
    if(dataArray[i]>maximumval)
    maximumval= dataArray[i];

    }
std::cout<<"maximum value\n"<<maximumval<<endl;

aifloat = maximumval/(25860.0);
std::cout << " by aiTblock #1 " << aifloat << " ***" <<endl;


mx->Unlock( x_ );


// send it
*ai_arg_p_write << aifloat;
```

Figure 7.16 A sample code showing the data processing functions in functional block of aiNCAP implementation

As the functionality of the function block is to control the application functionalities, it is used for data processing. It sends the data processing to the data abstracted from the transducer

block. After it performs the data processing it sends the data to JNCAP and data can be visualized on the JNCAP screen. The data processing mechanism can be observed from the sample code given in the Figure 7.16. It extracts the sensor values from the `'ai_arg_array_p'` to analog input `'ai'`. Then the `data array` takes all the values from the analog input `ai` using `'Get()'` function. Then the processed data after extracting the maximum value is sent to the output function. Once functional block receives *send command* from JNCAP it delivers the data it extracted from the transducer block.

7.4 Java Network Capable Application Processor (JNCAP)

Java NCAP implemented in the example implementation is responsible for controlling the tempNCAP as mentioned in section 7.1. To control the aoNCAP and aiNCAP JNCAP has been extended. It consists of different source files which are responsible for discovering, publishing, subscribing, disconnecting and many functions like this. The initial example implementation has been extended by a *Disconnect command* and a *SendCommandsToAll* functionality. These commands allow the user to perform a disconnect operation whenever it is needed, and allows the user to send commands to all NCAPs connected to the JNCAP in parallel. Once the send command is pushed on JNCAP, it starts sending the commands until we disconnect the aiNCAP and aoNCAP.

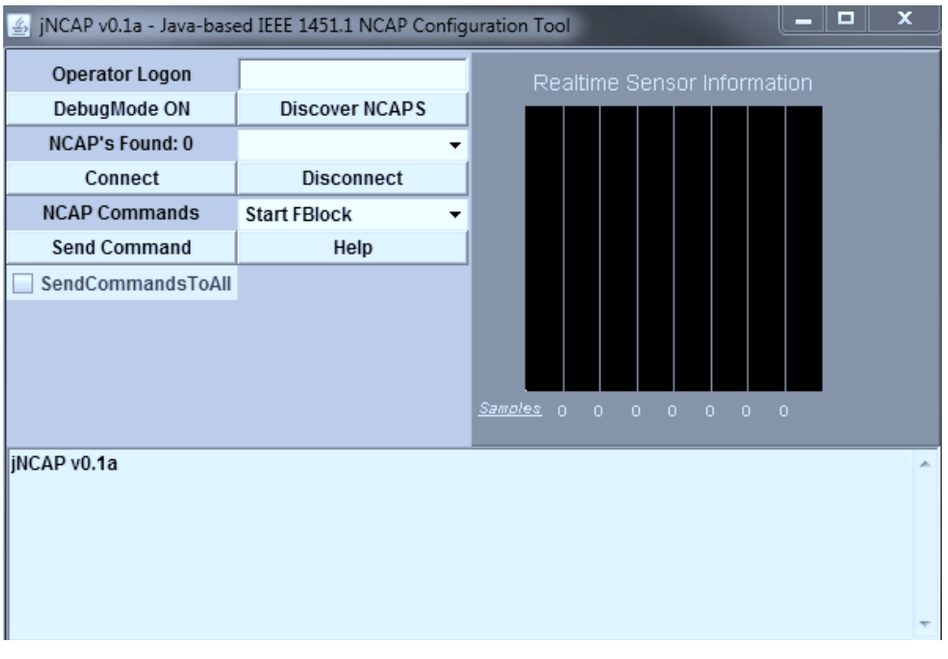

Figure 7.17 JNCAP output window with all the commands like discover, connect, disconnect, Send command and send command to all functions in Real implementation

JNCAP controls the aiNCAP and aoNCAP using the buttons as shown in the Figure 7.17. The functionality has been included along with the buttons in the source files. When user starts JNCAP, a window will be opened as shown in the above Figure 7.17.

Discover

