Eindhoven University of Technology

MASTER

Design of a data-acquisition system with a graphical user interface based on VxWorks and network communication

Aarts, A.F.J.

Award date: 1997

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.

• You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Design of a Data-Acquisition System with a Graphical User Interface based on VxWorks and Network Communication

A.F.J. Aarts
WFW-report 97.051

Research carried out at the Cyclotron Laboratory, Department of Applied Physics, Eindhoven University of Technology.
Summary

The group Nuclear Physics Techniques at the Department of Applied Physics of the Eindhoven University of Technology performs ion-beam experiments for material analysis purposes. These experiments are automated by means of a data-acquisition system that is build around two processors, a front-end and a back-end processor. To increase the performance of the data-acquisition system, the front-end processor, the operating system that runs on this processor and the connection between the front-end and back-end processor are changed. The new operating system is VxWorks. The two processors are connected via the local Ethernet network. To increase the user-friendliness of the data-acquisition system it will be equipped with a graphical user interface and the software that is needed to automate the different experiments will be integrated.

To be able to integrate all the software that is needed to automate the different experiments a model experiment is defined. This model experiment incorporates common aspects that are the same for all experiments as well as some aspects that are characteristic for a specific type of experiment. The model experimental set-up and the new data-acquisition hardware and software that are used to perform the model experiment are described.

The communication between application programs that run on the front-end and back-end processor is based on the TCP/IP network communication system. The application programs use the socket interface to communicate with the TCP/IP software inside the operating system. VxWorks applications can use an alternative interface called the zbuf socket interface. This TCP/IP software handles further network communication. TCP/IP provides two types of transport between application programs: reliable connection-oriented transport (TCP) and unreliable connectionless transport (UDP).

Network communication is implemented in front-end and back-end application programs. The front-end applications that run on the VxWorks operating system comprise applications that uses sockets as well as applications that use zbuf sockets to perform network communication. On the back-end processor the network communication is implemented in applications with a graphical user interface. These applications use sockets to perform network communication. The communication between the front-end and back-end applications is based on both TCP and UDP as transport type. Based on the reliability of network communication and network throughput, communication between application programs that use TCP-based sockets gives the best result. No data is lost or corrupted when transmitted from one application to another. The network throughput between these applications is ± 800000 bytes per second.
Contents

Summary iii

Contents v

Introduction 1

1 A Model Experiment 3
  1.1 General Parts of the Model Experimental Set-up 3
  1.2 Data-Acquisition System, the Hardware 5
    1.2.1 General Configuration 6
    1.2.2 Experiment Interfaces 6
  1.3 A Model Experiment 8

2 Data-Acquisition System, the Software 11
  2.1 Operating Systems and Software Development Environments 11
    2.1.1 VxWorks and Tornado 12
    2.1.2 OpenVMS/X Window and X-Designer 13
    2.1.3 Windows95 and LabWindows/CVI 14
  2.2 Application Software 14
    2.2.1 Conceptual Design 15
    2.2.2 The Graphical User Interface 16

3 The Structure of TCP/IP Communication System Software 21
  3.1 The Conceptual Layers 21
  3.2 Application Layer 23
  3.3 Transport Layer 24
  3.4 Internet Layer 25
  3.5 Network Interface Layer 26

4 The Interface between Application and Transport Layer 27
  4.1 The Socket Interface 27
    4.1.1 Create Sockets 29
    4.1.2 Connect Sockets 29
    4.1.3 Send and Receive Data through Sockets 31
## Contents

4.1.4 Close Sockets ............................................ 32
4.2 The Zbuf Socket Interface ........................................ 32

5 Network Communication between Application Layer Programs 33
  5.1 Implementation of Network Communication in Application Programs ........................................ 33
      5.1.1 Introduction ............................................ 33
      5.1.2 Create Sockets ........................................... 37
      5.1.3 Connect Sockets ........................................ 38
      5.1.4 Send and Receive Data through Sockets ........................................ 40
      5.1.5 Close Sockets ............................................ 43
      5.1.6 Concurrency in a Server Application Program ........................................ 43
  5.2 Reliability of Network Communication .................................. 44
  5.3 Network Throughput ............................................ 45
  5.4 Network Communication in Applications with a Graphical User Interface ........................................ 48
      5.4.1 General Concept .......................................... 48
      5.4.2 Implementation ........................................... 50

Conclusions ........................................ 53

References ........................................ 55

A Source Codes ........................................ 57
Introduction

The group Nuclear Physics Techniques at the Department of Applied Physics of the Eindhoven University of Technology develops material analysis techniques and uses these techniques to obtain information about different kinds of materials. The information that can be obtained depends on the specific analysis technique used. The techniques are based on interaction between a beam of protons or He ions with energies in the range from a few to 30 MeV and the material. Analysis techniques that are used are, amongst others, the Particle Induced X-ray Emission (PIXE) technique, the ion scattering techniques Rutherford Back-scattering Spectrometry (RBS) and Coincident Elastic Recoil Detection Analysis (CERDA), and the Channeling technique. Information about these analysis techniques can be found in [Voi 75]. Some of these techniques are used in experiments performed in the microprobe and channeling experimental set-ups. The ion beam is produced by a cyclotron and is guided from the cyclotron to an experimental set-up by means of a beam guidance system. Currently, microprobe experiments are performed to study element distributions in biological tissues. Channeling experiments are performed to obtain information about the structure of crystals.

A data-acquisition system is used to automate these experiments. The term data acquisition must be seen in its widest sense. The data-acquisition system is not only used to acquire experimental data, it is also used to control and monitor the experiment, to process and store the data and to analyse this data. The data-acquisition system comprises both hardware and software. At this moment both data-acquisition hardware and software are changing. The reasons for these changes are to increase the performance and the user-friendliness of the data-acquisition system. The new data-acquisition hardware and software described in this report are still in the development stage. The data-acquisition system is not yet used to automate actual experiments.

The new data-acquisition hardware is build around two processors, a front-end and a back-end processor. The two processors are connected via the local network. In addition, the front-end processor is connected to a PhyBUS. PhyBUS is an interface bus to which experiment interfaces are connected. These experiment interfaces are needed to be able to automate an experiment by means of a computer system. The differences with respect to the old hardware configuration are:

1) The type of front-end processor.
2) The network connection between the front-end and the back-end processor.
3) The connection between the front-end processor and the PhyBUS.

The changes to the data-acquisition software are:

1) A new operating system that runs on the front-end processor. This operating system is VxWorks.
2) The data-acquisition software on the back-end processor is equipped with a graphical user interface (GUI).
3) The data-acquisition software for all experiments is integrated into one software package.

The goal of this project is to make a new design for the data-acquisition software and to study the implementation of network communication between application programs that run on the two processors of the data-acquisition system.
Chapter 1 describes a model experiment. This model experiment combines all aspects of both microprobe and channeling experiments. It helps to make a new design for the data-acquisition software in which the software needed for the different experiments is integrated. Prior to the actual model experiment this chapter describes the general parts of the model experimental set-up and the new data-acquisition hardware that will be used to automate the model experiment.

Chapter 2 describes the data-acquisition software. The data-acquisition software can be divided into two parts: the operating systems that run on the processors of the data-acquisition system and, in addition, the application software that runs on these operating systems. Section 2.1 describes the operating systems and the software development tools that are used to develop the application software. First of all, the new VxWorks operating system that runs on the front-end processor and the corresponding software development tool Tornado are described. Besides for the development of applications that run on VxWorks, Tornado is also used to configure the VxWorks operating system itself. Furthermore, attention is paid to the OpenVMS/X Window system that runs on the back-end processor and the software development tool X-Designer. The combination of the operating system OpenVMS and the X Window system is needed to run the graphical user interface the data-acquisition system is equipped with. The software development tool X-Designer is used to develop graphical user interface applications for the OpenVMS/X Window system. In addition, some attention is paid to the combination of Windows95 and LabWindows/CVI. Windows95 is not used in the data-acquisition system described in this report. However, it is used as the operating system of a single-processor data-acquisition system that is used to automate smaller experiments in the group Nuclear Physics Techniques. Section 2.2 describes the new conceptual design of the application software that will be used to automate the microprobe and channeling experiments. One aspect of this concept, i.e. the graphical user interface, has been designed and is described in this section in more detail.

Chapter 3 describes the structure of the TCP/IP communication system software. This software provides the network communication between application software that runs on the two processors of the data-acquisition system. The TCP/IP communication system is used because it is supported by both the operating systems in the data-acquisition system and the specifications are publicly available. The TCP/IP communication system software is organised into four conceptual layers. The upper layer, i.e. the application layer, resides outside the operating system while the other layers reside inside the operating system. An interface is needed to pass data over the boundary of the operating system. The application layer comprises the application programs that perform network communication. The details of the network communication between these applications are handled by the successive layers inside the operating system.

Chapter 4 describes the interface between the application layer and the successive layers of the TCP/IP communication system software. More specific it describes an interface named the socket interface. This interface has become widely accepted and is used by both the VxWorks and OpenVMS operating systems. From the application's point of view this interface contains a set of routines that can be used to perform network communication.

Chapter 5 describes network communication between application programs in relation to the data-acquisition system. It describes the actual implementation of network communication into application programs running on VxWorks and OpenVMS. It studies the reliability of network communication between applications that run on the front-end and back-end processor and the network throughput. Finally it describes, how network communication can be incorporated into application programs with a graphical user interface.
Chapter 1

A Model Experiment

This chapter describes a model experiment that is based on both microprobe and channeling experiments. This model experiment incorporates common features that are the same for both types of experiments as well as features that are characteristic for a specific type of experiment only. The reason that these experiment specific features are also included in the model experiment is that the model experiment must comprises the functionality of all different experiments. This way the model experiment can be used to make a new design for the data-acquisition software in which all the software needed to automate the different experiments is integrated.

Section 1.1 describes the general parts of the model experimental set-up that is used to perform the model experiment. Like the model experiment the model experimental set-ups consists of parts that the same for both the microprobe and channeling set-up and parts that are only present in one of the two experimental set-ups.

Section 1.2 describes the new data-acquisition hardware that will be used to automate the model experiment. The new hardware configuration shows some differences with respect to the current hardware configuration. Section 1.2.1 describes the general configuration of the new data-acquisition hardware which is the same for all experiments The actual experiment interfaces that are incorporated in the data-acquisition hardware depend on the experiment that is performed. Section 1.2.2 describes the interfaces that are needed to perform the model experiment.

Section 1.3 describes the model experiment itself. The model experiment comprises more than the acquisition of experimental data only. Besides data acquisition it consists of experiment control, data processing, monitoring, storage and analysis.

This chapter gives only a brief description of the general parts of the model experimental set-up and the experiment interfaces incorporated in the data-acquisition hardware. More detailed information about the different set-ups and interfaces can be found in [Mut 95], [Dij 97], [Atz 92], [Aen 91], [Mee 96], [Sim 94] and [Beu 94].

1.1 General Parts of the Model Experimental Set-up

A cyclotron is used to produce the ion beam needed to perform the model experiment. In addition, a beam guidance system guides the beam to the model experimental set-up. Figure 1.1 depicts the general parts of the model experimental set-up. The different parts are briefly described below.

1) Beam Stop  A beam stop is present at the front of the experimental set-up. A beam stop is used to measure the beam current at a certain point in the beam guidance system. Furthermore, the beam stop can be used to prevent the beam entering the experimental set-up. The beam stop is operated manually. The operation of the beam stop will be automated by means of the data-acquisition system in the future.
2) Slits  The experimental set-up contains two pairs of slits, each pair consisting of a horizontal and a vertical slit. These slits are used to align the beam and to reduce the beam dimensions. The slits are operated manually or by means of stepper motors.

3) Quadrupole Magnets  The quadrupole magnets are used to focus the beam. They are operated by the data-acquisition system. Quadrupoles are only present in the microprobe set-up.

4) Faraday Cup  A Faraday cup is placed at the back of the experimental set-up. It is used to measure the beam current or charge impinging on the sample. The Faraday cup can only be used when there is no sample or a thin sample present in the line of the beam. The actual beam current or charge is measured and displayed by means of a Keithley electrometer that is connected to the Faraday cup. The Keithley electrometer can be operated by the data-acquisition system.

5) Rotating Vane  Another way to get an indication of the beam current and/or beam charge during the experiment is by means of a rotating vane in combination with a detector. Since the rotating vane is placed in front of the sample this method is well suited to measure the beam current and/or charge during experiments with thick samples. Beam particles that hit one of the blades of the rotating vane, when this blade passes through the beam, are scattered. These scattered particles can be detected by the detector. The count rate of the detector is used as a measure for the beam current. Integration of the count rate over a period of time is used as a measure for the beam charge. The rotating vane with corresponding detector is only present in the channeling set-up. The rotating vane is operated manually. The detector functions the same as the other detectors present in the experimental set-up. More information about detectors is given below (see Detectors).

6) Scanning Magnet  The scanning magnet is used to move the beam relative to the sample. Usually, it is used to scan the sample according to a predefined pattern. The beam is deflected by changing the magnetic field inside the magnet. The scanning magnet is operated by the data-acquisition system. The scanning magnet is only present in the microprobe set-up.
7) Sample Holder  The experimental set-up contains a sample holder that can hold a number of samples. The sample holder can be moved, with a certain accuracy and within a certain range, in a number of directions. The sample holder is moved by means of stepper motors. The axes of these stepper motors may contain decoders that provide information about the relative position of the sample holder. In addition, end switches determine the limitations of the range of the sample holder in the different directions. The stepper motors, decoders and end switches are operated by the data-acquisition system. The sample holder can be moved to place the sample that is to be studied in the beam line and to move the sample relative to the beam.

8) Detectors  The experimental set-up contains a number of detectors to detect scattered particles and/or X-rays that originate from interaction between the beam and the sample. The detectors are placed at a certain angle with respect to the beam line and at a certain distance from the sample. The position (i.e. both angle and distance) of some detectors can be changed. In addition, the angle with respect to the beam line of some of these detectors can be changed by means of stepper motors. The axes of these stepper motors contain decoders that provide information about the relative position of the detectors. End switches determine the limitations of the range of the angle. The detectors generate signals that provide information about the energy of the detected particle or X-ray. Several electronic devices are present to amplify, shape and manipulate the signals generated by the detectors. The specific way in which these signals are processed depends on the information that is desired in relation to the actual measurement that is performed. The different types of measurements are described in section 1.3. A number of ways are present to display the count rate of a detector (e.g. by means of a current digitiser or an oscilloscope). The count rate is the number of particles/X-rays detected by a certain detector per unit of time. However, when more detailed information about the detected particles/X-rays is desired the data-acquisition system is used to acquire the signals generated by the detectors (see section 1.2.2.).

1.2 Data-Acquisition System, the Hardware

This section describes the new data-acquisition hardware (see figure 1.2). The new configuration shows three major differences with respect to the current hardware configuration. These changes run parallel with a change in the operating system running on the front-end processor. The current operating system EPEP that is developed at the Physics department of the EUT is replaced by the commercial operating system VxWorks. The operating system VxWorks itself is described in more detail in chapter 2. The reason for both the change with respect to the operating system and the changes to the data-acquisition hardware is to increase the performance of the data-acquisition system. The differences in the hardware are:

1) The front-end processor. The current 25 MHz Motorola 68030 microprocessor is replaced by a 100 MHz Intel pentium processor.
2) The connection between the front-end and the back-end processor. In the new hardware configuration both processors are attached to the local Ethernet network. This is possible because, unlike EPEP, VxWorks supports network communication.
3) The connection between the front-end processor and the PhyBUS. The front-end processor is connected to a computer bus. In addition, this computer bus is connected to the PhyBUS. The main board of the Intel pentium processor contains a PCI computer bus, whereas the M68030 is connected to a VME computer bus. The PCI bus is a new standard with a higher performance. This connection is described in more detail in section 1.2.1.

The PhyBUS and the experiment interfaces connected to the PhyBUS are left unchanged. Section 1.2.1 describes the general configuration of the new data-acquisition hardware. Section 1.2.2 briefly describes the experiment interfaces that are needed to automate the model experiment performed with the model experimental set-up described in section 1.1.
1.2.1 General Configuration

The data-acquisition hardware is build around two processors, a front-end processor and a back-end processor. The front-end processor is an 100 MHz Intel pentium processor. It is equipped with 16 Mb of memory, an ISA and a PCI computer bus, and a floppy drive. This floppy drive is needed to boot the VxWorks system. More information about VxWorks in relation to the front-end processor is given in section 2.1.1. The front-end processor is connected to a PhyBUS by means of a highway. A highway is an ordinary cable that allows the front-end processor to be placed at a substantial distance from the PhyBUS. A PCI /Highway converter that is placed in on of the slots of the PCI bus connects the highway to the front-end processor. A highway/PhyBUS converter connects the highway to the PhyBUS. A PhyBUS is an interface bus to which the interfaces to the experimental set-up are connected. The front-end and back-end processors are connected via an local Ethernet network. The processors are attached to the Ethernet by means of Ethernet controller boards. The back-end processor is an 133 MHz DEC alpha-AXP processor. This processor is part of a fully equipped DEC alpha workstation. A cluster of hard disks and tape streamers are connected to the workstation.

1.2.2 Experiment Interfaces

The experiment interfaces described in this section are the interfaces that are needed to automate the model experiment. The interfaces are connected to the PhyBUS (see figure 1.2). All interfaces are currently in use with one exception, the beam stop interface. Detailed information can be found in [Mut 95], [Dij 97], [Atz 92], [Aen 91], [Mee 96], [Sim 94] and [Beu 94].
A Model Experiment

Beam Stop Interface  The data-acquisition system can use this interface to control the beam stop at the front of the experimental set-up.

I/O system/PhyPAD/Control Unit/Analogue Rack/DACs  The data-acquisition system uses digital to analogue converters (DACs) to control the settings of the quadrupole magnets. These DACs are placed in an analogue rack. In addition, the analogue rack is connected to the PhyBUS by means of a PhyPAD (Parallel Asynchronous Dataway) connection between an I/O system that is connected to the PhyBUS and a control unit in the analogue rack. The front-end processor writes the correct values to the DACs through the I/O system.

I/O Controller  The data-acquisition system uses DACs to control the settings of the scanning magnet as well. Again these DACs are placed in a analogue rack that is controlled by a control unit. However, the front-end processor does not write the values to the DACs itself through an I/O system. Since the scanning magnet is used to move the beam repeatedly over the sample according to a certain pattern, the DAC values have to be changed continuously. The data-acquisition system uses an I/O controller to write the correct values to the DACs. In addition, The I/O controller makes another value that corresponds to the current relative position of the beam with respect to the sample available to other interfaces. The way this value is used is described in below (see Multi-Channel Analysers) and section 1.3 (see Data Acquisition and Experiment Control). During operation the I/O controller functions independent of the front-end processor. It is triggered by an external clock. Up front the I/O controller receives information about the pattern from the processor.

IEEE-interface  A Keithley electrometer can be used to register the beam current measured by the Faraday cup. The data-acquisition system uses an IEEE-interface to program the Keithley electrometer to obtain the desired information (i.e. beam current and/or charge) and to make this information available to the data-acquisition system.

RS232-interface  The data-acquisition system uses a RS232-interface to operate so called micro-boards. These micro-boards control the drivers of the stepper motors for both the sample holder and the detectors. They also obtain information about the position of the sample holder and the detector from the decoders and the end switches. The RS232-interface and the micro-boards are only used in relation to the channeling set-up.

Stepper-Motor Interface  The data-acquisition system uses a stepper-motor interface to control the drivers of the stepper motors that are used to move the sample holder of the microprobe set-up.

Scaler/Preset Scaler/Clock  The data-acquisition system uses a combination of a scaler, a preset scaler and a clock to count the number of pulses at the input of the scaler during a time interval. The length of the time interval is determined by the preset scaler and the clock. For example, like the current digitiser and the oscilloscope this combination can be used to determine the count rate of a certain detector.

Multi-Channel Analysers (MCAs)  The data-acquisition system uses a multi-channel analyser (MCA) to measure the energy spectrum of the detected particles/X-rays. This type of measurement is called a histogram-mode measurement (see section 1.3). The MCA consists of an analogue-to-digital converter (ADC) and a histogram memory. The ADC converts the pulse generated by the detector, after it is shaped and validated, into a digital value. This value is a measure for the energy of the detected particle/X-ray. In addition, it is used as an address to access the histogram memory. The number at that address is increased by one each time the value occurs. The histogram memory can be accessed via PhyBUS.

The data-acquisition system uses a multi-channel analyser in combination with an additional memory, to measure the energies of the individual particles/X-rays that are detected by a detector. Again the ADC converts
valid pulses into digital values. However these values are listed in a separate memory that is accessed via a PhyPAD (Parallel Asynchronous Dataway). In addition, this memory can be accessed via PhyBUS. This type of measurement is called a list-mode measurement.

Sometimes the energy of the detected particle/X-ray is measured in relation to another parameter. This type of measurement is called a coincident list-mode measurement. For example, the energy of the detected particle is measured in relation to the energy of another particle detected by another detector at the same time. In that case two MCAs are used. The signals that arrive at the two MCAs are only valid when they arrive simultaneously. The valid signals are converted to digital values and stored in two separate lists. The ith value in one list corresponds to the ith value in the other list. Both lists may be stored in the same memory. Another possibility is the relation between the energy of the detected particle/X-ray and the position of the beam on the sample at the time of that the particle/X-ray is detected. In this case only one MCA is used. When a signal generated by a detector is valid, this value is converted to a digital energy value and stored in a memory. At the same time the I/O controller (see above) is triggered. In addition the value corresponding to the current relative position of the beam with respect to the sample that is available at the I/O controller is stored in another memory.

1.3 A Model Experiment

The actual measurement performed during the model experiment is the detection of scattered particles and/or X-rays upon the interaction of a beam of charged particles with a sample. However, the model experiment comprises much more. Besides the energy of the scattered particles and/or X-rays, other data is acquired. This additional data can be both directly and indirectly related to the actual measurement. To be able to acquire all this data the model experiment must be controlled. The experiment control consists of actions that are not directly related to the acquisition of data itself but that are needed to be able to acquire data. For example, changing the sample that is to be analysed and operating the beam stop to prevent the beam entering the experimental set-up while the sample is being changed. Furthermore, the acquired data may have to be monitored during the experiment and/or stored for further off-line analyse. The data may be processed before it is monitored or stored. This section discusses these different aspects of the model experiment. It does not specify whether a certain task is performed by the experimenter, an electronic device or the data-acquisition system.

Data Acquisition and Experiment Control

The actual measurement consists of the detection of scattered particles and/or X-rays. The experimenter may be interested in the detected particles/X-rays in four different ways:

1) The number of particles/X-rays detected by a detector during a certain time interval. For a short time interval this results in the count rate of the detector. The count rate of the detector that is used with the rotating vane is used as a measure for the beam current. For a time interval equal to the duration of the actual measurement this is a measure for the total beam charge during the measurement.

2) The number of particles/X-rays detected as a function of the energy of the particles/X-rays, i.e. an energy spectrum. When the experimenter is interested in this information the measurement is called a histogram-mode measurement.

3) The energy of each individual particle/X-ray detected by a detector. This type of measurement is called a list-mode measurement.

4) The energy of each individual particle/X-ray detected by a detector in relation to another parameter. This is called a coincident list-mode measurement. The other parameter may be another particle detected by another detector at the same time. This type of measurement is performed when the CERDA analysis technique is used. The other parameter may be the position at which the particle hits a position sensitive detector. When a position sensitive detector detects a particle it generates three signals. One signal corresponding to the energy
A Model Experiment

of the detected particle and two signals corresponding to the position at which the particle hits the detector. A third possibility for the other parameter is the position of the beam relative to the sample at the time that the particle/X-ray is detected. This type of measurement is performed during microprobe experiments in which the scanning magnet is used to move the beam over the sample.

The experimenter may want to perform the actual measurement as a function of the position of the beam with respect to the sample. Moving the beam relative to the sample is part of the experiment control. There are two possible ways to perform the actual measurement as a function of the position:

1) Perform the measurement at a certain position, change the position of the beam with respect to the sample, perform another measurement and so on. Usually the sample is displaced to change this position. The position can be changed according to a predefined pattern or based on direct interpretation of the measured data by the experimenter. This way is usually used in relation to the first two types of measurement described above.

2) Move the beam relative to the sample according to a certain pattern during the measurement. This way is used in coincident list-mode measurements performed during microprobe experiments. The scanning magnet is used to move the beam relative to the sample. Besides the energy of the individual particle/X-rays the position of the beam relative to the sample at the time the particle/X-ray is detected is obtained.

Besides the position of the beam relative to the sample the experimenter may need other information related to the actual measurement. This information may be needed to process the data on-line and/or to analyse the data off-line. Examples of this kind of information are: the duration of the measurement, a measure for the beam charge and the angles of the detectors with respect to the beam. On the other hand this information may serve as background information only. For example, the date of the experiment and the beam dimensions. Some of this information is already available before the experiment takes place (date, detector angles). Other information must be acquired during the experiment (beam dimensions) or even during the actual measurement (duration of the measurement, beam charge). Actually, information about the beam dimensions is obtained by performing an actual experiment. Information about the beam dimensions can only be obtained in case of microprobe experiments.

Next to performing actual measurements and measurements to obtain additional information there are other parts of the experiment that need to be controlled and/or data is obtained from. This data may be used to take further action upon or it may be stored for use in other experiments. For example:

1) Before the beam can be used to perform actual measurements the beam must meet some requirements. The beam must have the correct beam current. This beam current is measured by means of the Faraday cup. The beam must be aligned and focused. The slits and quadrupoles are used for this purpose. The correct settings of both the slits and quadrupoles can be stored for use in other experiments.

2) The sample holder must be operated. Beside to perform measurements as a function of the position of the beam relative to the sample the sample holder must also be moved to change the sample. The position of the sample holder must be known at all time. It must be stored to know the position of the sample at the beginning of another experiment. In addition, some characteristic positions may be saved for further use.

3) The beam stop must be operated and information about the current situation must be present.

Data Processing

Before the acquired data is monitored or stored and, in addition, analysed off-line it may be processed to obtain relevant information only. This reduces the amount of data in further calculations, transport and/or storage. For instance, all channels in an energy spectrum may be summed to obtain the total number of counts. Another example is the application program Dyana that can be used for on-line analysis of experimental data obtained from microprobe experiments on biological tissues.
Data Monitoring
The experimenter may want to monitor data acquired during the experiment. This data can be a measure for the beam current to check the beam during the experiment. It can be the current position of the sample holder. Or it may be a representation of the course of an actual measurement. For instance, by means of a spectrum or a distribution plot.

Data Storage
The experimenter may want to store data related to and obtained during the experiment. This data may comprise the actual (reduced) measured data, data needed for further analysis of this data, background information related to measurements, information related to other aspects of the experiment and information about the course of the experiment.

Off-line Data Analysis
The measured data is analysed by the experimenter to obtain the desired information about the sample after the experiment is finished. The analysis of the data takes too much time to perform it during the experiment.
Chapter 2

Data-Acquisition System, the Software

This chapter discusses the software of the new data-acquisition system. The data-acquisition software can be divided into two parts. First, the operating systems running on the processors of the data-acquisition system. These operating systems are described in section 2.1. This section also describes the software development environments that are used to develop applications that run on these operating systems. Second, the application software that runs on these operating systems. Section 2.2 discusses a new design for the application software that is used to perform microprobe and channeling experiments. Furthermore, a more detailed description of one aspect of this concept, that is the graphical user interface, is present in this section.

2.1 Operating Systems and Software Development Environments

The operating system that runs on the front-end processor of the data-acquisition system is changed from EPEP to VxWorks. VxWorks is a commercial operating system, whereas EPEP is developed at the Physics department of the EUT. Like the reason for the changes to the data-acquisition hardware the reason for this change is to increase the performance of the data-acquisition system. The change to the VxWorks operating system has consequences for the application software of the data-acquisition system. These changes are discussed in section 2.2. Section 2.1.1 describes the VxWorks operating system and the corresponding software development environment Tornado.

The operating system running on the back-end processor of the data-acquisition system is OpenVMS. The graphical user interface of the data-acquisition system runs on this processor. An X Window system is placed on top of this operating system to be able to run applications programs with a graphical user interface. Application programs with a graphical user interface for the data-acquisition system are developed using the development program X-Designer. Section 2.1.1 gives a brief description of the OpenVMS operating system, the X Window system and X-Designer.

The data acquisition in relation to microprobe and channeling experiments is too large to replace the DEC alpha workstation that serves as back-end processor (see figure 1.2) by a PC running Windows95. However, for the automation of smaller experiments the back-end processor of the data-acquisition system may be replaced by a PC. In addition, for even smaller experiments the data-acquisition system may even be build around a single PC. In this case a single processor provides all the functionality of both the front-end and the back-end processor. This means that the graphical user interface runs on this processor as well. This hardware configuration is used to automate experiments at the group Nuclear Physics Techniques as well. The software development environment LabWindows/CVI is used to develop application programs with a graphical user interface to run on the operating system Windows95. Section 2.1.3 briefly describes LabWindows/CVI.
### 2.1.1 VxWorks and Tornado

Both the VxWorks operating system and the Tornado development system are products of Wind River Systems, Inc. Actually VxWorks is a component of the Tornado development system. Both VxWorks and Tornado are described in this section. Information about VxWorks and Tornado can be found in [VxW 95a] and [Tor 96].

The VxWorks operating system will be used to run on the front-end processor. The VxWorks version that is currently used in the development stage of this data-acquisition system is version 5.3. VxWorks is a high-performance real-time operating system. The VxWorks kernel, wind, is at the heart of the operating system. It includes, amongst others, multitasking with pre-emptive priority scheduling, intertask synchronisation and communication facilities and interrupt handling support. A multitasking environment allows real-time applications to be constructed as a set of independent tasks, each with a separate thread of execution, its own set of system resources and a certain priority. The task priority determines the processor time available to a certain task in relation to the other tasks. The intertask communication facilities allow the tasks to synchronise and co-ordinate their activity. In addition, VxWorks provides a great number of facilities. For example, an I/O system and I/O drivers, local and remote file systems, network facilities and utility libraries. One can configure VxWorks to include only those facilities that are required to the applications that run on the system. This will save resources.

Tornado is a cross-development system that is used to configure the VxWorks operating system and to develop application programs that run on VxWorks. Tornado itself runs on a PC that is running Windows95. The PC that runs Tornado is called the host system. The system that runs VxWorks is called the target system. In relation to the data-acquisition system this is the front-end processor. Both systems are connected via the same local Ethernet network as described in section 1.2.1 (see figure 1.2). Note that the host system is not the same as the back-end processor of the data-acquisition system. Tornado includes a number of interactive development tools (see figure 2.1). The different tools are described below.

![Tornado Interactive Development Tools](image-url)

**Figure 2.1:** Tornado interactive development tools and the communication between the host and the target system.
WindConfig is the VxWorks configuration tool. It provides a way to choose the optional features to link into the VxWorks run-time system. For example, for the VxWorks system on the front-end processor network communication is included, whereas target-resident debugging facilities are excluded. When the configuration is completed a VxWorks system image is created and stored on the host system’s hard disk. This system image is a binary module that can run on a target system. The target system uses a boot diskette to copy this system image to the target system. This boot diskette is created by Tornado as well. This boot diskette contains amongst others software to run a FTP client on the target system. Furthermore, it contains information about the system that contains the VxWorks system image (in this case the host system). When this system runs a FTP server the system image can be transmitted from this system to the target system via FTP. In addition, the target system can run the VxWorks system image.

Tornado provides an editor to write the source code for VxWorks application programs. The application source codes modules, written in C or C++, can be compiled with the GNU cross-compiler provided as a part of Tornado.

Furthermore, Tornado comprises a number of tools that interact with the target system at run-time basis. One of these tools is WindSh, the host-resident Tornado Shell. For example, this shell can be used to load the compiled application object modules into a running VxWorks system by using the function Id(). These object modules are linked to VxWorks dynamically. In addition, it allows you to interactively execute all functions loaded on the target, including both VxWorks functions and application functions. Another tool is the Tornado browser. This browser provides display facilities to monitor the state of the target system. For example, memory allocations. CrossWind is the Tornado source-level debugger.

Communication between these Tornado development tools and the VxWorks operating system takes place via a connection between a target server and a target agent (see figure 2.1). The target server is an additional host-resident application that can be started from within the Tornado development environment. The function of the target server is to manage the communication between the Tornado tools and VxWorks. In some cases a tool’s service request is passed directly to the target agent. However, in other cases the request can be partially or completely fulfilled within the target server itself. For example, the target server manages the target’s symbol table on the host. This permits the target server to do most of the work of dynamic linking on the host system, before downloading a new module to the target. Furthermore, when a target-memory read hits a memory region already cached in the target server, no actual host-to-target transaction is needed. On the target, all Tornado tools are represented by the target agent. The target agent resides beneath the VxWorks kernel on the target system and operates independent of the operating system. The target agent responds to requests transmitted by the target server, and replies with the results.

The situation above describes the interaction between the Tornado development system and the VxWorks target system during the configuration of the VxWorks operating system and development of application programs. In the future, when used in the data-acquisition system, the target system runs independent of the host system. In that case both the VxWorks operating system and the application programs developed to run on the front end of the data-acquisition system are booted from an arbitrary disk attached to the Ethernet by means of a boot diskette. For example, this can be a disk connected to the back-end processor. In this situation the host system is superfluous.

2.1.2 OpenVMS/X Window and X-Designer

The operating system that runs on the back-end processor of the data-acquisition system is OpenVMS AXP version 6.1, a product of Digital Equipment Corporation (DEC). OpenVMS is a multitasking/multi-user operating system.

The graphical user interface of the data-acquisition system runs on the back-end processor. However, OpenVMS is not able to run applications with a graphical user interface. A X Window system is layered on top of the OpenVMS operating system to provide this functionality.

X-designer is a software development tool for building applications with graphical user interfaces that run on an OpenVMS/X Window system. X-designer itself is equipped with a graphical user interface and runs on an
OpenVMS/X Window system as well. Building an application with a graphical user interface with X-designer involves the following stages:

1) Designing the graphical user interface. X-designer provides an interactive editor to build a graphical user interface using so-called widgets as building blocks. Examples of such widgets are: PushButton, Label and MenuBar.

2) Generating code. X-Designer automatically generates all the C Code needed to display and operate the interface.

3) Writing code. The user must write the source code that must be executed when the user interacts with the interface.

4) Compiling, linking, running and debugging of the application program.

More information about X-Designer can be found in [XDe 95]

### 2.1.3 Windows95 and LabWindows/CVI

Like X-Designer, LabWindows/CVI, is a software development system to build graphical user interface applications. The different stages of the development cycle are the same as with X-Designer. LabWindows/CVI contains an interactive environment for editing, compiling, linking, running and debugging applications with a graphical user interface. However, both LabWindows/CVI and the applications that are developed with LabWindows/CVI run on the Windows95 operating system.

The LabWindows/CVI software development system is especially designed for creating data-acquisition and instrument control applications. It comprises a set of function libraries for data acquisition, data analysis, data presentation and network and interprocess communication. Furthermore, it provides graphical user interface building blocks that are related to data acquisition and experiment control. For example, complete strip charts, meters to display a certain parameter (e.g. current) and LEDs. More information about LabWindows/CVI can be found in [Lab 94a].

### 2.2 Application Software

The application software is undergoing a lot of changes. There are three major reasons for a thorough revision of the application software at this point. First, the new VxWorks operating system on the front-end processor and the changes in the hardware configuration corresponding to it. New software is needed to take advantage of the real-time features of VxWorks, to address the PCI computer bus and to transmit data via the network connection between the two processors. In addition, software that in principle can stay the same must be translated from the programming language EPEP to the C language. Second, the data-acquisition system is equipped with a graphical user interface. And third, the integration of the data-acquisition software for all experiments into one software package. This has been the goal from the beginning, however, the emphasis has always been on the microprobe experiments. The software for the other experiments is more or less added to the existing software. Besides the changes to the application software a lot of the software and/or the current ideas about the software may stay the same.

Section 2.2.1 describes a conceptual design for the new data-acquisition application software. This concept forms the starting point for the further design and implementation of the data-acquisition application software. The different components of this concept (see figure 2.2) can be designed and implemented relatively independent of each other. The design and implementation is based on the model experiment described in chapter 1. Of course details will depend on the specific experiment. Finally, the different components can be combined to form the complete data-acquisition application software. Section 8.2.2 describes a more detailed design of the graphical user interface part of the concept. The other parts of the conceptual design are for further investigation.
2.2.1 Conceptual Design

The conceptual design for the new data-acquisition software comprises four major components: a graphical user interface, on-line and off-line application software and data storage. In addition, some of these components are divided into smaller parts. The conceptual design is depicted in figure 2.2.

![Diagram of data-acquisition system](image)

**Figure 2.2**: The conceptual design of the data-acquisition application software. Explanation: see text.

The graphical user interface is the starting point of all interaction between the experimenter and the model experiment as described in section 1.3. The graphical user interface is situated on the back end of the data-acquisition system.

The underlying software provides the actual functionality of the data-acquisition application software. This software can be divided into on-line and off-line software.

The off-line application software consists of software for off-line analysis of experimental data and data management. The off-line analysis software contains application programs to analyse the experimental data that was acquired during actual measurements. The data management software comprises, for example, software to make back-ups of experimental data, software to access a database to store and/or obtain information related to experiments and software to access log files to obtain information about the course of previous experiments. The off-line application software runs on the back-end processor.

The on-line application software incorporates software for the control of the experiment, data acquisition and data processing. The on-line software is divided over both the front-end and the back-end processor. Therefore, network communication software forms an additional part of the on-line software. The different aspects of the on-line application software are closely related. They do not form completely separated parts of the on-line software.
Data can be stored on disk and/or tape. Both storage media are situated on the back end of the data-acquisition system. This data can be all kind of data related to the experiment. The sort of data determines the way the data is stored. For example, the data can be stored in experimental-data files, log files or a relational database. Experimental-data files can be used to store experimental data or experimental data that is analysed off-line. Log files can be used to store chronological information about the course of a certain experiment. A relational database can be used to store all kind of information related to the experiments. For example: Experiment specific information like the date and a description of a certain experiment. Information about the settings of the multi-channel analysers in relation to type of measurement (histogram mode, (coincident) list mode). The later information can be used in relation to all experiments that are performed. Both off-line and on-line application software can store and/or access data.