JNCAP searches for NCAPs by a multicast address using the *'object-tag'*, the JNCAP receives the information about the NCAP and establishes connection with it using its IP address. When it receives a *tag name* from the client, the number of discovered NCAPs will be displayed on the window as shown in the Figure 7.17. It displays the number in the box *NCAP's found* and the IP address of the NCAP's in a drop down menu besides.

Connect

Once JNCAP discovers the NCAP's running in the network, then the user will be pushing *"connect"* button on the window. It makes a connection with NCAP using the IP address displayed on the IP address box, just beside the *"NCAP's found"* box. Any number of NCAP's can be connected with the JNCAP using the IP address.

Disconnect

The Disconnect button is used to disconnect the NCAP or terminate the connection. When disconnect button is pushed the NCAP stops running and terminates the connection with JNCAP.

Send command

When *send command* is selected, JNCAP sends a command to start the FBlock there by to receive the data from transducer block using appropriate functions. This includes the command choice and node choice. Command choice consists of commands like *start FBlock, stop FBlock, Resume, Pause, Read FBlock* through which user can select the desired option. The node choice includes the IP addresses of the NCAPs to which these commands should be sent. The *send command* can send command only upon selecting required IP address. It uses the publish/subscribe mode of communication to send and receive data as mentioned in the chapter 5.

SendCommandToAll

The functionality of the *SendCommandToAll* is similar to *Send command*, it is used to send required commands to the connected NCAPs. The additional functionality with *SendCommandToAll* is to send commands to all NCAPS connected with JNCAP just on one click in parallel. So user doesn't have to select each and every IP address of the connected NCAPs to send the commands. It also uses the publish/subscribe mode of communication to receive and send data.

7.5 Controlling AONCAP and AINCAP using JNCAP

As mentioned in the above section, any number of NCAPs can be connected and controlled by the JNCAP. In order to connect the JNCAP with aiNCAP and aoNCAP, they have to be started first. When the user starts both NCAP's, the JNCAP program has to be started. When the JNCAP window will be opened, a series of steps have to be followed in order to send and receive data from aiNCAP and aoNCAP as shown in the Figure 7.18.

When NCAPs and JNCAP are started, in order to make a connection between JNCAP and NCAPs, *Discover* command has to be pushed. The JNCAP then gets the multicast address through the publish mode of communication. The number of NCAPs found will be displayed in the box *NCAPs found.*

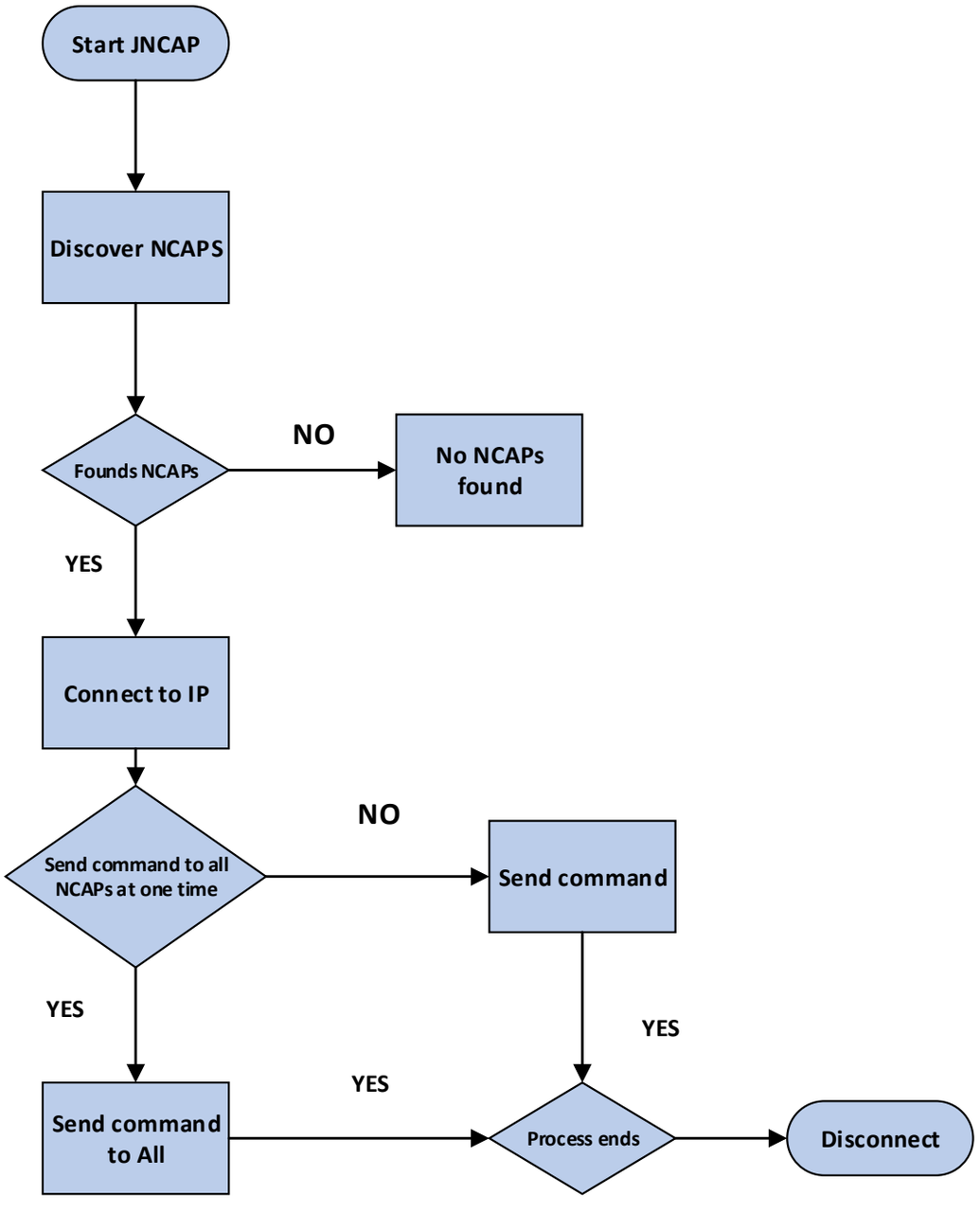