2.2.2 The Graphical User Interface

The graphical user interface is the starting point for all interaction between the experimenter and the experiment. It is designed from the experimenter's point of view. That is, the graphical user interface provides all functionality needed by the experimenter but hides the details of how this functionality is incorporated in the underlying application software. For example, to perform a highly automated measurement the experimenter only needs to specify a number of parameters in a certain window of the graphical user interface and start the measurement. However, the actual application software that is needed to perform this measurement will be much more complex. This section discusses the current status of the design of the graphical user interface component of the conceptual design.

The Main Menu

Upon start-up of the data-acquisition software the graphical user interface consists of only a menu structure. This menu is called the main menu and is depicted in figure 2.3. The menu items of the main menu are: Experiment Control and Monitoring, Off-line Analyse and Data Management.

The menu item Experiment Control and Monitoring contains options to control and monitor experiments performed using a certain experimental set-up. Current options are: Microprobe and Channeling. Selecting one of these options creates the Experiment Control and Monitoring window for that specific set-up. One of the demands with respect to the design of the data-acquisition software is to integrate the software needed for the automation of the different experiments into one software package. A model experiment is defined for this purpose. However, the graphical user interface is designed from the experimenter's point of view. An experimenter performs a specific experiment using a specific experimental set-up rather than a model experiment. Although microprobe and channeling experiments show a lot of similarities the differences between these experiments are too large to use a general Experiment Control and Monitoring window to control and monitor model experiments. Therefore, each type of experiment is provided with its own Experiment Control and Monitoring window. However, the structure of the different Experiment Control and Monitoring windows is the same. The Experiment Control and Monitoring window is discussed in more detail below.

The menu item Off-line Analysis contains options to perform off-line analysis on experimental data. The current options comprise shortcuts to existing off-line analysis application programs. Examples of these off-line analysis applications are Paneut, Verstim and the Egg. In the future the interfaces to these applications may be further integrated into the graphical user interface of the new data-acquisition system.

The menu item Data Management contains options to store and/or access data independent of a certain experiment or off-line analysis. For instance to make a back-up of experimental data or to access the database to specify or obtain certain data. The actual options are not yet specified.
Figure 2.3: The main menu of the graphical user interface of the data-acquisition system. Selecting an option from the menu item Experiment Control and Monitoring opens a window that is used to perform the type of experiment specified by the option. The menu item Off-line Analysis provides shortcuts to off-line analysis applications. In the future the interfaces to these applications may be further integrated into the graphical user interface of the data-acquisition system. The menu item Data Management provides the functionality to store and/or access data independent of a certain experiment or off-line analysis.

The Experiment Control and Monitor Window

The Experiment Control and Monitoring window is used for the control and monitoring of experiments. Both microprobe and channeling experiments are provided with a different window. However, the design of the Experiment Control and Monitoring window is based on demands with respect to both microprobe and channeling experiments. Therefore, the structure of this window is the same for both types of experiments. The window contains a menu structure and a status window. Interaction between the experimenter and the experiment is initiated by selecting an item from the menu structure. The status window only provides the user with information about the current status of the experiment.

The menu items of each Experiment Control and Monitoring window are:
1) Move
2) Measure
3) Tools
4) An Experiment Specific Item, this item contains experiment specific functionality and therefore is different for the microprobe and the channeling Experiment Control and Monitoring windows.

In addition, the options that are available under each menu item show a lot of similarities for the different Experiment Control and Monitoring windows as well. The functionality that is provided by the different menu items is discussed below. The examples that are used in this discussion are only related to the microprobe window. In that case the Experiment Specific Item is called Focus Beam.
1) Move

The menu item Move contains sub items that specify the objects in the experimental set-up that can be moved. In addition, these sub items contain options related to the movement of the specified object. The general idea behind these options is the same for all objects. The options comprise options related to the co-ordinate system that is used to describe the position of the object and options related to the actual movement of the object. For microprobe experiments the only sub item is Sample Holder. This sub item contains the following options: WCS, UCS, Rotate and Translate (see figure 2.3).

The position of a certain object in the experimental set-up is described by means of a so-called world co-ordinate system (WCS). The data-acquisition system does not provide a mechanism to obtain the absolute position of the object in this co-ordinate system. The data-acquisition software keeps track of this position by means of the relative displacement of the object with respect to the current position. When for some reason the current position is lost the experimenter uses the option WCS to move the object back to the origin of the WCS. Selecting this option moves the object in all possible directions until the end switches are reached. In addition, when an end switch is reached the object is moved a predefined distance in the opposite direction, that is to the origin of the world co-ordinate system.

For a matter of convenience the experimenter may want to define another position of the object as the origin of the co-ordinate system. For instance, when the object is moved around a certain point in the WCS. Selecting the option UCS defines the current position as the origin of a user co-ordinate system (UCS) and asks the experimenter whether or not to store this UCS for use in other experiments. The underlying application software translates the UCS co-ordinates into WCS co-ordinates.

The options related to the actual movement of the object depend on the degrees of freedom of the object and how these degrees of freedom can be combined into different options. For the sample holder of the microprobe set-up this results in two options: Rotate and Translate. The reason is that rotation and translation of the sample holder are usually performed independent of each other. The sample holder is usually rotated to change the sample that is placed in the line of the beam. The sample holder is translated to move the sample relative to the beam. Selecting these options opens corresponding windows that provides a number of ways to move the object. For example, by specifying the new co-ordinates or using arrow keys. The status window of the Experiment Control and Monitoring window contains information of the current position of the object in both the UCS and WCS.

2) Measure

The menu item Measure comprises functionality related to performing actual measurements. The actual measurements that are performed with the current data-acquisition system are usually highly automated. There is little interaction between the experimenter and the experiment during the measurement. The user specifies a number of parameters and then starts the measurement. Examples of these parameters are: the number of detectors that are used, whether the data must be acquired in histogram mode and/or (coincident) list mode and the duration of the measurements, and information about how the data must be monitored.

Selecting the menu item Measure activates a window in which the experimenter specifies these parameters and/or some new parameters and afterwards starts the measurement. The experimenter can use the database to obtain values about certain parameters that were stored after use in prior measurements, when the values are the same for the current measurement. In addition, the experimenter may store the current values in the database. During the measurement a new window is created to monitor the acquired data. Information about how the data is to be monitored can be specified in the parameter list as well. Usually the data is monitored by means of two types of graphs: spectra and/or distribution plots. This window provides the functionality to manipulated these graphs (e.g. to change the scale of the axes and to select a region of interest). Furthermore, this window provides a way to interrupt the measurement when the experimenter is not satisfied with the course of the measurement.
3) Tools
The menu item Tools contains options that provide stand-alone functionality that might be needed during the experiment. Current options are: Beam Stop and Spectrum.

   The option Beam Stop can be used to operate the beam stop that is placed at the front of the experimental set-up. The beam stop may be operated as part other actions during the experiment as well. For example, it may be incorporated in software that is used to change samples, to stop the beam automatically before the sample is changed. However, the experimenter may want to operate the beam stop for other reasons directly.

   Selecting the option Spectrum performs an energy spectrum measurement, using a predefined set of parameters. The experimenter may be interested in an energy spectrum at a certain point during the experiment to obtain some addition information about the experiment. Such a measurement can be performed by using the menu item Measure as well. However, in that case the experimenter must specify a number of parameters itself. The option Spectrum is included as a separate tool for a matter of convenience.

4) The Experiment Specific Item - Focus Beam
The Experiment Specific menu item of each Experiment Control and Monitoring window incorporates functionality that is needed to control and monitor aspects of the type of experiment of concern that are not know to other types of experiments. Therefore, this menu item is different for the different types of experiments.

   With respect to microprobe experiments such an aspect is the focusing of the ion beam by means of the quadrupole magnets that are present in the experimental set-up (see figure 1.1). The beam is focused in a number of steps:

1) The data-acquisition system uses DACs to control the current supply to the quadrupoles. The experimenter supplies a set of DAC values that approximately correspond to the desired beam dimensions (usually 3x3 μm). These values are obtained during prior experiments. Since the properties of the beam are different during each experiment these values do not automatically result in the desired beam dimensions. They are used as a starting point for the further focusing of the beam. In addition, the quadrupoles are saturated. For a period of 30 seconds the maximum current is supplied to the quadrupoles. Afterwards the initial settings are restored.

2) Next the experimenter focuses the beam roughly by changing the DAC values based on the visualisation of the beam spot on a monitor. To do this a sample of ceramic material is placed in the line of the beam. A property of this ceramic sample is that it becomes fluorescent upon interaction with the beam, thereby revealing the beam dimensions. This beam spot is made visible to the experimenter on a monitor connected to a photo camera that is present in the experimental set-up. This part is called manual focusing because the experimenter changes the DAC values himself.

3) The fine tuning of the beam dimensions is performed automatically. That is, the corresponding DAC values are calculated by application software. To do this a focus sample must be positioned in the line of the beam in such a way that the beam impinges on the focus sample at a specific point. The focus sample that is most frequently used is a cross wire. Other focus samples are an edge and a chip. For the cross wire this position is the intersection of the two wires. At this moment the focus sample can not be positioned automatically by specifying the correct co-ordinates to the data-acquisition system. Therefore, the focus sample must be positioned manually. To do this the experimenter moves the focus sample and at the seem time watches the count rate of one of the detectors. This count rate provides information about the position of the focus sample with respect to the beam. For example, for the cross wire a high count rate means that the beam impinges on one of the wires of the cross wire. By searching the position with the highest count rate for both the x and y direction the intersection of the two wires can be found.

4) When the focus sample is placed at the correct position the beam can be focused automatically. To do so the beam is moved relative to the focus sample automatically according to a predefined pattern. During this trajectory the count rate of the detector is measured. This information is used to measure the beam dimensions. In
addition, new DAC values are calculated based on the measured dimensions. This is repeated until the experimenter is satisfied with the beam dimension.

The Experiment Specific menu item that provides all the functionality needed to control and monitor the focusing of the beam as described above is called Focus Beam. The Focus Beam menu item contains three options: Manual Focus, Position Focus Sample and Auto Focus. These options and their relation to the different steps are discussed below.

**Manual Focus**  This option provides the functionality that is needed to perform the first two steps described above. Selecting this option opens a window in which the settings (i.e. the DAC values) of the quadrupole magnets can be specified. These settings can be specified in two ways. First, the settings can be selected from a list of predefined or previously used settings. This way is used to specify the DAC values that are used as a starting point for the further focusing of the beam. Second, the settings can be specified by the experimenter himself. For example, by using the arrow keys to change the current settings. This way is used to adjust the settings during the focusing of the beam based on the visualisation of the beam spot on the ceramic sample. The experimenter can include the current settings in the list of settings for future use. The current quadrupole magnet settings are monitored on the status window. Furthermore, the window provides a way to saturate the quadrupole magnets and to place the ceramic sample in the line of the beam. For example, by means of command buttons.

**Position Focus Sample**  This option provides the functionality to place the focus sample at the correct position with respect to the beam manually. Selecting the option Position Focus Sample first opens a window in which the experimenter must specify the following parameters. The focus sample and the detector to be used and some additional parameters to be able to obtain the count rate of the detector by means of the scaler described in section 1.2.2. When these parameters are specified the proper focus sample is moved in the line of the beam automatically. In addition, a window is opened that the experimenter uses to move the focus sample. This window is the same window that is opened when the experimenter selects the option Translate under the menu item Move. A second window is created in which the count rate of the detector is plotted as a function of the position of the focus sample. The experimenter moves the focus sample to the correct position based on the information displayed the second window. For the cross wire this is the position with highest count rate in both the x and y direction. This position is used as a starting point for the automated focusing of the beam.

**Auto Focus**  Finally, when the focus sample is placed at the proper position the option Auto Focus is used to focus the beam automatically. Like with the option Position Focus Sample selecting this option opens a window in which a number of parameters must be specified. Besides the parameters described above this window contains parameters to specify the trajectory of the focus sample. In addition the window contains a command button to start the procedure. An new window is created that informs the experimenter about the measurements. In addition this window provides the experimenter with the possibility to stop the measurements when he is satisfied and to confirm that these settings must be used and maybe stored for further use.
Chapter 3

The Structure of TCP/IP Communication System Software

TCP/IP is an internet communication system that is officially named the TCP/IP Internet Protocol Suite. TCP/IP stands for Transport Control Protocol/Internet Protocol. TCP and IP are the two main protocols incorporated in the TCP/IP protocol suite. A protocol specifies a part of the communication system. TCP/IP originated from research that led to the construction of one global internet, also known as the Internet. An internet is an interconnection of networks that may use different hardware technologies. The TCP/IP internet communication system specifies a mechanism for applications to communicate across an internet independent of their physical network connections, by hiding the details of network hardware. The communication system itself does not comprise software or hardware.

The two processors in the data-acquisition system are connected to the same physical network (see section 1.2). Communication between applications running on these processors could have been based on a communication system that specifies communication between applications on a single network only. The reason TCP/IP is used is that, unlike proprietary internet and network communications systems available from one specific vendor, the specifications of TCP/IP are publicly available. Therefore, TCP/IP is implemented in many operating systems, including the ones used in the data-acquisition system, and even more important a great deal of information is available about TCP/IP. The term network communication is used to address both communication across an internet and a physical network from this point on.

This chapter describes the general structure of the TCP/IP communication system software as implemented in the operating systems. Section 3.1 gives an overview of the conceptual layers of the TCP/IP communication system software. These layers are described in more detail in the sections 3.2 to 3.5 respectively. More information about TCP/IP can be found in [Com 95].

3.1 The Conceptual Layers

The TCP/IP communication system software is organised into four conceptual layers (see figure 3.1): Application layer, Transport layer, Internet layer and Network interface layer. Providing network communication is a complex task. Therefore, this task is divided into smaller tasks that can be performed sequentially. Each layer of the communication system software performs one of these tasks. The specific tasks of the different layers are discussed in the next sections. Each of these layers comprises one or more protocols from the TCP/IP Internet Protocol Suite.
Figure 3.1: The 4 conceptual layers of the TCP/IP internet communication software and the objects passed between these layers. Each layer comprises one or more TCP/IP protocols. The socket interface provides a mechanism to pass objects from the application to the transport layer across the boundary of the operating system.

Sending data from one application program to another means transferring the data down through the successive layers of the communication system software on the sender’s side, across the network and up through the successive layers on the receiver’s side. In general, data that is passed between two successive layers is called an object. The actual name of a certain object depends on the specific layers (see figure 3.1). When an object is passed down through the successive layers of the software the object received from a higher layer is provided with a header, containing additional information. Both header and the object form the object passed on to a lower layer. When an object is passed from a lower layer to a higher layer the header is removed from the current object and the remaining object is passed to the next layer (see figure 3.2).

The application layer resides outside the operating system, while the lower layers reside inside the operating system. Passing data across this boundary is more complex than passing data inside the operating system. An interface is needed that provides a mechanism to pass the data from the application to the transport layer across this boundary. This interface is described in chapter 4.
Objects Passed Between Layers

![Diagram showing the objects passed between layers.]

**Figure 3.2:** The objects passed between the conceptual layers. Each object passed from a higher layer to a lower layer is encapsulated in the object of a lower layer, provided with additional information by means of a header. When the object is passed in the other direction the header is removed from the current object and the remaining object is passed to a higher layer.

### 3.2 Application Layer

The application layer comprises all application programs that perform network communication. The data that the application programs send to and/or receive from each other are individual messages or continuous streams of bytes, depending on the type of transport the applications use. In fact, the actual transport of data between application programs is handled by the transport layer. The application programs pass the data to the transport layer for delivery or receive the data from this layer upon arrival.

Each application program is uniquely identified by a communication endpoint. A communication endpoint comprises a port number and a TCP/IP internet address. Each application program uses a port to communicate with the transport layer software. Each port has a corresponding port number. More information about the interface between the application and transport layer can be found in chapter 4. In addition, each computer attached to a TCP/IP based internet is assigned a unique TCP/IP internet address, also called IP address. This IP address is used in all network communication related to that computer. An IP address encodes the identification of the network to which the computer attaches as well as the identification of the computer on that network. Providing a method to uniquely identify each computer that attaches to an internet is an essential aspect of hiding physical network details.

Most of the network communication between two application programs is based on the client-server model of communication. One of the applications is waiting to be contacted by the other application. This other application initialises the communication between the two application programs by contacting the waiting application. Often the initiating application requests a service from the waiting application. Therefore, this initiating application is called a client, the waiting application a server and the way they communicate the client-server model of communication.
The TCP/IP internet protocol suite includes protocols for a number of application services, called standard application services. Examples are the TELNET protocol for remote login, the Simple Mail Transfer Protocol (SMTP) for electronic mail, File Transfer Protocol (FTP) and Network File System (NFS) for remote file transfer and access and RPC for remote procedure calls. Customised services designed for personal use are called non-standard application services. Both standard and non-standard services can build upon existing services to provide new services themselves. For example, NFS uses remote procedure calls to provide a remote file access service. The actual client and server application programs that access and provide these services can be provided by the vendor of the operating system, other vendors or written for personal use.

3.3 Transport Layer

The transport layer takes care of the actual transport of data from one application program to another. The transport layer software encapsulates the data to be transmitted into one or more transport protocol packets, depending on the amount of data to be transmitted. A packet contains a header and a data area. The data area contains the data to be transmitted. The header contains, amongst others, the port numbers of both source and destination application program. The destination port number is included to deliver the data to the correct application on the remote computer (see below). The source port number is included to inform the destination application about the source of the data. An application program is identified by both a port number and an IP address. The transport layer includes only the port numbers in the header of a packet because it handles transport between application programs. It passes each packet to the internet layer for transport from one computer to another on an internet. The internet layer uses the IP address to deliver the data to the correct computer. On the remote computer the transport layer receives the packets from the internet layer. It uses the destination port number in the header of a packet to deliver the actual data in the data area to the correct application program.

The transport layer comprises two ways of transport, reliable connection-oriented and unreliable connectionless transport. These ways of transport correspond directly to the two major transport layer protocols included in the TCP/IP protocol suite, the transport control protocol (TCP) and the user datagram protocol (UDP) respectively.

TCP uses different mechanisms to provide a reliable connection-oriented stream delivery service. For example, before data is transported between two application programs the TCP software modules on both sides communicate to check whether the data can be transported. In addition, during transport TCP verifies that data arrives by a technique called positive acknowledgement, and automatically retransmits packets that are lost. The transport protocol packets are called segments, when TCP is used as transport protocol. TCP also uses sequence numbers to ensure that the data can be reorganised when the individual segments arrive out of order and to automatically eliminate duplicate packets.

UDP provides an unreliable connectionless datagram delivery service. The transport protocol packets are called user datagrams when using UDP as transport layer protocol. UDP merely forwards individual datagrams from one application program to another. UDP adds no mechanism to control the transport of the datagrams. The control of data transport depends on the underlying layers. These layers provide an unreliable, connectionless delivery service as well. When data is transmitted using UDP the data may be lost, duplicated, delivered out of order. However, UDP is well suited to provide transport between application programs in situations where reliability is not that important or data corruption is limited. In addition, because UDP comprises less computational overhead the data throughput between two application programs may be higher than when TCP is used.
3.4 Internet Layer

The internet layer handles the data transport from one computer to another on an internet. The internet layer encapsulates a transport protocol packet in an IP datagram. Like a transport protocol packet an IP datagram contains a header and a data area. The data area contains the transport protocol packet. The information in the header includes both source and destination IP addresses and the transport layer protocol used at the transport layer. The internet layer passes the IP datagram to the network interface layer for transport on the underlying physical network. At the destination the internet layer receives the IP datagram from the network interface layer. The internet layer software uses the information about the transport layer protocol to pass the transport protocol packet to the correct transport protocol software in the transport layer.

The network interface layer software handles transport between computers on the same physical network only. The internet layer must determine whether the IP datagram can be send to the destination directly or whether it must be send indirectly. The IP datagram can be send directly when the destination and the source are attached to the same physical network. The IP datagram must be send indirectly when the destination is attached to another physical network than the source. In the latter case the IP datagram must be send to one or more routers before it can be send to the destination. A router is a computer that interconnects two or more physical networks (see figure 3.3). A router can be the source or the destination itself as well.

![Figure 3.3](image)

*Figure 3.3:* The source and destination are connected to different physical networks. A router interconnects the two networks. The Internet layer handles the transport across an internet. The Network Interface layer handles transport across a physical network.

The internet layer uses a routing algorithm to determine to which computer the IP datagram should be send next. In addition, it passes this IP address of this computer to the network interface layer along with the IP datagram. Note that the destination address in the IP datagram is not changed. The algorithm compares the network portion of the destination IP address to the network portion of its own IP address. When the two network portions match the datagram can be send directly. In this case the IP address passed to the network interface layer is the same as the destination address in the IP datagram header. When the network portions are not the same the
IP datagram is send to a router. In this case the internet layer passes the IP address of the router along with the IP datagram to the next layer. Datagrams pass from router to router until they reach a router that can deliver the datagram directly. At the remote computer the network interface layer passes the IP datagram up to the internet layer. If the destination address of the IP datagram matches the computer's IP address the internet layer software passes the datagram to the transport layer for further processing. If the destination IP address does not match the computer's IP address, a router will provide a new IP address and pass the IP datagram along with this new address back to the network interface layer, all other computers discard the IP datagram. The new IP address can be the IP address of a new router or the destination address.

The main TCP/IP protocol related to the internet layer is the Internet Protocol (IP). The functionality described above is provided by the Internet Protocol software. The transport type provided by the protocol is an unreliable, connectionless packet delivery service.

3.5 Network Interface Layer

The network interface layer handles data transport across a physical network. The network interface layer software maps the IP address that is passed by the internet layer software to a physical network address when needed, encapsulates the IP datagram in a network specific frame and passed the frame to the correct computer at the physical network. On the remote computer the network interface layer extracts the IP datagram and passes it to the internet layer for further use. The physical network can be based on different network hardware technologies. The network interface layer contains device drivers for these hardware technologies (e.g. for Ethernet, backplane, serial line interconnections). Each hardware technology has its own intended use, addressing scheme and frame format.
Chapter 4

The Interface between Application and Transport Layer

The application layer of the TCP/IP internet communication system software resides outside the operating system, while the subsequent layers reside inside the operating system. An interface is needed that provides a mechanism to pass data from the application to the transport layer across this boundary. In the TCP/IP protocol suite such an interface is not standardised. The TCP/IP protocol suite was designed to operate in a multi-vendor environment. Interface details depend on the operating system’s mechanism to pass data from an application program to the operating system. However, the TCP/IP standards do suggest a conceptual interface based on the system call mechanism. Most operating systems use the system call mechanism to transfer control between an application program and the operating system procedures that supply services to the application program. When TCP/IP was implemented into the UNIX operating system, the designers created an interface based on this concept, that has become known as the socket interface. The socket interface has become widely accepted and is used in many other operating systems, including the ones that are described in chapter 2. The socket interface is described in section 4.1. More information about the socket interface can be found in chapter 20 of [Com 95] and chapter 4 and 5 of [Com 93] The operating system VxWorks adds a new feature to the socket interface not known to the other operating systems used in the data-acquisition system. This alternative interface is called the zbuf socket interface and is discussed in section 4.2. More information about this interface can be found in [VxW 95a] and [Bur 95].

4.1 The Socket Interface

Two examples of the communication process between a client and a server application, help to understand the socket interface (see figure 4.1). One example is based on a connection-oriented way of transport. The other is based on a connectionless way of transport. These communication processes comprise different stages: create sockets, connect sockets (connection-oriented transport only), send and receive data through sockets and close sockets. These stages are described in section 4.1.1 to 4.1.4. From the programmer’s point of view the socket interface is a set of routines, data types and structures and symbolic constants supplied by a specific vendor’s operating system. The set of routines contains system calls to transfer control between an application and the communication software inside the operating system and library routines that perform useful functions related to networking. Operating system specific routine names, arguments, data types and structures and symbolic constants are used in the discussion of the implementation of network communication in application programs in chapter 5. In this chapter names are used that describe the function of different aspects of the socket interface.
Application Program Communication Process

Connection-oriented communication (TCP)  

Client  
1. create socket  
2. socket options  
3. bind socket  

Server  
1. create socket  
2. socket options  
3. bind socket  

Connect Sockets  
(Section 2.1.2)  
4. listen for connection requests  
5. request connection  
6. accept connection request  

(7) connection established  

send and receive data  
(Section 2.1.3)  
8. send data  
9. receive data  

(10) receive data from  

(11) send data to  

Close Sockets  
(Section 2.1.4)  
12. close socket  

Connectionless communication (UDP)  

Client  
1. create socket  
2. socket options  
3. bind socket  

Server  
1. create socket  
2. socket options  
3. bind socket  

Connect Sockets  
(Section 2.1.2)  
4. listen for connection requests  
5. request connection  
6. accept connection request  

(7) connection established  

send and receive data  
(Section 2.1.3)  
8. send data to  
9. receive data from  

(11) send data to  

Close Sockets  
(Section 2.1.4)  
12. close socket  

Figure 4.1: This figure depicts the application program communication process between a client and a server application for both the connection-oriented and connectionless transport type. The different stages are discussed in the specified sections. The numbers above the frames correspond to the numbers behind the headers in these sections. The frames that are placed between brackets contain optional functionality.
4.1.1 Create Sockets

Create a Socket (1)

An application program that wants to perform network communication requests the operating system to create a socket. The application program uses this socket to communicate with the remote application program. Although the socket interface originates from the implementation of TCP/IP in the UNIX operating system, the socket interface was designed to provide an interface between application programs and network communication software inside the operating system in general. Therefore, the application program passes some information to specify the network communication software desired, along with this request for a socket. This information includes the protocol family to be used and the type of communication desired by the application program. The only protocol family supported by the socket interface implemented in the operating system used in the data-acquisition system is the TCP/IP protocol suite. Possible types of transport include connection-oriented and a connectionless transport. Since only the TCP/IP protocol suite is supported these services are provided by TCP and UDP respectively.

The operating system allocates a data structure to hold information related to the network communication that the application program performs using this socket. For example, the information specified along with the request to create the socket, as described above. The operating system returns a so-called socket descriptor to the application program. The application program uses this socket descriptor to address the socket for further use.

Specify Socket Options (2)

In addition to the protocol family and the way of transport that must be specified explicitly the application program can also specify additional information related to network communication. For example, the size of system network buffer space reserved for an application program's socket.

Bind the Local Communication Endpoint to the Socket (3)

An application program is uniquely identified by its communication endpoint. For TCP/IP the communication endpoint consists of a port number and an IP address. Server applications, i.e. applications that await to be contacted by client applications, must operate at a well-known local communication endpoint. Otherwise they can not be contacted by client applications. Therefore, server applications must specify this communication endpoint themselves. Client applications, i.e. applications that initiate contact with a server application, do not need a well-known local communication endpoint. The client communication endpoint is send to the server application. The client application may have the TCP/IP software inside the system specify a local communication endpoint. For both server and client applications the local communication endpoint is bound to the socket that is used to communicate. I.e. both port number and IP address are stored in the socket's internal data structure.

4.1.2 Connect Sockets

This section only applies to the communication process that uses a connection-oriented way of transport. To establish a connection between a client and a server application, both the client and the server application's socket must be associated with both the client and server application's communication endpoint. The previous section described how the local communication endpoint is bound to the socket. This section describes how the remote communication endpoint is bound to the socket. Note that the local communication endpoint associated to one application is the remote communication endpoint associated to the other application.
Listen for Connection Requests (4)

When the socket is created and a well-known local communication endpoint is bound to it, the server application program prepares to await incoming connection requests on this socket. The application program informs the operating system that the socket should not be associated to a specific remote communication endpoint. In fact, the remote communication endpoint must specify a wildcard, allowing the server to receive connection requests from an arbitrary client. In addition, it informs the operating system that the protocol software should place multiple simultaneous connection requests that arrive at the socket in a queue. Otherwise a new connection request that arrives before the server finishes processing the current connection request would be discarded by the system. However, when the queue is full the connection request will still be discarded. The length of the queue can be specified by the server program, but is limited to a certain length depending on the operating system. Connection requests that arrive before the server’s socket is enabled to receive connection requests, the requests are discarded as well. When a connection request is discarded the client application program that made the connection request is notified that establishing a connection failed. The fact that a connection request is placed in the queue does not mean that the connection request is accepted and that the connection is established. Accepting a connection request is the next step in establishing a connection between applications and is described below.

Request a Connection (5)

When a client application program wants to communicate with server application it requests a connection with that server application. Therefore, the client application must specify a remote communication endpoint (i.e. the well-known local communication endpoint of the server application). The transport layer software inside the system sends a connection request to the server application specified by the client. This connection request is placed in the queue associated to the server application’s socket.

In addition, the transport layer software handles further details of establishing the connection with the transport layer software on the server’s side when the connection request is extracted from the queue. When a connection is established the server’s communication endpoint is bound to the client’s socket (i.e. inserted in the client’s socket’s internal data structure).

Accept Connection Requests (6)

Once the server is prepared to await connection requests, the server transfers control to the system to accept connection requests that are placed in the queue associated to the server application’s socket. When no connection request is present in the queue the server application program waits for the next connection request. When a connection request arrives in the queue or when one is already present the connection request is extracted from the queue. In addition, the transport layer software inside the system handles the details of establishing a connection with the transport layer software on the clients’ side. When a connection is established the system creates a new socket with the same local communication endpoint bound to it and in addition the communication endpoint of the client. The system returns the socket descriptor of the newly created socket as well as the client’s communication endpoint back to the server application. The server uses this new socket to communicate with the client application. The original socket still has a wildcard foreign destination, and still remains open to receive connection requests from other clients on its queue while the application program is communicating with the client. After the communication is finished the server application returns control back to the operating system to accept or wait for the next connection request.
As said in chapter 3 an application program is uniquely identified by its communication endpoint. However, this does not mean that an application can not use more than one socket and that these sockets are associated to the same local communication endpoint. Data is passed from the application program to the transport layer software and the other way around through the same port. A distinction between the different sockets is made based on both the local and the remote communication endpoints associated to the sockets.

**Connection Established (7)**

Now that both the socket used by the client application and the newly created socket of the server application have the local and the remote communication endpoint associated the connection is established.

### 4.1.3 Send and Receive Data through Sockets

#### Through Connected Sockets ((8) and (9))

Once client and server applications are connected they start exchanging actual data. Sending data by one application must correspond to receiving data by the other application.

When an application program sends data to a remote application program it actually sends the data to the transport layer software through its socket. The application program passes control to the operating system, specifying the address of the application buffer containing the data to be sent. The system copies the data into a socket buffer (i.e. system memory space associated to a socket) and allows the application to continue execution while the TCP/IP software inside the system transmits the data across the network. If the socket buffer is full or becomes full, the application program blocks temporarily until the data is sent to the remote application and the buffer can be filled with new data.

In addition, when an application program receives data from a remote application it actually receives the data from the underlying transport layer through its socket. When an application program expects data it passes control to the system specifying the address of an application buffer in which the data should be placed. When data arrives at in a socket buffer, the system copies this data to the applications buffer. If no data is present the application program blocks until it is.

#### Through Unconnected Sockets ((10) and (11))

Exchanging data using unconnected sockets means that the sockets are not associated to a remote communication endpoint. Application programs using a connectionless way of transport start exchanging data directly after they created their socket. Again sending data by one application must correspond to receiving data by the other application. The server application starts by waiting for data to arrive. The client application starts sending data directly. The mechanism of sending and receiving data by copying data from the application buffers to the sockets internal buffers and having the application layer transport data further is the same as with connected sockets. Because the application programs are not connected the client must specify the remote communication endpoint the data must be send to, besides the address of the application buffer containing the data to be send. In addition, the transport layer software on the remote computer passes the local communication endpoint of the application the data is received from to the server application. The server may need this communication endpoint as a remote communication endpoint to send data back to the client. Because, when using a connectionless way of transport these remote communication endpoints are not bound to the local sockets, the remote communication endpoint must be specified each time data is send. When communication between the client and the server is finished the server waits for new data to arrive.
4.1.4 Close Sockets

Close a Socket (12)

When an application program is finished using a socket it closes the socket. The application passes control to the operating system, specifying the socket to be closed. The operating system closes the socket and frees the socket's internal data structures and system memory space. When a client and a server that use connected sockets finish communication they both close their socket related to this communication. With respect to the server this is the socket that is created when the server accepts a connection request. The socket that the server uses to receive connection request remains open as long as the server keeps providing its service (see figure 4.1). When a client and a server that use unconnected sockets finish communication only the client closes it socket. The servers socket remains open.

4.2 The Zbuf Socket Interface

When using the socket interface to transmit data, the data is copied between application and operating system buffers. Especially when large amounts of data are transmitted this is a very time-consuming operation and slows down the overall performance of the network throughput. Therefore, some operating systems provide a zero-copy interface. A zero-copy interface allows the application program to send data to and receive data from the network communication software inside the system without copying the data.

VxWorks can provide such a zero-copy facility because it has two features not known to the other systems. One, VxWorks provides a facility which permits separate software modules to share data buffers and manipulated these buffers, leaving them unchanged for other modules, by copying pointers to the data instead of the data itself. This facility is based on a data abstraction called a zbuf. And two, VxWorks uses a flat memory scheme, which allows network and application buffers to coexist in the same memory space. In addition, this allows the operating system and application programs to share the same data buffers. This makes it possible to use the zbuf facility in relation to software modules in and outside the operating system.

The VxWorks operating system has implemented the zero-copy interface in what is called the zbuf socket interface. The zbuf socket interface allows applications to send and receive data through UNIX based sockets, without copying data between application buffers and operating system buffers. The communication process is the same as described in section 4.1. However, instead of specifying the address of the application buffers the system should copy the data from or to, the application program sends or receives a zbuf. From the programmer's point of view the zbuf socket interface adds a new set of routines for sending and receiving data to existing socket interface routines.
Chapter 5

Network Communication between Application Layer Programs

This chapter discussed different aspects of network communication between application programs in the context of the data-acquisition system that is discussed in chapter 1 (hardware) and 2 (software). In this data-acquisition system, network communication between application programs takes place between a front-end processor running the VxWorks operating system and a back-end processor running the OpenVMS operating system. Section 5.1 describes the implementation of the different steps of the two communication processes discussed in chapter 4 in applications that run on VxWorks and OpenVMS. In these applications connection-oriented and connectionless communication are based on the transport layer protocols TCP and UDP respectively as provided by the TCP/IP communication system software. Section 5.2 describes the reliability of network communication between application programs. Section 5.3 discusses network throughput. Both reliability and throughput are discussed from the application's point of view. These aspects are studied using client and server application programs in which the control flows from top to bottom. These application programs are used because they help to study aspects related to network communication without the aspects of a graphical user interface that might interfere. The applications that run on the back-end processor of the data-acquisition system run in a graphical environment. In an application with a graphical user interface the control flow is much more complex. This is discussed in section 5.4.

5.1 Implementation of Network Communication in Application Programs

5.1.1 Introduction

Both VxWorks and OpenVMS use a UNIX-like socket interface (see chapter 4) to facilitate communication between the applications and the transport layer module of the TCP/IP software inside the operating system. From the application's point of view, the socket interface is a set of routines that is used to implement network communication into the application program.

VxWorks provides a UNIX BSD 4.3 compatible socket library to implement network communication into application programs. Furthermore, it provides an additional library that contains routines that are used to send and receive data through zbuf sockets (see section 4.2).

OpenVMS provides two ways to implement network communication into application programs. First, by means of a set of DEC C socket routines. The DEC C socket routines are fully compatible with the routines of the VxWorks socket library. This set of routines hides the details of the OpenVMS implementation of the socket interface. This increases the portability of the source code of an applications written for OpenVMS to other operating systems. Second, by means of OpenVMS system service routines. The OpenVMS operating system kernel has many services that are made available to application and system programs by means of system service routines. One of these services comprises network communication. Using system service routines to implement
network communication into applications corresponds directly to the implementation of the socket interface in OpenVMS. This can increases the performance of an application.

Both the UNIX-like routines provided by VxWorks and OpenVMS and the system service routines provided by OpenVMS are described in this section. The UNIX-like routines are described because these routines will be used in data-acquisition applications running on the front-end processor, that is in applications running on VxWorks. The system service routines are described because they are used in applications running on the back-end processor, i.e. running on OpenVMS. The reason that system service routines are used instead of the VxWorks compatible DEC C routines is that the system service routines provide a mechanism that can be used to implement network communication in applications with a graphical user interface. This mechanism is provided by the Queue I/O Request system service. This section describes the use of system service routines in relation to the implementation of network communication. Section 5.4 describes the use of the system service routines (more specific the Queue I/O Request system service routine) in relation to network communication in an application with a graphical user interface. The Queue I/O Request system service plays a general role in the implementation of network communication whether or not the application is running a graphical user interface. Therefore, the Queue I/O Request system service is described in more detail first.

The Queue I/O Request system service provides a way to queue an I/O request (i.e. an I/O operation) on a channel associated to a pseudo-device. The OpenVMS operating system uses a pseudo-device as an intermediary to perform input/output operations on an actual device. A channel is a mechanism that is used to communicate with a pseudo-device. More about how to create a pseudo-device and to assign a channel to this pseudo-device in relation to network communication is described in section 5.1.2. The Queue I/O Request system service has two variations:

The Queue I/O Request system service. The Queue I/O Request system service completes asynchronously. The Queue I/O Request system service is available to the application through the SYSSQIO() routine. The SYSSQIO() routine returns to the application immediately after queuing the I/O operation request, without waiting for the I/O operation to complete. As will be seen in section 5.4 the asynchronous completion of the SYSSQIO() routine and the I/O operation queued is essential in the implementation of network communication in an application with a graphical user interface.

The Queue I/O Request and Wait system service. The Queue I/O Request and Wait system service completes synchronously. The Queue I/O Request and Wait system service is available to the application through the SYSSQIOW() routine. The SYSSQIOW() routine returns to the caller after the I/O operation completes.

The use of the SYSSQIO() and the SYSSQIOW() routine in relation to network communication is the same. In addition, the parameters passed with the SYSSQIO() and the SYSSQIOW() system service routine are the same (see figure 5.1).

\[
\text{SYSSQIOW}(\text{efn}, \text{channel}, \text{IO\_function}, \text{iosb}, \text{astadr}, \text{astprm}, p1, p2, p3, p4, p5, p6);
\]

**Figure 5.1:** The syntax of the SYSSQIO() and the SYSSQIOW() system service routine. The parameters passed with the routines are the same. See the text for a description of the parameters.
Two types of parameters are passed with the Queue I/O Request system service routines: function-independent parameters and function-dependent parameters. The function is the I/O operation that is queued. This function itself is one of the function-independent parameters. The first six parameters passed with the routine are the function-independent parameters. The last six parameters are the function-dependent parameters.