Figure 7.18 Algorithm explaining the control mechanism of JNCAP with aiNCAP and aoNCAP

As shown in the Figure 7.18, once the JNCAP discovers NCAPS, then *Connect* command have to be selected in order to establish a connection. This connection will be done by selecting the desirable IP address of the NCAP from the drop down box beside the *NCAPs found box* as shown in Figure 7.18. Any number of NCAPs can be connected by selecting their IP address.

In order to start the FBlock, send command should be selected. As there are two possibilities to start FBlock, either *SendCommandsToAll* or *Send command* option has to be selected. If *send command* is selected then user have to pick appropriate IP address of the NCAP which is

intended to start. As per the requirement of the project it is sufficient to select *SendCommandsToAll* in order to send commands to aoNCAP and aiNCAP in parallel. Once the JNCAP sends *SendCommandsToAll* command, aoNCAP starts sending analog sine data to actuator and aiNCAP starts receiving sensor values from sensor. As mentioned in the section 4.3, the requirement has been fulfilled.

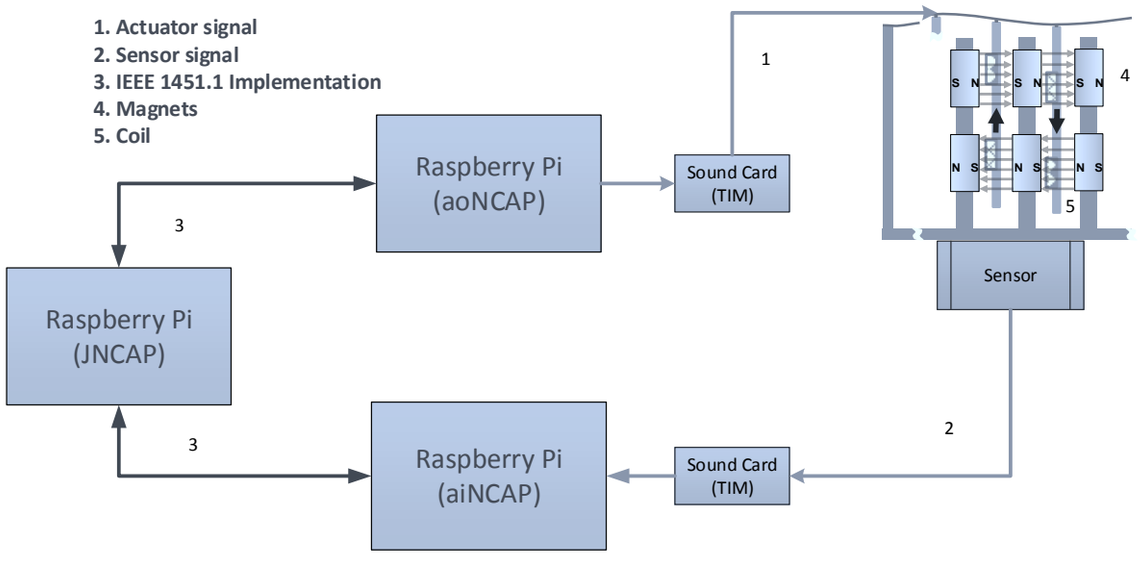

Figure 7.19 diagram showing the connections between the Raspberry Pis , actuator and sensor.

The algorithm shown in Figure 7.18 can be compare to the block diagram of the project (see Figure 7.19). When JNCAP discovers the aiNCAP and the aoNCAP, it tries to connect with two NCAPS using IP address. Once the connection has been established, JNCAP is ready to control the aiNCAP and aoNCAP as shown in Figure 7.19. When JNCAP sends command to aiNCAP and aoNCAP to start the FBlock, aoNCAP starts sending the analog signal to actuator through sound card and aiNCAP starts receiving sensor values and sends them to JNCAP. Then the values can be visualized on the JNCAP application window.

8 Time Measurement

This chapter explains about the propagation time through IEEE 1451 layer. It explains about the time measurement parameters in the earlier sections and about the results in the last section.

8.1 Time measurement in aiNCAP and aoNCAP

The time measurement has been measured in IEEE 1451 layer for aiNCAP and aoNCAP during the process. The time measurement functions has been implemented in IEEE 1451 library, aiNCAP and aoNCAP during the invocation , during dispatching the multicast address and during the publication of data. Each of the implemented functions in the IEEE 1451.1 implementation is explained in the following.

When JNCAP sends *Discover command*, a process is invoked on aiNCAP and aoNCAP. This time is taken as invocation time as shown in Figure 8.1.