1) efn: The event flag number of an event flag that the system service routine sets when the I/O operation completes. An event flag number is specified in the applications described in this chapter. However, the event flag is not used any further.

2) channel: The number of the I/O channel that is assigned to the pseudo-device the I/O operation must be performed on.

3) IO_function: Specifies the pseudo-device function (and modifier) that defines the operation to be performed on the pseudo-device. A function modifier affects the operation of a specified function. A vertical bar is used to separate the function and the modifier. This parameter determines how the function-dependent parameters must be used.

4) iosb: The address of the I/O status block which receives the final status of the complication of the I/O operation.

5) astadr: The address of an asynchronous system trap (AST) routine to be executed when the I/O operation is completed. A feature of an AST routine is that it is executed as soon as possible after a certain event takes place. In this case this event is the completion of the I/O operation. This parameter is not used in relation to the implementation of network communication in this section. However, it plays an important role in the implementation of network communication in a application with a graphical user interface.

6) astprm: A value to be passed to the AST routine (optional).

7) to 12) p1 to p6: Function dependent parameters. The I/O function that is specified by the IO_function parameter determines which parameters must be passed and the type of these parameters. The parameter passed can be a value, a symbolic constant, a reference to the address of a value or a descriptor address. A descriptor contains information about the actual parameter(s) passed with the system service routine.

More information about the Queue I/O Request system service can be found in chapter 4 of Dig 95.

The following sections describe the implementation of the different steps of the communication processes discussed in chapter 4 using both the UNIX-like routines and the system service routines. Section 5.1.2 describes how to create sockets for both client and server applications that use a connection-oriented transport type (TCP) or a connectionless transport type (UDP). Section 5.1.3 describes how to connect sockets in case of TCP as transport layer protocol. Section 5.1.4 describes how to send and receive data through TCP- and UDP-based sockets. In addition it describes how to send and receive data through TCP-based zbuf sockets (VxWorks only). Section 5.1.5 describes how to close sockets. Finally, section 5.1.6 describes the implementation of a mechanism that makes it possible for one server application to send data to and receive data from more clients at the same time. Such a server is called a concurrent server. A concurrent server application creates a sub-process to handle further communication with each client application. This mechanism is available on both VxWorks and OpenVMS. This section only describes the implementation in an application running on VxWorks. In this context create a sub-process is called spawn a task.

In the description of the implementation of the different steps of the communication process, operating system specific routine names are used in stead of descriptive terms. Table 5.1 contains both the UNIX-like routine names and the system service routines names that implement the different steps of the communication process in an application program. The IO_function parameter and its additional function-dependent parameters determine the actual functionality of the Queue I/O Request system service routine in relation to the communication process. Therefore, this routine is denoted as SYSSQIOW(IO_function(additional parameters)). In this section only the SYSSQIOW() routine is used to queue I/O operations.
Appendix A gives source codes examples of the implementation of the different steps of network communication by means of both UNIX-like and system service routines. More information about these routines can be found in [Dig 95] and [VxW 95b].

Table 5.1: The routines that implement the different steps of the communication process in an application program. The first column contains the different steps of the communication process. The second and the third column contain two alternative sets of routines that implement these steps. The second column contains UNIX-like routines. These routines are available from both VxWorks and OpenVMS DEC C libraries. The third column contains OpenVMS system service routines. Because the IO_function parameter and its additional function-dependent parameters determine the actual functionality of the Queue I/O Request system service routine in relation to the communication process this routine is denoted as SYSSQIOW(IO_function(additional parameters)).

<table>
<thead>
<tr>
<th>Description</th>
<th>VxWorks and OpenVMS DEC C socket routine</th>
<th>OpenVMS system service routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 5.1.2 Create Sockets</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create a socket.</td>
<td>socket()</td>
<td>SYSSASSIGN() and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SYSSQIOW(IO$SETMODE(p1))</td>
</tr>
<tr>
<td>Specify socket options.</td>
<td>setsockopt()</td>
<td>SYSSQIOW(IO$SETMODE(p5))</td>
</tr>
<tr>
<td>Bind local communication endpoint to socket.</td>
<td>bind()</td>
<td>SYSSQIOW(IO$SETMODE(p3))</td>
</tr>
<tr>
<td><strong>Section 5.1.3 Connect Sockets</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Listen for connection requests.</td>
<td>listen()</td>
<td>SYSSQIOW(IO$SETMODE(p4))</td>
</tr>
<tr>
<td>Request a connection.</td>
<td>connect()</td>
<td>SYSSQIOW(IO$ACCESS(p3))</td>
</tr>
<tr>
<td>Accept connection requests.</td>
<td>accept()</td>
<td>SYSSASSIGN() and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SYSSQIOW(IO$ACCESS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOM_ACCEPT(p3, p4)</td>
</tr>
<tr>
<td><strong>Section 5.1.4 Send and Receive Data through Sockets</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send data through a TCP-based socket.</td>
<td>send()</td>
<td>SYSSQIOW(IO$WRITEVBLK(p1, p2))</td>
</tr>
<tr>
<td></td>
<td>receive()</td>
<td>SYSSQIOW(IO$READVBLK(p1, p2))</td>
</tr>
<tr>
<td>Send data through a UDP-based socket.</td>
<td>sendto()</td>
<td>SYSSQIOW(IO$WRITEVBLK(p1, p2, p3))</td>
</tr>
<tr>
<td>Received data through a UDP-based socket.</td>
<td>recvfrom()</td>
<td>SYSSQIOW(IO$READVBLK(p1, p2, p3))</td>
</tr>
<tr>
<td>Send data through a TCP-based zbuf socket.</td>
<td>*zbufSockBufSend(), *zbufSockSend()</td>
<td></td>
</tr>
<tr>
<td>Received data through a TCP-based zbuf socket.</td>
<td>*zbufSockRecv()</td>
<td></td>
</tr>
<tr>
<td><strong>Section 5.1.5 Close Sockets</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shutdown a connection.</td>
<td>shutdown()</td>
<td>SYSSQIOW(IO$DEACCESS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO$M_SHUTDOWN(p4)</td>
</tr>
<tr>
<td>Close a socket.</td>
<td>close()</td>
<td>SYSSQIOW(IO$DEACCESS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SYSSDASSIGN()</td>
</tr>
<tr>
<td><strong>Section 5.1.6 Concurrency in a Server Application Program</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spawn a task.</td>
<td>*taskSpawn()</td>
<td></td>
</tr>
</tbody>
</table>

*) Available on VxWorks only.
5.1.2 Create Sockets

socket() / SYS$ASSIGN() and SYS$QIOW(IO$SETMODE(p1))

To create a UNIX-like socket, the VxWorks socket library and the OpenVMS DEC C socket routines provide the socket() routine. The implementation of the socket interface in the OpenVMS operating system differs a little from the UNIX BSD 4.3 socket interface. The OpenVMS DEC C socket routines hide these differences. However, the OpenVMS system services do not. One of the differences is the use of a device-socket instead of a socket. A device-socket is the OpenVMS implementation of a socket and is described in more detail below. To create a device-socket both the SYS$ASSIGN() and the SYS$QIOW(IO$SETMODE(p1)) system service routines are needed.

The socket() routine is used to create a UNIX-like socket. The three parameters passed with this routine are (see appendix A.1.1):
1) A symbolic constant that specifies the protocol family to be used. AF_INET for TCP/IP.
2) A symbolic constant that specifies type of transport. SOCK_STREAM for connection-oriented and SOCK_DGRAM for connectionless transport.
3) A parameter that specifies the transport protocol to be used. Since TCP/IP provides only one protocol per transport type this parameter is 0. The protocol used in case of connection-oriented transport is TCP. The protocol used in the other case is UDP.

The socket() routine returns a socket descriptor. This socket descriptor is passed with all subsequent routines that perform actions on the socket. In the remainder of this text, when discussing parameters passed with other UNIX-like socket routines, the socket descriptor will not be explicitly mentioned anymore.

The SYS$ASSIGN() and the SYS$QIOW(IO$SETMODE(p1)) system service routines are used to create the device-socket.

First, the SYS$ASSIGN() system service routine is used to create a pseudo-device. The OpenVMS operating system uses pseudo-devices as an intermediary to perform input/output operations on actual devices. To create an internet pseudo-device, the pseudo-device needed for network communication, the character string "UCX$DEVICE" is passed with the SYS$ASSIGN() routine (see appendix A.1.2). An internet pseudo-device provides an interface to the internet protocols. The system service provides the application with an I/O channel so I/O operations can be performed on the internet pseudo-device. The system returns a channel number, corresponding to the assigned I/O channel. This channel number is used in all subsequent I/O operations on the internet pseudo-device. For the internet pseudo-device the channel number is equivalent to the socket descriptor returned by the socket() routine. This channel number is used by all subsequent system service routines.

Second, the SYS$QIOW() routine is used to queue the IO$SETMODE function with the p1 parameter specified. This parameter is an array containing two symbolic constants.
1) A symbolic constant that specifies both the protocol family and transport layer protocol. UCXSC_TCP for TCP/IP and UCXSC_UDP for UDP/IP.
2) A symbolic constant that specifies the transport type. UCXSC_STREAM for connection oriented and UCXSC_DGRAM for connectionless transport.

These characteristics are the same as those specified with the C socket routine socket() to create a socket. Specifying these characteristics creates a socket. Together the OpenVMS internet pseudo-device and the socket are known as a device-socket. More information about the implementation of the socket interface in the OpenVMS operating system can be found in [DEC 94].
setsockopt() / SYS$QIOW(IO$_SETMODE(p5))

This routine is used to set options related to network communication. The setsockopt() is used in relation to the UNIX-like socket. the SYS$QIOW(IO$_SETMODE(p5)) is used in relation to the OpenVMS device-socket. In both cases options are indicated by two symbolic constants:

1) A constant that indicates the network communication level the option is related to. The applications discussed in this chapter have only socket level options set. The symbolic constant for this level is SOL_SOCKET in case of the setsockopt() routine and UCX$C_SOCKOPT in case of the system service routine.

2) A constant that indicates the actual option. The actual options set are:
   a) SO_SNDBUF (UCX$C_SNDBUF) and SO_RCVBUF (UCX$C_RCVBUF). These two options set the maximum size of the socket level send and receive buffer respectively. On OpenVMS these options require that a user has a system UIU, SYSPRV, BYPASS, or OPER privilege. The options are used to study the effect of the socket buffer size on network throughput, discussed in section 5.3.
   b) SO_REUSEADDR (UCX$C_REUSEADDR). The reason to set this option is discussed below.

The setsockopt() routine can be used to set only one option each time the routine is called (see appendix A.2). The SYS$QIOW(IO$_SETMODE(p5)) routine can be used to set a number of options with one call to the routine. Therefore, all information related to these options is stored in a list. The p5 parameter passed with the IO$SETMODE function specifies the address of this list (see appendix A.2.2). Information about current values of options can be obtained by using the routines getsockopt() and SYS$QIOW(IO$SENSEMODE(p6)).

bind() / SYS$QIOW(IO$_SETMODE(p3))

This routine is used to bind the local communication endpoint to the socket. For TCP/IP the communication endpoint consists of a port number and a IP address. This communication endpoint is stored in a sockaddr_in structure and passed as a parameter with the routine. The sockaddr_in structure is the same for both the VxWorks and the OpenVMS operating system as well as for the UNIX-like and the system service routines. Appendix A.3 shows how the members of the structure are initialised and how it is passed to the UNIX-like and the system service routine.

When the local port number that is specified is in use or not released by another application the routine returns an error, indicating that the port is already in use. When an application finishes communication correctly (see section 5.1.5) it releases the local port. In addition, on OpenVMS the local port is released as well when the application terminates communication incorrectly (e.g. when the application crashes). However, when using a TCP-based socket, using a local port number that has been previously used by another application, returns the same error, unless the REUSEADDR option is set.

5.1.3 Connect Sockets

This section applies only to TCP-based sockets, because only network communication through TCP-based sockets is based on a connection-oriented way of transport.

listen() / SYS$QIOW(IO$_SETMODE (p4))

Server applications use this routine to be able to receive incoming connection requests. The parameter passed with this routine is the length of the queue in which multiple simultaneous connection request are stored. However, in practice the number of simultaneous connection requests that can be stored differs from the queue length specified by the routine. The reason for this difference is unknown. Table 5.2 shows the differences. The differences are the same for the VxWorks and the OpenVMS operating system.
Network Communication between Application Layer Programs

Table 5.2: The length of the queue in which simultaneous connection request are stored as specified by the listen() or the SYSSQIOW(IO$_SETMODE(p4)) routine compared to the number of simultaneous connection requests that are possible in practice.

<table>
<thead>
<tr>
<th>Maximum queue length specified by the application.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of simultaneous connection requests possible in practice</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**connect() / SYSSQIOW(IO$_ACCESS (p3))**

Client applications use this routine to request a connection to a server application. The server application’s communication endpoint is passed as parameter with the routine (see appendix A.5). Again the communication endpoint is stored in a sockaddr_in structure. When a server is not yet able to receive connection requests, the routine returns an error that indicates that establishing a connection failed. When the server application’s queue is full the routine blocks until the connection request can be queued. However, on both operating systems a connection request times out in 75 seconds. The routine returns an error indicating that the connection request timed out. This time-out interval can be changed. When the connection request is queued, the routine returns successful and the client application continues execution. Note that the actual connection between the client and the server application is not yet established. For the actual connection to be established the connection request must be accepted by the server application. In addition, the TCP/IP software inside the operating system handles the establishing of the actual connection with the TCP/IP software inside the system on the server’s side. The client application does not know whether the connection is (yet) established or establishing the connection failed until it calls the next network communication related routine (usually a send or receive routine).

**accept() / SYSSASSIGN() and SYSSQIOW(IO$_ACCESS(IO$M_ACCEPT (p3, p4))**

When a server application is able to receive connection requests it enters a loop in which it accepts a connection request and handles the actual communication with the client application. This section describes how the server application accepts a connection request. Section 5.1.4 describes the actual communication. As will be seen, accepting a connection request comprises the creation of a new socket. Therefore, the implementation is a little different for the UNIX-like socket interface and the OpenVMS system service interface.

The accept() routine extracts a connection request from the queue and creates a socket that is used in further communication with the client application. The routine blocks until a connection request is present in the queue. Again, when a connection request is extracted from the queue, the TCP/IP software inside the operating system handles the details of establishing the actual connection with the TCP/IP software inside the system on the client side. Parameters are passed with the routine to obtain the communication endpoint of the client application, which makes it available to the application itself when the routine returns (see appendix A.6.1). The accept() routine returns the socket descriptor of the newly created socket.

An application that uses system services uses the SYSSASSIGN() and SYSSQIOW(IO$ACCESS(IO$M_ACCEPT(p3, p4)) routines to accept a connection request. First, the server application must create an internet pseudo device itself by means of the SYSSASSIGN() routine. The SYSSASSIGN() routine is described at the beginning of section 5.1.2. Second, the SYSSQIOW(IO$ACCESS(IO$M_ACCEPT(p3, p4)) routine extracts a connection request from the queue and creates a device-socket that is used in further communication with the client application. To create this device-socket the channel number provided by the SYSSASSIGN() routine is passed by reference with the routine. This channel number is used in all further actions on the newly created device-socket as well. Furthermore, the routine provides the same functionality as the accept() routine.
5.1.4 Send and Receive Data through Sockets

Applications use the routines described in this section to exchange data. There is a difference between the UNIX BSD 4.3-based socket interface (see section 4.1) and the zbuf socket interface (see section 4.2) in the way data is exchanged.

The first four sections describe the routines that are used to transmit data in case of the UNIX BSD 4.3-based socket interface. Both VxWorks and OpenVMS socket interface are based on the UNIX BSD 4.3 socket interface. These routines copy data between the application buffer and the socket buffer inside the operating system. The first two sections describe the routines that are used in relation to TCP-based sockets, the latter are used in relation to UDP-based sockets.

The last three sections describe the routines that are used to transmit data in case of the zbuf socket interface. The zbuf socket interface is a zero-copy interface based on a data abstraction called a zbuf. The zbuf socket interface is only available on VxWorks. From the programmer’s point of view the zbuf socket interface adds an additional set of routines to send and receive data to the existing socket interface routines. These routines do not copy the data from the application to the socket buffer. Instead they transmit zbufs between the application and the TCP/IP software inside the system. The three routines described below are used in relation to TCP-based sockets. Corresponding routines related to UDP-based sockets are available as well.

send() / SYS$QIOW(IO$_WRITEVBLK(p1, p2))

This routine is used to send data through TCP-based sockets. The routine copies the data from the application buffer to the socket buffer. Parameters passed with the routine are the address of the application buffer containing the data to be send and the number of bytes to be send (see appendix A.7). The routine blocks when the socket buffer is full until buffer space becomes available. Buffer space becomes available when the data is transmitted to the remote application. In addition, because TCP-based sockets use connection-oriented transport, this depends on whether space is available in the remote application’s socket buffer. The routine returns when all data is copied to the socket buffer. The number of bytes that can be send in one call to the routine is not limited.

recv() / SYS$QIOW(IO$_READVBLK(p1, p2))

This routine is used to receive data through TCP-based sockets. Parameters passed with this routine are the address of the application buffer in which to receive the data and the number of bytes to receive. The routine copies data from the socket buffer to the application buffer. The number of bytes to receive passed to the routine is not limited, like with the send routine. However the routine does not block until the number of bytes to receive is actually received. The routine blocks when the socket buffer is empty. When data is or becomes available the routine returns after it has copied the specified number of bytes or less, when less data is available, from the socket buffer to the application buffer. The routine returns the actual number of bytes received. The total TCP/IP protocol software determines how the data send from the application buffer one application becomes available at the socket buffer of another application. The data may be split into smaller parts and recombined again, depending on the different buffer or packet sizes and protocols used at the different layers of the TCP/IP software. The routine recvAllSend() is written to make sure that all the data send by the other application is received. The recvAllSend() routine repeatedly calls the receive routine until the number of bytes to be send, as specified in the send routine, is actually received. The principle is the same for both UNIX-like and system service applications. Implementation details depend on the little differences in the recv() and SYS$QIOW(IO$_READVBLK(p1, p2)) routines (see appendix A.8). Parameters passed with this routine are the address of the application buffer to receive the data in, the number of bytes to receive and a pointer to an integer that contains the number of bytes received when the routine returns.
**Network Communication between Application Layer Programs**

**sendto() / SYS$QIOW(IO$.WRITEVBLK(p1, p2, p3))**

This routine is used to send data through UDP-based sockets. The routine copies data from an application buffer to the socket buffer. Parameters passed with this routine are again the address of the application buffer containing the data to be send and the number of bytes to be send. In addition, a parameter is passed with the routine that specifies the communication endpoint the data should be send to. The communication endpoint is again stored in a sockaddr_in structure (see appendix A.9).

The routine blocks when the socket buffer is full until space becomes available in the buffer. Buffer space becomes available when the data is transmitted to the remote application. However, because UDP-based sockets use connectionless transport this does not depend on whether buffer space is available in the remote application's socket buffer. The sending application transmits the data whether or not the remote application is ready to receive the data. The routine returns when all data is copied to the socket buffer.

The value of the parameter that specifies the number of bytes to be send is limited to the size of the socket buffer. In addition, the VxWorks operating system imposes an absolute maximum of 32739 bytes on the value of this parameters. The cause of this maximum is unknown. The routine sendtoMult() is written to be able to send any amount of data with one call to this routine. The sendtoMult() routine repeatedly calls the send routine until the number of bytes to be send, passed to sendtoMult(), is copied from the application buffer to the socket buffer. The routine uses the limitations on the number of bytes that can be send in one call to the send routine to determine the number of bytes to be send that is passed in each call to the send routine. Again the general idea of the sendtoMult() routine is the same for both UNIX-like and system service routines and implementation details depend on the specific send routine used. The parameters of the sendtoMult() routine are a little bit different than those of the send routines. The parameter that specifies the number of bytes to be send is not limited anymore. An additional parameter is a pointer to an integer that contains the number of bytes send when the routine returns.

**recvfrom() / SYS$QIOW(IO$._READVBLK.(p1, p2, p3))**

Applications use this routine to receive data through UDP-based sockets. The routines copy data from the socket buffer to the application buffer. Parameters passed with this routine are the same parameters as passed with the routine to receive data through TCP-based sockets. An additional parameter contains the communication endpoint the data is received from when the routine returns (see appendix A.9). The routine behaves exactly the same as the receive routine discussed above with respect to blocking and returning. When using UDP the data transmitted with one call to the send routine becomes available for the remote application as a whole (although the data may be split during transmission). Therefore, one call to the receive routine is always enough to receive all data that is send with one call to the send routine, provided that the correct number of bytes to receive is passed.

The sending application uses the routine sendtoMult() to send any amount of data using repeated calls to the send routine. The recvfromAllSend() routine enables the receiving application to receive the same amount of data that is send with the sendtoMult() routine. This routine operates the same as the recvAllSend routine(). An additional parameter contains the remote application's communication endpoint address when the routine returns. The number of times the receive routine is called is equal to the number of times the send routine is called.

**zbufSockBufSend()**

This routine is used to send data through TCP-based zbuf sockets. Instead of copying data from an application buffer to the socket buffer, this routine creates a zbuf from the application buffer and transmits it to the socket. Parameters passed with this routine are: the address of the application buffer containing the data to be send, the number of bytes to be send and a user-provided free routine callback function (see appendix A.11.1). This free
routine is a callback function that is executed when the application buffer is no longer in use by the TCP/IP software inside the operating system (i.e., when the application buffer is free). The functionality provided by the free routine is determined by the user (i.e., the application) itself. Applications can use this callback function to receive notification that the application buffer is freed by the system. In addition, the application buffer can be filled with new data or the memory space can be freed.

The value of the parameter that specifies the number of bytes to be send is limited to the half the socket buffer size or 32767 bytes when the half the socket buffer size is larger than 32767 bytes. 32767 bytes is the maximum size of a zbuf segment. Apparently, when creating a zbuf from the application buffer, the zbufSockBufSend() routine places all data into one zbuf segment. Since the maximum size of a zbuf segment is 32767 bytes, the number of bytes send with one call to this routine is limited to this value. The zbufSockBufSendMult() routine is written to be able to send any amount of data in one call to zbufSockBufSendMult(). The routine repeatedly calls the zbufSockBufSend() routine until the number of bytes to be send, passed as parameter with the zbufSockBufSendMult() routine, is send. The routine determines the number of bytes to be send in each individual call to zbufSockBufSend() based on the limitations imposed on the number of bytes that can be send with this routine.

zbufSockSend()

This routine is also used to send data through a TCP-based zbuf socket. Instead of creating a zbuf from an application buffer and transmitting it to the socket, this routine transmits a previously created zbuf to the socket. The application must create a zbuf from the application buffer it wants to send itself (see appendix A.11.2). The routine is passed the zbuf ID of the zbuf to transmit and the length of the zbuf. The routine transfers ownership of the zbuf from the application to the TCP/IP software inside the system. The routine deletes the zbuf ID so it can not be used anymore when the routine returns. The maximum length of the zbuf to be transmitted with zbufSockSend() is limited to the half the size of the socket buffer. The zbufSockSendMult() routine is written to send any amount of data. The parameters passed with this routine are the same as those of zbufSockBufSendMult(). The routine repeatedly creates a zbuf and calls zbufSockSend() to transmit it to the socket until the specified amount of data is send. The number of bytes inserted in each zbuf is determined by the size of the socket buffer. In addition care is taken that the zbuf segments of the zbuf are smaller than 32767 bytes. The same sort of free routine callback function as described above can be passed to routines that are used to place an application buffer into a zbuf.

zbufSockRecv()

This routine is used to receive data through a TCP-based zbuf socket. This routine creates a zbuf from the data that arrives at a socket and transmits this zbuf to the application. The routine returns the zbuf ID of the zbuf created (see appendix A.12). Once the application finishes using the zbuf, the zbufDelete() routine should be called. This informs the TCP/IP software inside the system that the system buffer space can be filled with new data again. An argument to this routine is a pointer to an integer containing the number of bytes requested when the routine is called and the number of bytes received when the call returns. The routine zbufSockRecvAllSend() routine is written for the same reason as the recvAllSend() discussed above. The data received by successive zbufSockRecv() routines are combined in one zbuf instead of a normal application buffer.
5.1.5 Close Sockets

**shutdown() / SYS$QIOW(IO$_DEACCESS|IO$M_SHUTDOWN(p4))**

This routine disallows further sending and/or receiving of data through a socket, specified by the parameter passed to the routine. Pending data that is to be send and/or received is discarded.

**close() / SYS$QIOW(IO$_DEACCESS) and SYS$DASSIGN()**

The close() routine is used to close a UNIX-like socket. It deletes the socket descriptor and deallocates the internal data structures containing the information related to the network communication performed on this socket. The port associated to the socket is cleared. The SYS$QIOW(IO$_DEACCESS) routine is used to close the socket part of the OpenVMS device-socket. In addition, the SYSSDASSIGN() system service routine is used to close the internet pseudo device and releases the I/O channel associated with it.

5.1.6 Concurrency in a Server Application Program

In both communication processes described in chapter 4 the server application handles communication with one client application before it communicates with the next client application. The course of client applications contacting a certain server application in time depends on the purpose of the client-server communication as well. When the communication between a client and a server application takes a long time, subsequent clients have to wait long before they can communicate with this server application. A possible solution would be to have the server application handle communication with different client applications concurrently. When a concurrent server is contacted by a client application it creates a sub process to handle further communication. In the meantime the main process awaits to be contacted by other client applications. Note that this does not increase the average number of client applications that can be helped per unit of time. On the contrary, because of the overhead of creating the sub process the communication between client and server even takes longer. More information about concurrency in both server and client applications can be found in [Com 93]. In the application programs discussed in this chapter concurrency is only implemented in a TCP-based server application running on VxWorks. The routine that provides this functionality is discussed at the bottom of this section. More information about routines providing the same functionality on the OpenVMS operating system can be found in [DEC 93].

**taskSpawn()**

This routine is used to create and activate (i.e. spawn) a new task. Parameters passed to this routine are a task name, the priority and the stack size of the new task (see appendix A.14). The priority determines the CPU time assigned to the task and is specified by means of a value between 1 and 100 (the priority increases with increasing value). The stack, which size is passed as parameter, is the only resource allocated to the new task. The task spawned by the server application to handle further communication with a client application has priority 100 and a stack size of 32768 bytes. Furthermore the routine is passed the address of the main routine of the spawned task with ten required parameters. When the number of arguments to the main routine is less than ten, the additional parameters are be zero. The main routine of the task spawned by the server application performs the same functionality as the other applications studied in this chapter (i.e. sending and receiving data). Arguments of this routine are the socket descriptor of the new socket created by the accept routine and the IP address and port number of the client application. The socket descriptor is used in subsequent socket routines that handle further network communication.
5.2 Reliability of Network Communication

This section discusses the reliability of network communication from the application’s point of view. A number of reasons are described why network communication between application programs can go wrong.

Errors in the design and/or implementation of the application programs

Network communication between two applications is based on sending and receiving data by means of the routines described in section 5.1.4.

When the number of bytes to be received by the receive routine is larger than the number of bytes to be send by the send routine, the receive routine will block waiting for the remaining data to be received. This data will never be send by the corresponding send routine. In addition, when no other data will be send at all the receive routine will block forever. When data is send by a following send routine this data is (partially) received by the waiting receive routine and not by the receive routine that corresponds to this send routine. In this case communication between the two applications becomes corrupted. This is the same for both TCP and UDP as transport layer protocol.

What happens in the reverse situation, i.e. the number of bytes to be send by the send routine is larger than the number of bytes to be received by the receive routine, depends on the transport layer protocol used. When using TCP the remaining data to be send will be placed in the remote and the local socket buffer. The send routine returns when the amount of data remaining is less than the sum of both buffer sizes. In that case all remaining data is placed in the remote and local socket buffer. When both buffers are full but still there remains data to be send the send routine blocks until buffer space becomes available. No data is lost. When the remaining data is not received by a following receive routine, the send routine will block forever. However, when the remaining data is received by a following receive routine communication becomes corrupted. When using UDP all remaining data will be send and the send routine returns. No data is kept in the local socket buffer. When the remote socket buffer becomes full the remaining data is lost.

Synchronisation between the send routine of one application and the receive routine of another application

When using TCP, sending data from one application to another is suspended when the latter is not ready to receive the data because it is performing other tasks at that time. A send routine blocks when both remote and local socket buffers are full until buffer space becomes available. Buffer space becomes available when the remote application uses a receive routine to copy the data from the socket buffer to the application buffer. This way network communication is synchronised and no data is lost.

When using UDP, the sending application sends all data whether or not the receiving application is ready to receive the data. A send routine only blocks temporarily when the local socket buffer is full. All data that is send to the remote application after the remote application’s socket buffer has become full is discarded. When the remote application finally calls the corresponding receive routine this routine will block. It will never receive the number of bytes to be received.

Reliability of the underlying transport layer protocol

The transport layer provides transport of data from one application program to another. When data is lost during transport the number of bytes received by the receiving application is less than the number of bytes to be received. As a result the receive routine will block. In theory, TCP provides a reliable way of transport, whereas UDP provides an unreliable way of transport (see section 1.2). In practice, when using TCP, data was never lost.
When using UDP data was lost. Therefore communication between the two applications was corrupted. At this stage no concrete indications about the circumstances that cause the loss of data (e.g. the amount of data send at once or the rate at which small amounts are send) are known.

**Two (client) applications that communicate with a third (server) application at the same time**

When using TCP this does not cause any problems. A server application that handles communication sequentially handles communication with one client application at a time using a unique connection between the two applications, while a connection request from another client is placed in a queue. A concurrent server application handles communication with both client applications at the same time, using a unique connection to communicate with each client application.

When using UDP a client and a server application do not establish a unique connection before they actually start communication. The server application uses the same socket to communicate with all client applications that contact the server application. Even when the server communicates with different client applications concurrently it uses the same socket. Finally, the server application does not use a mechanism (e.g. a queue) to suspend communication with one client application while it finishes communication with another client application. When the communication between a client and a server application comprises more send and receive routines different communication processes may interfere. For example, a server, waiting for data at some point of the communication process with a client application, can receive data from any client application that sends data at that time. In practice the consequences of two or more client applications interfering depends on the situation. In general, the communication process between the server applications and all subsequent client application contacting it will be corrupted.

**One of the applications terminates execution premature**

If for example an application closes the socket and terminates execution based on some condition, or the application or the operating system crashes for some reason in which case the socket is closed incorrectly. TCP provides a mechanism to detect such a situation. The remaining application is notified by means of errors returned by the send and receive routines. UDP does not provide such a mechanism and the remaining application does not know whether the other application is running or not. A receive routine will block and a send routine will send all data.

### 5.3 Network Throughput

This section describes network throughput from the applications point of view. In this context network throughput is defined as the number of bytes transmitted from one application to another divided by the time interval between the first byte passed from the application buffer to the socket buffer on the sender's side and the last byte passed from the socket buffer to the application buffer on the receiver's side. The unit of network throughput is bytes per second (bps). In literature the abbreviation bps may refer to bits per second as well. However, in this section it always refers to bytes per second.

The network throughput is studied between application programs running on the front-end and the back-end processor of the data-acquisition system. The operating system running on the front-end processor is VxWorks, the operating system running on the back-end processor is OpenVMS. The applications programs running on OpenVMS are written using the OpenVMS system service routines. The network throughput is studied in three different situations:
1) using TCP as transport layer protocol
2) using UDP as transport layer protocol
3) using the VxWorks zbuf socket interface on the front-end (and TCP as transport layer protocol)

The network throughput is measured as follows. One application starts a clock and sends a certain amount of data to the other application. Care is taken that, at this time, the other application is waiting for this data. When this other application has received the data it sends back a message. When the first application has received this message it stops the clock. The time it takes to send back the message is deducted from the total time. The number of bytes send is divided by this time to obtain the network throughput. The results are described below.

TCP as transport layer protocol

TCP provides a reliable connection-oriented way of transport. When using TCP as transport layer protocol no data is corrupted when transmitted from one application to another. To provide this way of transport the TCP software inside the operating system has a lot of computational overhead, e.g. for the positive acknowledgement mechanism (see section 3.3). In the description of the network throughput the following aspects are taken into account: the direction of data transport, the number of bytes transmitted, the size of the socket buffers and the variation in network load.

As a starting point the following situation is used. Data is sent from the VxWorks application to the OpenVMS application. The default socket buffer sizes are used. On both VxWorks and OpenVMS the default socket buffer size is 4096 bytes. Table 5.3 shows the network throughput as a function of the number of bytes transmitted.

Table 5.3: Network throughput as a function of the number of bytes transmitted. The first row shows the number of bytes transmitted. The second row shows the time in seconds needed to transmit the data. The third row shows the network throughput in bytes per second (bps). The data is transmitted from an application running on VxWorks to an application running on OpenVMS. The transport layer protocol used is TCP. The socket buffer sizes are 4096 bytes. The time values in this table are the mean values of 10 measurements. The error in the time values in the first two columns is 0.0005 ± 0.0005. The error is 1 in the least significant number for the time values in the remaining columns.

<table>
<thead>
<tr>
<th>Number of bytes send (bytes)</th>
<th>4</th>
<th>100</th>
<th>1000</th>
<th>2000</th>
<th>4096</th>
<th>10000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.002</td>
<td>0.003</td>
<td>0.006</td>
<td>0.014</td>
<td>0.13</td>
<td>1.32</td>
</tr>
<tr>
<td>Network throughput (bps)</td>
<td>8000</td>
<td>20000</td>
<td>50000</td>
<td>666666</td>
<td>682666</td>
<td>714285</td>
<td>769230</td>
<td>757575</td>
</tr>
</tbody>
</table>

For small amounts of data send the network throughput increases with an increasing number of bytes send. For larger amounts of data the network throughput is rather constant. This constant value is ± 750000 bps.

The size of the socket buffers does not influence the network throughput. The size of the send and receive buffers can be changed with routines to specify socket options (see section 5.1.2). Both the send and receive sockets where increased to 16384 bytes (4 x 4096). The network throughput showed the same function of the number of bytes transmitted as for socket buffers of 4096 bytes (see table 5.3).

Variations in the network load influence the network throughput. The figures 5.2a and 5.2b depict the time needed to send 100000 bytes and 100000000 bytes as a function of the time of day. The direction and the socket buffer sizes are the same as above. Since both processors are attached to the local network the influence of the variations of the network load on the network throughput is always present. In addition, the network load as a function of the time of day is always different. This kind of variations in the network load are also present in the situations described in the remainder of this section.
Network Communication between Application Layer Programs

Figure 5.2: a) The time needed to send 100000 bytes from a VxWorks application running on a Pentium 100 MHz processor to an OpenVMS application running on a DEC-alpha 133 MHz workstation. The mean value is 0.14 sec, the minimum value is 0.13 sec and the maximum value is 0.59 sec. These values correspond to a network throughput of 714285 bps, 769230 bps and 169491 bps respectively.

b) The time needed to send 1000000 bytes using the same configuration. The mean, minimum and maximum value of the time needed to send the data correspond to network throughput values of 732745 bps, 774234 bps and 239785 bps.

When data is send in the opposite direction, i.e. from the OpenVMS application to the VxWorks application, the course of the network throughput as a function of the number of bytes send is the same. However, the constant value that is reached for large amounts of data send is $\pm 800000$ bps. In addition, when the socket buffer sizes are increased to 16384 bytes this value is even $\pm 950000$ bps. When the socket buffer sizes are increased to 52428 bytes, the maximum socket buffer size on VxWorks, the network throughput decreases again to $\pm 700000$ bps. This value shows larger variations than in the case of smaller socket buffer sizes. An explanation for this result is not known at this moment. There seems to be an optimal socket buffer size.

UDP as transport layer protocol

The transport layer protocol UDP provides an unreliable, connectionless way of transport. Data may be lost or corrupted. The TCP/IP software that provides this transport is less complex than the software that provides reliable, connection-oriented transport. The software has less computational overhead. In theory this must result in faster network throughput. This section describes the network throughput measured in practice.

Section 5.2 showed that when UDP is used as transport layer protocol data is sometimes lost when it is
transmitted from one application to another, especially when large amounts of data are transmitted. This makes it difficult to measure the network throughput for large amounts of data. Furthermore, only the default socket buffer sizes are used. On VxWorks the default send socket buffer is 2014 bytes and the default receive socket buffer is 4028 bytes. On OpenVMS both default send and receive socket buffer sizes are 9000 bytes.

When sending data from the VxWorks application to the OpenVMS application the course of the network throughput as a function of the number of bytes sent is the same as when using TCP. The largest amount of data that is transmitted to measure the network throughput is 40000 bytes. For small amounts of data the network throughput increases with an increasing number of bytes sent. The time needed to send these small amounts of data is the same as when using TCP. For larger amounts of data the network throughput is again rather constant. This constant value is ± 950000 bps, where it was ± 750000 bps when using TCP.

When sending data from OpenVMS to VxWorks the largest amount of data that could be transmitted was 4096 bytes. Up to this amount of data the network throughput is the same as when using TCP.

**Zbuf socket interface**

The zbuf socket interface is a zero-copy interface. Data is passed between application layer and transport layer without copying the data from the application buffer to the socket buffer. Instead the data is placed in a zbuf and in addition, this zbuf is transmitted between the two layers without copying the actual data. Copying data is a time consuming operation. Especially when large amounts of data are copied. In theory this must result in a higher network throughput than when using the socket interface. This section describes the network throughput when using the zbuf socket interface in practice. The zbuf socket interface can be used with both TCP and UDP as transport layer protocol. The network throughput is studied with TCP as transport layer protocol. Only the default TCP socket buffer sizes are used (i.e. 4096 bytes).

Data can be sent from VxWorks to OpenVMS in two ways. The application can place the data to be transmitted in a zbuf itself or it can have the operating system place the data in a zbuf. There is no significant difference in the network throughput when using either of these two options. In addition, in both cases the network throughput is the same function of the number of bytes send as with the socket interface (see table 5.3).

When sending data from OpenVMS to VxWorks the same situation occurs. The network throughput as a function of the number of bytes send is the same for using the socket interface and the zbuf socket interface.

### 5.4 Network Communication in Applications with a Graphical User Interface

This section describes how network communication is incorporated in an application program with a graphical user interface. The control flow of such an application makes the implementation of network communication more complex. Therefore, a brief description of the general and how this concept affects the implementation of the network communication is given in section 5.4.1. Implementation details depend on the operating system that runs the application and the software development tool used to develop the application program and the graphical user interface (GUI). Section 5.4.2 describes the actual implementation of network communication in an application program with a GUI.