```
   case PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT:
      {

#ifdef TIMEMEASUREMENT
       /*Invoke time to file */
       clock_gettime(CLOCK_REALTIME, &time);
       ultime = time.tv_sec*1000000ULL + time.tv_nsec / 1000ULL;
       fprintf(f,"#Invoke time \n %llu \n", ultime);
       printf("Invoke time \n%llu \n", ultime);
       fflush(f);

#endif
    std::cout << "received PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT" << std::endl;

    myNCAP->RespondToRequestNCAPBlockAnnouncement();
      }
```

Figure 8.1 The time measurement function implemented in aiNCAP and aoNCAP during the invocation of NCAPs address discovering

When the invocation is done, NCAPs will send the acknowledgement through IEEE 1451 library. The time measurement has been done during sending the multicast address to JNCAP. The implemented function is shown in the Figure 8.2. A time measurement function has been implemented before and after the send function.

```
#ifdef TIMEMEASUREMENT
     /*Time before send */
    clock_gettime(CLOCK_REALTIME, &time);
    ultime = time.tv_sec*1000000ULL + time.tv_nsec / 1000ULL;
    fprintf(f,"#Time before send \n%llu \n", ultime);
    printf("Time before send \n%llu \n",  ultime);
    fflush(f);


#endif


    // publish the CDR stream to the multicast address
    ssize_t n = publishSocket_->send(out_mb_->rd_ptr(), out_mb_->length(), *multicast_addr_);

 #ifdef TIMEMEASUREMENT
     /*Time after send */
     clock_gettime(CLOCK_REALTIME, &time);
     ultime = time.tv_sec*1000000ULL + time.tv_nsec / 1000ULL;
     fprintf(f,"#Time after send \n %llu \n", ultime);
     printf(" After send \n%llu \n",  ultime);
     fflush(f);
```

Figure 8.2 Time measurement before and after send command i.e. during sending the dispatch address of NCAPs for the request from JNCAP

From the above measurements, the difference between the *invoke time* and *time before send* and the time difference between the invoke time and *time after send* will be considered as a propagation time for IEEE layer and for the whole communication stack.

The time measurement for aiNCAP, when it publishes data upon receiving the *send command* from JNCAP is measured during the publishing of data. The implemented time measurement function for aiNCAP can be observed from the Figure 8.3.

```
                        // send it
                        *ai_arg_p_write << aifloat;

                        // set the publishing contents with the new value
                        if (fblock_data_pub_port != 0){
#ifdef TIMEMEASUREMENT
                            /* Publish time to file */
                            clock_gettime(CLOCK_REALTIME, &time);
                            ultime = time.tv_sec*1000000ULL + time.tv_nsec / 1000ULL;
                            fprintf(f,"#Publish time \n%llu \n", ultime);
                            printf("Publish time \n%llu \n", ultime);
                            fflush(f);

 #endif
                            fblock_data_pub_port->SetPublicationContents(*ai_arg_p_write);
                        }

                        // reset >>
```

Figure 8.3 The time measurement for aiNCAP when it publishes the data upon receiving the send command from JNCAP

For aiNCAP an arbitrary time is also used to publish the data. As shown in the Figure 8.4, aiNCAP publishes the data to JNCAP with a time span of 1 sec. It starts publishing data by

polling the appropriate process with a predefined time period (see time representation in Figure 8.4).