#### 5.4.1 General Concept

A graphical user interface allows the user to interactively control the execution of an application program. Operations on the GUI, such as clicking on a button or selecting a menu option, generate events. These events are recognised and processed by the application program (see figure 5.3). In general, applications with a GUI use
the mechanism of callback functions to provide the link between the events and the application program. When building a graphical user interface callback functions can be associated with operations on the GUI that generate events. The associated callback function is executed when an event is generated. The source code for these callback functions is specified in the application program (see figure 5.3, application program A). Besides the code for the callback functions the application program consists of a relatively small main() routine. The final routine of main() is a routine that runs the graphical user interface (e.g. RunUserInterface()). After this routine is called all events are passed, along with some additional information, to the associated callback functions, which then executes. In addition, a callback function may influence the graphical user interface. For example, it may display the results of a certain operation on the GUI. The RunUserInterface() routine never returns so posterior routines in main() will not be executed.

Network Communication between Application Layer Programs

Figure 5.3: The general concept of an application program with a graphical user interface. Application program A shows the use of the callback mechanism to run a graphical user interface. For a description of the callback mechanism see the text. Application program B shows how network communication can be incorporated in an application running a graphical user interface. For a description of this mechanism see the text.

To perform network communication, the communication process as discussed in chapter 4 must be incorporated into the application program with a graphical user interface. The implementation of the server functionality into such an application differs from the implementation of the client functionality. This difference is based on the fact that a server awaits to be contacted and that a client initiates contact.

Usually a server awaits to be contacted by clients from the moment it starts execution. When the server is contacted, it communicates with the client and awaits to be contacted by other clients again. This loop is repeated infinitely.

A problem arises when the server must be implemented in an application with a GUI. When the server is implemented at the beginning of the main() routine the RunUserInterface() routine will not be executed. The application will not be able to run the graphical user interface. When the server is implemented at the end of the
main() routine it will not be executed because the RunUserInterface() routine does not return. Finally, the server may be implemented in a callback function. However, once this callback function is executed no other callback functions can be executed anymore because the server callback function contains an infinite loop. Therefore, the application will not be able to perform network communication and to run the GUI at the same time.

To solve this problem the following mechanism is used to make the application program respond to clients that contact the server part of the application and events that are generated on the GUI in the same way (see figure 5.3, application program B). The InitServer() routine is executed before the RunUserInterface in the main() routine. This routine initiates a mechanism that generates a sort of event when a client contacts the server and specifies the corresponding callback function that must be executed (the ServerCallback() routine). In addition the routine prepares the server to be able to await contact by clients. However, it actually awaits contact in the background of the application while the application starts to run the GUI. When the server is contacted by a client (see figure 5.3, remote application) the event is generated and the callback function is executed. This callback function handles the actual network communication between the client and the server. Furthermore the callback function enables the server to await to be contacted by other clients in the background. This closes the infinite loop of the server part of the communication process.

Usually a client starts execution when communication with a server is wanted. In addition, the client terminates execution after the communication between the client and the server is finished. Therefore, the complete client part of the communication process can be implemented in one callback function that is executed when network communication is wanted by the application. For example, clicking on the Client button executes the ClientCallback() routine (see figure 5.3). This routine contacts the server (see figure 5.3, remote application) and handles further communication. When communication is finished the callback function returns and other callback functions can be processed.

For both client and server the actual network communication is based on sequence of send and receive routines. All actual network communication may be performed within one callback function. When the network communication takes a long time the execution of callback functions associated to operations on the GUI is blocked. For example, the ClientCallback() routine sends a request to a server and waits for the result. The server application performs a time consuming task based on this request and sends the result back. To solve this problem the actual network communication may be divided over more than one callback function. One client callback function can send the request to the server and use the same mechanism as described above to await the result in the background and to have a second callback function executed when the result arrives. In the mean time the application can run the graphical user interface. The way the actual communication is best implemented depends on the purpose of the communication.

5.4.2 Implementation

Details of the implementation of network communication in an application with a GUI depend on the operating system that runs the application and on the software development tool used to develop the application program and the GUI. The first part of this section describes the implementation of network communication in applications with a GUI running on an OpenVMS operating system. An X Window system must be running to be able to run applications with a GUI. The application is developed using the software development tool X-Designer. This situation corresponds to the situation that is used to develop and run applications with a GUI on the back-end processor of the data-acquisition system. The second part describes the structure of GUI applications running on Windows 95 that are build with the use of the software development tool LabWindows/CVI. This configuration can be used in data-acquisition systems used for less complex experiments than the model experiment described in this report. More information about the X Window system and GUI application development tools for both OpenVMS and Windows95 can be found in chapter 2.
Network Communication between Application Layer Programs

OpenVMS/X Window and X-Designer

The description of the implementation of network communication in an application program with a GUI running on OpenVMS/X Window is based on an example. In this example the server side of a communication process based on TCP as transport layer protocol is implemented in a GUI application. The application awaits contact by a client application and in the meantime runs the GUI. It uses the callback function mechanism to run the GUI. When the application is contacted by a client further communication is performed in one single callback function. The source code of this application can be found in appendix A.15.

The main() routine of an application with a GUI is rather small. A part of the main() routine is related to the GUI, the other part is related to network communication. The part of the main() routine related to the GUI is generated by X-Designer automatically. It comprises a few routines to initialise the GUI and a routine that runs the graphical user interface (see section 5.4.1). This routine is called `XtAppMainLoop()` for the OpenVMS/X Window system. This routine is the last routine called in main(). The part of the main() routine that is related to network communication consists of the `XtAppAddInput()` routine provided by X Window and the user provided `JinitServer()` routine.

The `XtAppAddInput()` routine provides a mechanism to add an alternative source of input to the application program besides events generated by the GUI and to have a callback function executed in response to input of this kind. The routine specifies an event flag (ADD_INPUT_EFN) that functions as the additional source of input. Furthermore the routine specifies a callback function (`SockInCallback()`) to be invoked when input becomes available, that is, when the event flag is set. The X Window system monitors the event flag and handles the callback function like all other callback functions. Because this routine adds an alternative source of input it must be called prior to the routine that runs the GUI. This mechanism is used to have the application program respond to incoming connection requests from client applications, because the server part of the application awaits connection requests in the background. The event flag is set when a new connection request from a client application is accepted by the server. How the event flag is set is discussed below. The actual communication is handled by the `SockInCallback()` routine.

The `JinitServer()` routine prepares the application to be able to await incoming connection requests in the background. This routine must be called before the application starts running the GUI as well. This routine comprises all functionality from creating the device socket to accepting a connection request. This functionality is implemented by means of the same system service routines and in the same order as described in section 5.1. However, the routines that are used to accept a new connection request are combined in the `QIOAccept()` routine to be able to await the connection request in the background.

The `QIOAccept()` routine awaits new connection requests in the background and provides a mechanism to set the event flag when a new connection request is accepted. In the meantime the application program can run the GUI. First of all, the `QIOAccept()` routine resets the event flag. It creates a new internet pseudo-device and queues the appropriate I/O function to accept a new connection request on the channel associated to the internet pseudo-device. The `QIOAccept()` routine uses a `SYSS$QIO()` system service routine to queue this I/O operation (see section 5.1.1). The `SYSS$QIO()` system service provides asynchronous completion of the queued I/O function, i.e., it returns after the I/O function is queued rather than after the I/O function is completed. Therefore, the application can continue execution while it waits for new connection requests in the background. In addition, the `SYSS$QIO()` routine provides a mechanism to notify the application that the I/O operation is completed. An asynchronous system trap (AST) routine can be specified with the call to `SYSS$QIO`. This routine is executed as soon as possible after the accept I/O operation completes. The AST routine `Specified` with the call to `SYSS$QIO()` in the `QIOAccept()` routine is called `AstCompletion()` routine.

The `AstCompletion()` routine checks the I/O status block specified with the call to `SYSS$QIO()` to see whether the accept I/O operation is completed successful or not. When it is completed successful the routine sets the event flag.
The associated callback function `SockInCallback()` is called to perform further network communication, when the event flag is set. In this example further network communication comprises receiving data from and sending data to the client application. And in addition close the device socket created to communicate with this client application. Again the routines used are the same as described in section 5.1.4. and 5.1.5. Finally the QIOAccept() routine is called again to await a new connection request in the background. This closes the loop in the server part of the connection oriented communication process.

The final part of the communication process, i.e. closing the socket that the server uses to receive connection requests, is implemented in another callback function. Namely the callback function that is executed when the server application is closed by the user. For example, by means of selecting the menu option Quit. In this example this callback function is called `Micro_system_quit()`. The body of this callback function is generated by X-Designer itself when it was associated to the menu item Quit. The code of the callback function is specified by the user. In this case it contains the system service routines to close a socket and to exit the application.

Windows95 and LabWindows/CVI

This section briefly describes the possibilities with respect to implementation of network communication in an application program with a GUI developed by means of LabWindows/CVI. No concrete examples are presented. These possibilities are studied to determine whether network communication can be incorporated in applications with a graphical user interface developed by LabWindows/CVI.

Unlike X-Designer, LabWindows/CVI provides a set of routines to perform network communication in an application with a GUI itself. More information about these routines and how they should be used can be found in chapter 7 of [Lab 94b].

The network communication as implemented by these routines is based on TCP/IP and the socket interface between application layer and transport layer. The set of routines provided by the TCP Library hides a lot of network communication details. In addition, these routines have integrated network communication with another application further into the mechanism of event generation and callback functions.

This makes it easy to implement network communication into the application. For example, the RegisterTCPServer() routine needs only the local port number as parameter to prepare the server to be able to be contacted by a client application. In addition a callback function is specified with the RegisterTCPServer() routine. This callback routine is called upon every network communication related interaction with the remote application. In addition, from the application's point of view, different events are generated upon different interactions with the remote application. The specific event is passed to the callback function. The callback function performs the task corresponding to the event it is passed. For instance the callback function calls the ServerTCPread() routine to receive data when the event TCP_DATAREADY is passed to it.

A disadvantage of these predefined routines is that they reduce the possibilities with respect to the implementation of communication processes. For example UDP can not be used as transport layer protocol, socket buffer sizes can not be modified and server applications can not work concurrent. In addition, because the routines hide the mechanism of both network communication and callback functions, they make correct implementation of more complex communication processes more difficult.
Conclusions

The new data-acquisition system described in this report is still in the development stage. A number of aspects related to this data-acquisition system are used for the first time: the network connection between the front-end and back-end processor, VxWorks and Tornado and two software development tools to build application programs with a graphical user interface, X-designer and LabWindows/CVI.

A conceptual design for the data-acquisition software is made as a starting point for the further design and implementation of this software. To integrate the application software that is needed to automate the different experiments, this design is based on a model experiment. The design comprises four components: a graphical user interface, on-line application software, off-line application software and data storage. The different components can be designed and implemented relatively independent of one another. The design of the individual components must be based on the model experiment as well. Of course details will depend on the demands with respect to the different experiments. In addition, these components can be combined to form the complete data-acquisition software. A design is made for the graphical user interface, the other components of are for further investigation.

The Ethernet network connection between the front-end and the back-end processor of the data-acquisition system together with the TCP/IP communication system software provide a transparent way of communication between data-acquisition application software that runs on the two processors. All kinds of data can be send directly from one application to another and in both directions. The applications can use TCP-based sockets to perform fast and reliable network communication. The network throughput is \( 800000 \) bytes per second. No data is lost or corrupted. Due to this transparency the data-acquisition software can be distributed over the front end and back end of the data-acquisition system in such a way that it provides the best performance and user-friendliness with respect to the automation of experiments and at the same time still form a whole. This is used in the new design of the data-acquisition software. The application software that is needed for real-time experiment control and data acquisition runs on the front-end processor. For the same reason the operating system that runs on this processor is the real-time operating system VxWorks. In addition, the application software needed for the control and monitoring of the experiment by means of a graphical user interface and off-line analysis runs on the back-end processor.

The real-time features of the VxWorks operating system (e.g. task priority scheduling and interrupt handling) are not studied as part of the work described in this report. VxWorks is only used in relation to network communication. The TCP/IP communication software incorporated in VxWorks and the set of socket routines included with VxWorks provide a good way to implement network communication into application software. However, the zbuf socket interface, presented as a VxWorks facility to increase network throughput with respect to the standard socket interface, does not realise the desired result in practice. VxWorks and the corresponding cross-development environment Tornado form a good combination for the configuration of the VxWorks operating system and the development and interactive testing and running of VxWorks applications.
X-Designer is a good tool to build the graphical user interface of the data-acquisition system. It generates all the code needed to display and operate the graphical user interface automatically. The generated code is organised in such a way that the experimenter only has to fill in the source code that needs to be executed upon interaction with the interface at places that are left blank for that purpose by X-designer. Furthermore, X-designer makes it possible to run a prototype of the graphical user interface even before the experimenter has provided this code.

LabWindows/CVI incorporates both functionality to write an application for data-acquisition purposes (function libraries) and to build a graphical user interface for this application. These applications run on Windows95. In general, however, (real-time) data acquisition and running a graphical user interface are in contradiction with each other. This is the reason why these tasks are split up in the data-acquisition system described in this report. Therefore, LabWindows/CVI can only be used to write graphical user interface applications for the automation of experiments with little demands.
References


Appendix A

Source Codes

This appendix contains source codes related to the routines described in chapter 5.

A.1 Create a Socket
   A.1.1 the socket() routine
   A.1.2 the SYSSASSIGN() and SYSSQIOW(IO$SETMODE(p1)) routines

A.2 Specify Socket Options
   A.2.1 the setsockopt() routine
   A.2.2 the SYSSQIOW(IO$SETMODE(p5)) routine

A.3 Bind the Local Communication Endpoint to the Socket
   A.3.1 the bind() routine
   A.3.2 the SYSSQIOW(IO$SETMODE(p3)) routine

A.4 Listen for Connection Requests
   A.4.1 the listen() routine
   A.4.2 the SYSSQIOW(IO$SETMODE(p4)) routine

A.5 Request a Connection
   A.5.1 the connect() routine
   A.5.2 the SYSSQIOW(IO$ACCESS(p3)) routine

A.6 Accept Connection Requests
   A.6.1 the accept() routine
   A.6.2 the SYSSASSIGN() and SYSSQIOW(IO$ACCESSIO$ACCEPT(p3, p4)) routines

A.7 Send Data through a TCP-based Socket
   A.7.1 the send() routine
   A.7.2 the SYSSQIOW(IO$WRITEVBLK(p1, p2)) routine

A.8 Receive Data through a TCP-based Socket
   A.8.1 the recv() routine
   A.8.2 the SYSSQIOW(IO$READVBLK(p1, p2)) routine

A.9 Send Data through a UDP-based Socket
   A.9.1 the sendto() routine
   A.9.2 the SYSSQIOW(IO$WRITEVBLK(p1, p2, p3)) routine

A.10 Receive Data through a UDP-based Socket
    A.10.1 the recvfrom() routine
    A.10.2 the SYSSQIOW(IO$READVBLK(p1, p2, p3)) routine

A.11 Send Data through a TCP-based Zbuf Socket
    A.11.1 the zbufSockBufSend() routine
A.11.2 the zbufSockSend() routine

A.12 Receive Data through a TCP-based Zbuf Socket
A.12.1 the zbufSockRecv() routine

A.13 Shutdown a Connection and Close a Socket
A.13.1 the shutdown() and close() routines
A.13.2 the SYSSQIOW(IO$_DEACCESSIO$M_SHUTDOWN(p4)), SYSSQIOW(IO$_DEACCESS )
and SYSS$DASSIGN() routines

A.14 Spawn a Task
A.14.1 the taskSpawn() routine

A.15 An Application with a Graphical User Interface

A.1 Create a Socket

A.1.1 the socket() routine

```c
int main()
{
    int sFd;
    
    sFd = socket(AF_INET, SOCK_STREAM, 0);
    
    return (0);
}
```

A.1.2 the SYSSASSIGN() and SYSSQIOW(IO$_SETMODE(p1)) routines

```c
int main()
{
    short channel;
    struct dsc$descriptor
        inet_dev = {10, DSC$CLASS_S, DSC$DTYPE_T, "UCX$DEVICE"};
    short sck_parm[2];
    int status;
    short iosb[4];
    
    status = sys$assign(&inet_dev, &channel, 0, 0);
    sck_parm[0] = UCXSC_TCP;
    sck_parm[1] = INET_PROTPSC_STREAM;
    status = sys$qiow(3,
        channel,
        IO$_SETMODE,
        iosb,
        0, 0,
        &sck_parm, /* p1 */
        0, 0, 0, 0);
    
    return (0);
}
```
A.2 Specify Socket Options

A.2.1 the setsockopt() routine

```c
int main()
{
    int sFd;
    int soptval;
    int goptval, goptlen;

    soptval = 1;
    setsockopt(sFd, SOL_SOCKET, SO_REUSEADDR, (char *)&soptval, sizeof(soptval));

    goptlen = sizeof(goptval);
    getsockopt(sFd, SOL_SOCKET, SO_REUSEADDR, (char *)&goptval, &goptlen);

    return (0);
}
```

A.2.2 the SY$QIOW(IO$_SETMODE(p5)) routine

```c
int main()
{
    short channel;
    int status;
    short iosb[4];
    static int one= 1;
    static int setsbufsize=1000;
    static int setrbufsize=1000;

    int getsbufsize, getrbufsize;
    int retlen_getsbufsize, retlen_getrbufsize;

    struct {short lengte, parameter; void *pointerl; void *pointer2;
        static item-lijst = {
            {sizeof(getsbufsize), UCX$C-SNDBUF, &getsbufsize, &retlen_getsbufsize},
            {sizeof(getrbufsize), UCX$C-RCVBUF, &getrbufsize, &retlen_getrbufsize}};

    struct {short len, param; void *ptr[1];
        static item-list = {
            {sizeof(one), UCX$C-REUSEADDR, &one},
            {sizeof(setsbufsize), UCX$C-SNDBUF, &setsbufsize},
            {sizeof(setrbufsize), UCX$C-RCVBUF, &setrbufsize}};

    options = {sizeof(item-list), UCX$C-SOCKOPT, item-list},
    getoptions = {sizeof(item-lijst), UCX$C-SOCKOPT, item-lijst};

    status = sys$qiow(3,
        channel,
        IO$_SETMODE,
        iosb,
        0, 0,
        0, 0,
        0, 0,
        &options,
        /* p5 */
        0);

    status = sys$qiow(3,
        channel,
        IO$SENSEMODE,
        iosb,
```
A.3 Bind the Local Communication Endpoint to the Socket

A.3.1 the bind() routine

```c
#define SERVERPORT-NUM 2000
#define SERVER-INET-ADDR "131.155.114.181"

int main()
{
    int sFd;
    struct sockaddr_in localAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);

    bzero ((char *) &localAddr, sockAddrSize);
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons (SERVER_PORT-NUM);
    localAddr.sin_addr.s_addr = inet_addr (SERVER-INET-ADDR);
    bind (sFd, (struct sockaddr *) &localAddr, sockAddrSize);

    return (0);
}

A.3.2 the SYS$QIOW(IO$_SETMODE(p3)) routine

#define SERVER_PORT_NUM 2000
#define SERVER_INET_ADDR "131.155.114.11"

int main()
{
    short channel;
    int status;
    short iosb[4];
    struct sockaddr_in localAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);
    struct IL2 {unsigned int il2_length; char *il2_address;}
    iilist_adrs = {sockAddrSize, (char *)&localAddr};

    bzero ((char *) &localAddr, sockAddrSize);
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons (SERVER_PORT_NUM);
    localAddr.sin_addr.s_addr = inet_addr (SERVER_INET_ADDR);

    status = sys$qiow(3,
        channel,
        IOS_SETMODE,
```
A.4 Listen for Connection Requests