```
    // create a event publisher port for FBlock Data
    cout << "Creating a EventGeneratorPublisherPort ...." << endl;
    fblock_data_pub_port = new IEEE1451_EventGeneratorPublisherPort(mcast_props, *topic);

    // set the PubSub key to PSK_PHYSICAL_PARAMETRIC_DATA, meaning data
    fblock_data_pub_port->SetPublicationKey(PSK_PHYSICAL_PARAMETRIC_DATA);

    // event pub has been retrofitted (non-std) to include time-based event generation

    // create a time period for publishing
    TimeRepresentation timerep;
    timerep.seconds = 1;
    timerep.nanoseconds = 0; //500000; // 1/2 a second

    // set the timer in the eventpublisher
    fblock_data_pub_port->SetTimer(timerep);
}
```

Figure 8.4 Arbitrary time has been set to send data from aiNCAP to JNCAP when aiNCAP receives send command from JNCAP

This time period can be changed as per the time required to publish data. If the time period is reduced to very less value, then aiNCAP starts publishing data frequently. The mean values of several measurements are shown in Table 8.1.

Table 8.1 The mean values of the time through IEEE layer and the time for the whole communication process

	Mean value before send and invoke(Milliseconds)	**Mean value after send and invoke**
aiNCAP	2.8ms	3.5ms
aoNCAP	2.3ms	3.2ms

Considering the resulting values from Table 8.1 a general result of propagation time is less than 3ms through IEEE layer.

The reduction of *time representation* to 1ms also delivers a result less than 3ms propagation time through IEEE communication layer. The minimum time seems to depend on calculation power and network capability as aiNCAP is flooding the network with data in this configuration.

9 Experimental setup and validation

This chapter explains about the experiments and results obtained from the setup. It also gives a description about the validation for the results obtained.

9.1 Experimental setup

The experimental setup has been made as shown in the Figure 1.1. The connections from the controlling node i.e. JNCAP to the aoNCAP and aiNCAP have been made according to the description given in the section 4.1. All the connections were made as per the requirement without any compromises.

9.2 Experiment validation

The validation has been done according to the mentioned theory in section 4.5. The output from the aoNCAP can be visualized using the MacLab by ADI instruments [28]. The analog output from the aoNCAP is given to the MacLab with the help of the sound card as mentioned in the section 4.5. When the aoNCAP receives *send command* from JNCAP, it starts sending the analog signal.

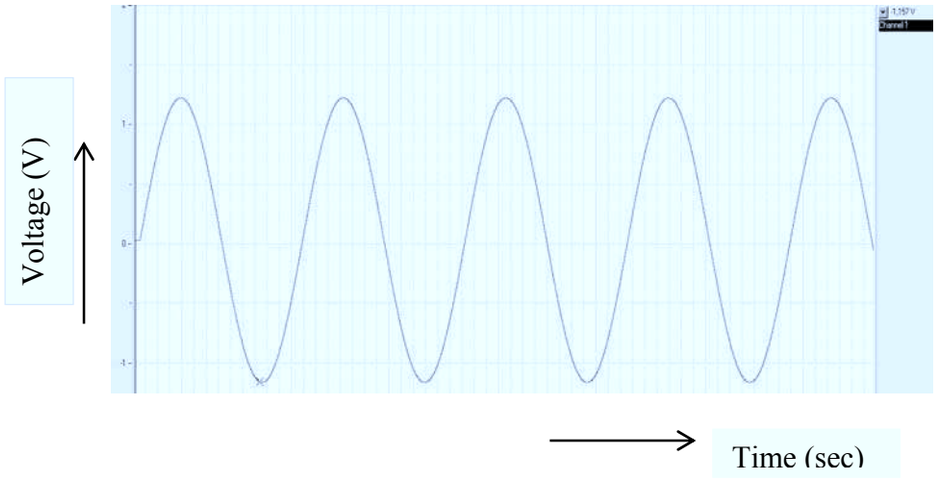

Figure 9.1 An analog signal obtained from the aoNCAP when it receives send command from JNCAP

The result obtained from the aoNCAP is shown in the Figure 9.1. An analog signal of 150 Hz frequency and $1V_{p-p}$ amplitude is sent from aoNCAP to MacLab using sound card.

The result is visualized and validated using the MacLab. The validated result using MacLab is shown in the Figure 9.2. The analog sine wave obtained from the aoNCAP has a frequency of 156.25 Hz instead of 150 Hz and amplitude of -1.15V to 1.18 V peak to peak.

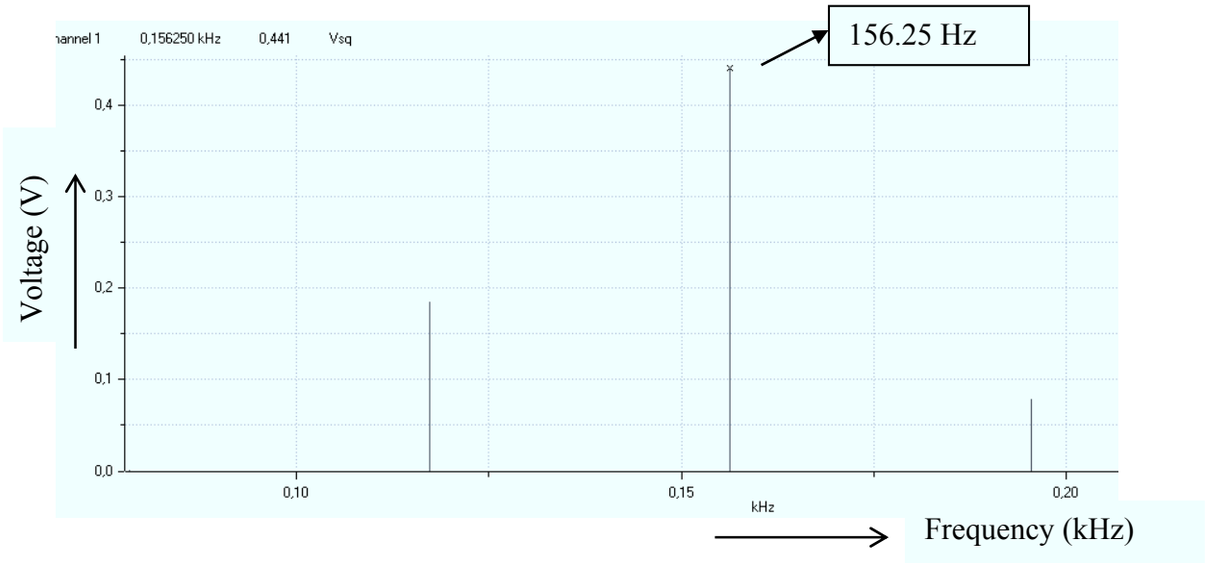

Figure 9.2 The validation report of the analog sine wave obtained from aoNCAP using MacLab at 150 Hz

This result is almost accurate. There is a slight change in the frequency and the difference is due to the capacitance problems obtained from the sound card.

The result from the aiNCAP can be validated using the function generator. An analog sine wave at particular amplitude is sent to aiNCAP using sound card. When JNCAP sends *send command*, aiNCAP starts receiving the data from function generator using the sound card. A sample sine wave of amplitude 1.1 V_{pp} has been sent to aiNCAP through sound card from function generator (see Figure 9.3).

Figure 9.3 A function generator is sued to generate a sinusoidal signal. The signal with amplitude of 1.1 V was sent to the aiNCAP using sound card upon receiving the send command from JNCAP to aiNCAP

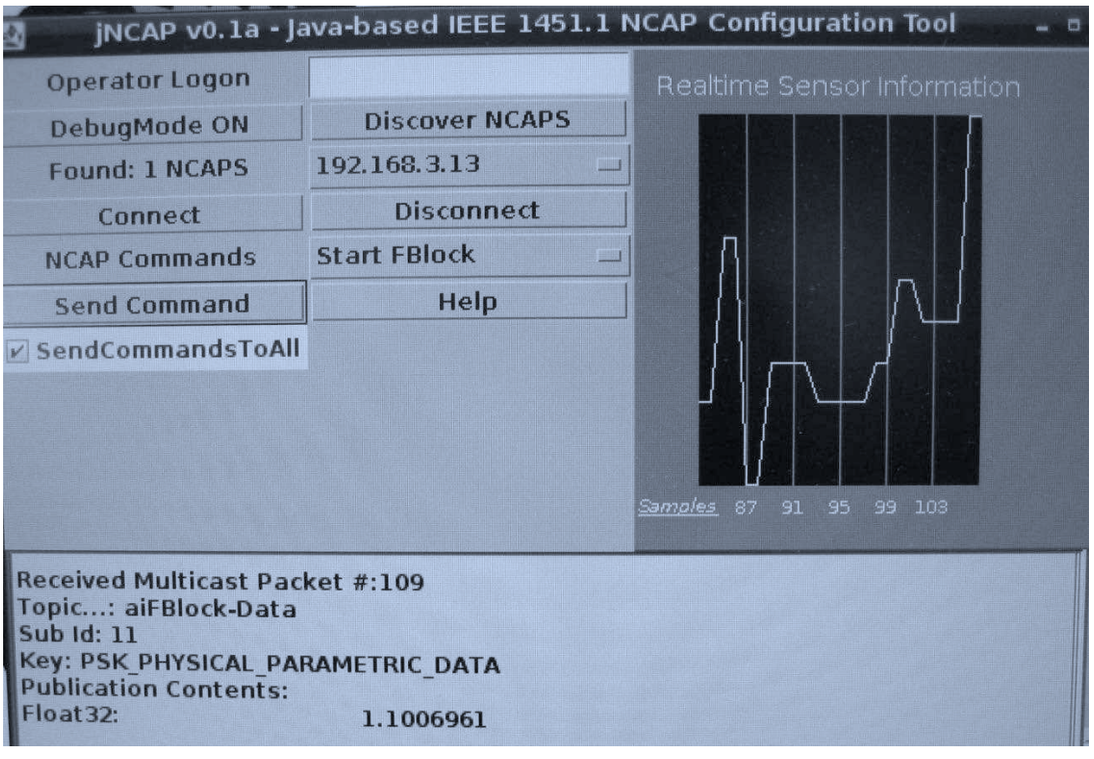

Figure 9.4 The result obtained from function generator can be visualized on JNCAP application window, when aiNCAP publishes the data to JNCAP

The data sent from function generator to aiNCAP can be visualized on JNCAP window. When aiNCAP receives data from function generator with the help of sound card, it starts publishing the data on JNCAP. As observed from the Figure 9.3 and Figure 9.4, the data sent and received are almost equal just with a minor difference. This difference has been occurred due the capacitance problems with the sound card.

The results obtained from the testing environment are satisfactory. It can be assumed that the actuation and measurements will work properly with the actuator and sensors for the flat plate actuation in further experiments.

10 Conclusion and Outlook

The last chapter of the report contains the conclusions of the thesis work, outlines and ideas concerning work.

10.1 Conclusion

The aim of the project is to create a Raspberry Pi Network based on IEEE 1451.1 Smart Transducer Standard Interface Protocol. A network consisting of three Raspberry Pi nodes allows steering actuator and recording sensor values. The example implementation IEEE 1451 from NIST Open Gaithersburg is used as a reference model for IEEE 1451.1 implementation [20].

As described in section 4.1 three Raspberry Pis are connected using a network switch. One of the three Raspberry Pis acts as a Control node (JNCAP) and two other Raspberry Pis are used as analog output (aoNCAP) and analog input (aiNCAP) nodes respectively. These aiNCAP and aoNCAP are connected to actuator and sensor with the help of a sound card. The Raspberry Pi setup works properly and everything is done as per the initial requirement.

The results obtained from the aiNCAP and aoNCAP are validated using the MacLab [28] and a function generator as described in the section 9.2. All the results obtained are as expected and show minimal variations. The mean of the timing values obtained from the validation for propagation time through IEEE 1451.1 layer are less than 3ms which are very promising.

The Raspberry Pi Network which has been established using the IEEE 1451.1 Smart Transducer Interface Standard Protocol is now ready for implementation of a wave control as inner loop of the distributed cascade flow control.

10.2 Future work

The statistical timing results obtained from aoNCAP and aiNCAP using IEEE 1451.1 Smart Transducer Interface Standard Protocol will be imported into actuator and sensor network model to ensure realistic behavior of a large scale actuator and sensor network

There are a few possible options for extending the Raspberry Pi based testbed for actuation control.

The modules can be improved to record a continuous measurement and exiting of the analog output data for aoNCAP. Now the analog output data is processed for every 5 seconds and in the future this may be extended as continuous flow of data.

A closed loop for wave control to realize desired sinusoidal waveform on the surface will be created where the aiNCAP and aoNCAP can be implemented on one node and the combinational node can be controlled using the JNCAP. The combinational node is responsible to receive and use wave parameters. In addition to this, sensor and actuator node can be used to record and publish sensor data for flow control.

The function setup on JNCAP to enter the amplitude and frequency values can be automated instead of hard coded values in the aoNCAP module.

The minor differences occurred during the analog output generation and recording sensor values using sound card could be improved by using external ADC and DAC modules.

11 References

[1] DFG Project description [27.02.2015]

http://gepris.dfg.de/gepris/OCTOPUS/;jsessionid=3E0AABC4DFDF934CD2C55F55E7BAD81C?context=projekt&id=202175528&language=en&module=gepris&task=showDetail

[2] M. Dueck, M. Kaparaki, S. Srivastava, S. van Waasen, and M. Schiek. Development of a real time actuation control in a network-simulation framework for active drag reduction in turbulent flow. In Automatic Control Conference (CACS), 2013 CACS International, pages 256–261, Dec 2013.

[3] Marcel Dück, Mario Schloesser, Stefan van Wassen and Miceal Schiek. Deterministic transport protocol verified by a real-time actuator and sensor network simulation for dustributed active turbulent flow control

[4] Raspberry organization blog [13.01.2015]

http://www.raspberrypi.org/help/what-is-a-raspberry-pi/

[5] Raspberry Pi review by Gareth Halfacree [13.01.2015]

http://www.bit-tech.net/hardware/pcs/2012/04/16/raspberry-pi-review/1

[6] The Edimax dongle specifications from Edimax blog [04.02.2015]

http://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/global/wireless_adapters_n150/ew-7811un

[7] GNU Emacs [16.02.2015]

http://www.gnu.org/software/emacs/

[8] Raspberry Pi resources from IBEX [14.01.2015]

http://www.raspberry-projects.com/pi/category/programming-in-c

[9] Codeblocks organisation [16.02.2015]

http://www.codeblocks.org/

[10] Eclipse blog [16.02.2015]

http://www.eclipse.org/home/index.php

[11] IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Network Capable Application Processor (NCAP) Information Model [6.7.2014]

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=841361

[12] Creative products blog [17.02.2015]

http://en.europe.creative.com/p/archived-products/sound-blaster-play

[13] Creative sound blaster support [14.01.2015]

http://support.creative.com/kb/ShowArticle.aspx?sid=53419

[14] A users guide to ALSA programming [3/09/2014]

http://www.linuxjournal.com/node/8234/print

[15] Jack audio kit organisation [17.02.2015]

http://jackaudio.org/applications/

[16] Simple Directmedia layer organisation [17.02.2015]

https://www.libsdl.org/

[17] OpenAL 1.1 specifications and Reference [17.02.2015]

http://www.openal.org/documentation/openal-1.1-specification.pdf

[18] Advanced Linux Sound Architecture (ALSA) Open Hub [3/09/2014]

https://www.openhub.net/p/alsa

[19] The advanced Linux Sound Architecture for ALSA [3/09/2014]

http://www.alsa-project.org/main/index.php/Introduction

[20] The open Gaithersburg IEEE 1451 implementation from source forge [16.06.2014]

http://sourceforge.net/projects/open1451/files/open1451-gaithersburg/1.00/

[21] NIST IEEE 1451 implementation

http://www.nist.gov/el/isd/ieee/ieee1451.cfm

[22] The ADAPTIVE communication Environment (ACE) [15.10.2014]

http://www.dre.vanderbilt.edu/~schmidt/ACE.html

[23] Implementing IEEE 1451.1 in a wireless Environment by Rick Schneeman, NIST [8.07.2014]

http://ieee1451.nist.gov/Workshop_04June01/Schneeman.pdf

[24] AT-FS708 Unmanaged Fast Ethernet Switches [20.02.2015]

http://www.alliedtelesis.com/media/datasheets/fs708_ds.pdf

[25] How to setup Raspberry Pi to have a static IP address [25.02.2015]

http://www.raspberrypi.org/learning/networking-lessons/rpi-static-ip-address.md

[26] IEEE 1451 A Universal Transducer Protocol Standard by Dr. Darold Wobschall [29.12.2014]

https://eesensors.com/media/wysiwyg/docs-pdfs/ESP16_Atest.pdf

[27] A brief tutorial on IEEE 1451.1 standard by V.Viegas, M.Pereira, and P.Girao [08.01.2015]

http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4483732

[28] ADI Instruments [26.02.2015]

http://www.adinstruments.com/company

Jül-4389
Juli 2015
ISSN 0944-2952