A.4.1 the listen() routine

```c
int main()
{
    int sFd;
    listen (sFd,5);
    return (0);
}
```

A.4.2 the SYSSQIOW(IOS_SETMODE(p4)) routine

```c
int main()
{
    short channel;
    int status;
    short iosb[4];
    status = sys$qiow(3,
        channel,
        IOS_SETMODE,
        iosb,
        0, 0,
        0, 0,
        0,
        5,  /* p4 */
        0, 0);
    return (0);
}
```

A.5 Request a Connection

A.5.1 the connect() routine

```c
#define SERVER_PORT_NUM 2000
#define SERVER_INET_ADDR "131.155.114.11"
```
int main()
{
    int sFd;
    struct sockaddr_in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);

    bzero ((char *)&localAddr, sockAddrSize);
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons (SERVER_PORT_NUM);
    localAddr.sin_addr.s_addr = inet_addr (SERVER_INET_ADDR);

    connect(sFd, (struct sockaddr *)&localAddr, sockAddrSize);

    return (0);
}

A.5.2 the SYSSQIOW(IO$-ACCESS(p3)) routine

#define SERVER_PORT_NUM 2000
#define SERVER_INET_ADDR "131.155.114.181"

int main()
{
    short channel;
    int status;
    short iosb[4];
    struct sockaddr_in localAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);
    struct II2 {unsigned int il2_length; char *il2_address;}
    rhst_adrs = {sockAddrSize, (char *)&localAddr};

    bzero ((char *)&serverAddr, sockAddrSize);
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons (2000);
    localAddr.sin_addr.s_addr = inet_addr (131.155.114.181);

    status = sys$qiow(3,
        channel,
        IO$-ACCESS,
        iosb,
        0, 0,
        0, 0,
        &rhst_adrs, /* p3 */
        0,
        0, 0);

    return (0);
}
A.6 Accept Connection Requests

A.6.1 the accept() routine

```c
int main()
{
    int sFd, newFd;
    struct sockaddr_in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);

    for(;;)
    {
        newFd = accept (sFd, (struct sockaddr *) &remoteAddr, &sockAddrSize);
        ...
    }
    ...
    return (0);
}
```

A.6.2 the SYS$ASSIGN() and SYS$QIOW(IO$_ACCESS|IO$M_ACCEPT(p3, p4)) routines

```c
int main()
{
    short channel, channel_1;
    struct dsc$descriptor inet_dev = [10, DSC$K_CLASS_S, DSC$K_DTYPE_T, "UCX$DEVICE"];
    int status;
    short iosb[4];
    struct sockaddr_in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);
    int r_relen;
    struct IL3 {unsigned int il3_length; char *il3_address; unsigned int il3_relen;}
    rhst_adrs = {sockAddrSize, (char *)&remoteAddr, &r_relen};

    for(;;)
    {
        status = sys$assign(&inet_dev, &channel_1, 0, 0);

        status = sys$qiow(3,
            channel,
            IOS$_ACCESS|IOS$M_ACCEPT,
            iosb,
            0, 0,
            0, 0,
            &rhst_adrs, /* p3 */
            &channel_1, /* p4 */
            0, 0);

        ...
    }

    return (0);
}
```
A.7 Send Data through a TCP-based Socket

A.7.1 the send() routine

```c
#define AsBUFSIZE 1000
int main()
{
    int sFd;
    int *sendbufptr;
    int totalbytesend;

    sendbufptr = (int*)malloc(AsBUFSIZE);

    totalbytesend = send(sFd, (char*)sendbufptr, AsBUFSIZE, 0);

    free ((void*)sendbufptr);
    return (0);
}
```

A.7.2 the SYSSQIOW(IOS_WRITEVBLK(p1, p2)) routine

```c
#define AsBUFSIZE 1000
int main()
{
    short channel;
    int status;
    short iosb[4];
    int *sendbufptr;
    int totalbytesend;

    sendbufptr = (int*)malloc(AsBUFSIZE);

    status = sys$qiow(3,
        channel,
        IOS_WRITEVBLK,
        iosb,
        0, 0,
        (char*)sendbufptr, /* p1 */
        AsBUFSIZE, /* p2 */
        0, 0,
        0, 0);

    totalbytesend = iosb[1];

    free ((void*)sendbufptr);
    return (0);
}
```
A.8 Receive Data through a TCP-based Socket

A.8.1 the recv() routine

```c
#define ArBUFSIZE 1000
int recvAllSend(int sFd, int *RecvBufPtr, int BytesRecvReQ, int totalBytesRecv);
```

```c
int main()
{
  int sFd;
  int retval;
  int *recvbufptr;
  int totalbytesrecv;

  recvbufptr = (int*)malloc(ArBUFSIZE);
  retval = recvAllSend(sFd, recvbufptr, ArBUFSIZE, &totalbytesrecv);
  if ((retval == ERROR) || (retval == 0))
  {
    /* Take the proper actions upon an error in the recvAllSend routine. These actions depend on the purpose of the
     communication between the local and the remote application. For example, the application may be closed or the remote
     application may be notified that an error occurred and that it should send the data again. */

    free((void*)recvbufptr);
    return (0);
  }

  while (*totalBytesRecv < BytesRecvReQ)
  {
    BytesRecv = recv(sFd, recvbufptr, BytesLeft, 0);
    if ((BytesRecv == ERROR) || (BytesRecv == 0))
      return (BytesRecv);
    *totalBytesRecv += BytesRecv;
    BytesLeft -= BytesRecv;
    tmpRecvPtr += BytesRecv;
  }
  return (13);
}
```

A.8.2 the SYS$QIOWIOS_READVBLK(p1, p2) routine

```c
#define ArBUFSIZE 1000
int recvAllSend(short channel, int *RecvBufPtr, int BytesRecvReQ, int totalBytesRecv);
```
int main()
{
    short channel;
    int status;
    int *recvbufptr;
    int totalbytesrecv;

    recvbufptr = (int*)malloc(ArBUFSIZE);

    status = recvAllSend(channel, recvbufptr, ArBUFSIZE, &totalbytesrecv);
    if (!(status & 1))
    {
        /* Take the proper actions upon an error in the recvAllSend routine. These actions depend on the purpose of the
           communication between the local and the remote application. For example, the application may be closed or the remote
           application may be notified that an error occurred and that it should send the data again. */
    }

    free ((void *)recvbufptr);
    return (0);
}

int recvAllSend(short channel, int *RecvBufPtr, int BytesRecvReQ, int *totalBytesRecv)
{
    int BytesLeft;
    char *tmpRecvPtr;
    int status;
    short iosb[4];

    *totalBytesRecv = 0;
    BytesLeft = BytesRecvReQ;
    tmpRecvPtr = (char *)RecvBufPtr;
    while (*totalBytesRecv < BytesRecvReQ)
    {
        status = sys$qiow(3,
            channel,
            IO$_READVBLK,
            iosb,
            0, 0,
            tmpRecvPtr,       /* p1 */
            BytesLeft,        /* p2 */
            0, 0,
            0, 0);
        if (status & 1) status = iosb[0];
        if (!(status & 1))
            return(status);
        *totalBytesRecv += iosb[1];
        BytesLeft -= iosb[1];
        tmpRecvPtr += iosb[1];
    }
    return (1);
}
A.9 Send Data through a UDP-based Socket

A.9.1 the sendto() routine

```c
#define SERVER_PORT_NUM 2000
#define SERVER_INET_ADDR "131.155.114.11"
#define AsBUFSIZE 1000

int SendtoMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend, struct sockaddr_in *SendTo);

int main()
{
    int sFd;
    int *sendbufptr;
    struct sockaddr_in remoteAddr;
    int totalbytessend;

    sendbufptr = (int*)malloc(AsBUFSIZE);

    bzero ((char *) &remoteAddr, sockaddrSize);
    remoteAddr.sin_family = AF_INET;
    remoteAddr.sin_port = htons (SERVER_PORT_NUM);
    remoteAddr.sin_addr.s_addr = inet_addr (SERVER_INET_ADDR);

    SendtoMult(sFd, sendbufptr, AsBUFSIZE, &totalbytessend, &remoteAddr);

    free ((void *)sendbufptr);
    return (0);
}

int SendtoMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend, struct sockaddr_in *SendTo)
{
    int BytesLeft, BytesSend;
    char *tmpSendPtr;
    int maxbytes, nbytes;
    int sockaddrSize = sizeof (struct sockaddr_in);

    *totalBytesSend = 0;
    BytesLeft = ByteSendReQ;
    tmpSendPtr = (char *)SendBufPtr;
    maxbytes= min(AsBUFSIZE,32739);
    while(*totalBytesSend < ByteSendReQ)
    {
        nbytes = min(BytesLeft, maxbytes);
        BytesSend = sendto (sFd, tmpSendPtr, nbytes, 0, (struct sockaddr *) SendTo, sockaddrSize);
        if (BytesSend == ERROR)
            return (ERROR);
        *totalBytesSend += BytesSend;
        BytesLeft -= BytesSend;
        tmpSendPtr += BytesSend;
    }
    return (0);
}
```
A.9.2 the SYSSQIOW(IO$_WRITEVBLK(p1 p2, p3)) routine

```c
#define SERVER_PORT_NUM 2000
#define SERVER_INET_ADDR "131.155.114.181"
#define AsBUFSIZE 1000

int SendtoMult(short channel, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend, struct IL2 *SendTo);

int main()
{
    short channel;
    int *sendbufptr;
    struct sockaddr-in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr-in);
    struct IL2 {unsigned int il2_length; char *il2_address;};
    rhst_addrs = {sockAddrSize, (char *)&remoteAddr};
    totalbytessend;

    sendbufptr = (int*)malloc(AsBUFSIZE);
    bzero ((char *)&remoteAddr, sockAddrSize);
    remoteAddr.sin_family = AF_INET;
    remoteAddr.sin_port = htons (SERVER_PORT_NUM);
    remoteAddr.sin_addr.s_addr = inet_addr (SERVER_INET_ADDR);

    SendtoMult(channel, recvbufptr, ArBUFSIZE, &totalbytessend, &rhst_addrs);

    free ((void *)sendbufptr);
    return (0);
}

int SendtoMult(short channel, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend, struct IL2 *SendTo)
{
    int BytesLeft;
    char *tmpSendPtr;
    int maxbytes, nbytes;
    int status;
    short iosb[4];

    *totalBytesSend = 0;
    BytesLeft = ByteSendReQ;
    tmpSendPtr = (char *) SendBufPtr;
    if (SsBUFSIZE <= 32739)
        maxbytes = SsBUFSIZE;
    else
        maxbytes = 32739;
    while(*totalBytesSend < ByteSendReQ)
    {
        if (BytesLeft <= maxbytes)
            nbytes = BytesLeft;
        else
            nbytes = maxbytes;
        status = sys$qiow(3,
            channel,
            IOS_WRITEVBLK,
            iosb,
            0, 0,
            tmpSendPtr, /* p1 */
            nbytes, /* p2 */
            SendTo, /* p3 */
```
0,
0, 0);
if (status & 1) status = iosb[0];
if (!(status & 1))
    return(status);
*totalBytesSend += iosb[1];
BytesLeft -= iosb[1];
tmpSendPtr += iosb[1];
}
return(1);
}

A.10 Receive data through a UDP-based socket

A.10.1 the recvfrom() routine

#define ArBUFSIZE 1000
int RecvfromAllSend(int sFd, int *RecvBufPtr, int BytesRecvReQ, int *totalBytesRecv, struct sockaddr_in *RecvedFrom);

int main()
{
    int sFd;
    int *recvbufptr;
    struct sockaddr_in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr_in);
    int totalbytesrecv;

    recvbufptr = (int*)malloc(ArBUFSIZE);
    RecvfromAllSend(sFd, recvbufptr, ArBUFSIZE, &totalbytesrecv, &remoteAddr);

    free ((void *)recvbufptr);
    return (0);
}

int RecvfromAllSend(int sFd, int *RecvBufPtr, int BytesRecvReQ, int *totalBytesRecv, struct sockaddr_in *RecvedFrom)
{
    int BytesLeft, BytesRecv;
    char * tmpRecvPtr;
    int addrlen;
    int sockAddrSize = sizeof (struct sockaddr_in);

    *totalBytesRecv = 0;
    BytesLeft = BytesRecvReQ;
    tmpRecvPtr = (char *)RecvBufPtr;
    while (*totalBytesRecv < BytesRecvReQ)
    {
        addrlen = sockAddrSize;
        BytesRecv = recvfrom (sFd, tmpRecvPtr, BytesLeft, 0, (struct sockaddr *) RecvedFrom, &addrlen);
        if (BytesRecv == ERROR)
            return (BytesRecv);
        *totalBytesRecv += BytesRecv;
        BytesLeft -= BytesRecv;
        tmpRecvPtr += BytesRecv;
    }
    return (0);
}
#define ArBUFSIZE 1000

int RecvfromAllSend(short channel, int *recvBufPtr, int BytesRecvReQ, int *totalBytesRecv, struct IL3 *RecvedFrom);

struct IL3 { unsigned int il3Length; char *il3_address; unsigned int *il3_retlen; };

int main()
{
    short channel;
    int *recvbufptr;
    struct sockaddr-in remoteAddr;
    int sockAddrSize = sizeof (struct sockaddr-in);
    struct IL3 rhst_adrs = {sockAddrSize, (char *)&remoteAddr, &r_retlen};
    int totalbytesrecv;

    recvbufptr = (int*)malloc(ArBUFSIZE);

    RecvfromAllSend(channel, recvbufptr, ArBUFSIZE, &totalbytesrecv, &rhst_adrs);

    free ((void *)recvbufptr);
    return (0);
}

int RecvfromAllSend(short channel, int *recvBufPtr, int BytesRecvReQ, int *totalBytesRecv, struct IL3 *RecvedFrom)
{
    int BytesLeft;
    char *tmpRecvPtr;
    int status;
    short iosb[4];

    *totalBytesRecv = 0;
    BytesLeft = BytesRecvReQ;
    tmpRecvPtr = (char *)recvBufPtr;
    while (*totalBytesRecv < BytesRecvReQ)
    {
        status = sys$qiow(3,
                          channel,
                          IOS_READVBLK,
                          iosb,
                          0, 0,
                          tmpRecvPtr, /* p1 */
                          BytesLeft, /* p2 */
                          RecvedFrom,
                          0,
                          0, 0);

        if (status & 1) status = iosb[0];
        if(!((status & 1))
            return (status);
        *totalBytesRecv += iosb[1];
        BytesLeft -= iosb[1];
        tmpRecvPtr += iosb[1];
    }

    return (1);
}
A.11 Send Data through a TCP-based Zbuf Socket

A.11.1 the zbufSockBufSend() routine

```c
#define AsBUFSIZE 1000

int zbufSockBufSendMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend);
void freeBuf(char * buf, int freeArg);

int main()
{
    int sFd;
    int *sendbufptr;
    int totalbytessend;
    
    sendbufptr = (int*)malloc(AsBUFSIZE);
    
    zbufSockBufSendMult(sFd, sendbufptr, AsBUFSIZE, &totalbytessend);
    
    free ((void *)sendbufptr);
    return (0);
}

int zbufSockBufSendMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend)
{
    int BytesLeft, BytesSend;
    char *tmpSendPtr;
    int maxbytes, nbytes;

    *totalBytesSend = 0;
    BytesLeft = ByteSendReQ;
    tmpSendPtr = (char *)SendBufPtr;
    maxbytes= min((SBUFSIZE/2),32767);
    counter = 0;
    while(*totalBytesSend < ByteSendReQ)
    {
        ++counter;
        nbytes = min(BytesLeft, maxbytes);
        BytesSend = zbufSockBufSend (sFd, tmpSendPtr, nbytes, freeBuf, 0, 0);
        if (BytesSend == ERROR)
            return(BytesSend);
        *totalBytesSend += BytesSend;
        BytesLeft -= BytesSend;
        tmpSendPtr += BytesSend ;
    }
    return(0);
}

void freeBuf(char * buf, int freeArg)
{
    counter—;
}
```
A.11.2 the zbufSockSend() routine

#define AsBUFSIZE 1000

int zbufSockSendMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend);
void freeBuf(char * buf, int freeArg);

int counter;

int main()
{
    int sFd;
    int *sendbufptr;
    int totalbytessend;

    sendbufptr = (int*)malloc(AsBUFSIZE);
    zbufSockSendMult(sFd, sendbufptr, AsBUFSIZE, &totalbytessend);

    free ((void *)sendbufptr);
    return (0);
}

int zbufSockSendMult(int sFd, int *SendBufPtr, int ByteSendReQ, int *totalBytesSend)
{
    int BytesLeft, BytesSend;
    char *tmpSendPtr;
    int nbytes, restbytes;
    ZBUF_ID zbufid;
    ZBUF_SEG zbufsegm;
    int zbuflen;

    *totalBytesSend = 0;
    BytesLeft = ByteSendReQ;
    tmpSendPtr = (char *)SendBufPtr;
    counter = 0;

    while(*totalBytesSend < ByteSendReQ)
    {
        counter++;
        nbytes = min(BytesLeft, (SBUSIZE/2));
        zbufid = zbufCreate();
        if (zbufid == NULL) return (ERROR);
        if (nbytes < 32767)
        {
            zbufsegm = zbufInsertBuf (zbufid, NULL, 0, tmpSendPtr, nbytes, freeBuf, 0);
            if (zbufsegm == NULL)
                return 0;
            tmpSendPtr += nbytes;
        }
        else
        {
            zbufsegm = zbufInsertBuf (zbufid, NULL, 0, tmpSendPtr, 32767, freeBuf, 0);
            if (zbufsegm == NULL)
                return 0;
            tmpSendPtr += 32767;
            restbytes = nbytes-32767;
            zbufsegm = zbufInsertBuf (zbufid, NULL, ZBUF_END, tmpSendPtr, restbytes, freeBuf, 0);
            if (zbufsegm == NULL)
                return 0;
            tmpSendPtr += restbytes;
        }
    }
}
A.12 Receive Data through a TCP-based Zbuf Socket

the \texttt{zbufSockRecv()} routine

#define ARBUFSIZE 1000

\texttt{ZBUF_ID} \texttt{zbufRecvAllSend(int sFd, int BytesRecvReQ, int \*totalBytesRecv);}
else 
{
  zbuftemp = zbufSockRecv(sFd, 0, &BytesLeft);
  if (zbuftemp == NULL)
    return(NULL)
  segmtmp = zbufInsert(zbufid, NULL, ZBUF_END, zbuftemp);
  if (segmtmp == NULL)
    return(NULL);

  (*totalBytesRecvd) += BytesLeft;
  BytesLeft = BytesRecvReQ - (*totalBytesRecvd);
}
return (zbufid); 

A.13 Shutdown a Connection and Close a Socket

A.13.1 the shutdown() and close() routines

int main()
{
  int sFd;

  shutdown(sFd,2);
  close(sFd);
  return (0);
}

A.13.2 the SYSSQIOW(IOS_DEACCESS|IOSM_SHUTDOWN(p4)), SYSSQIOW(IOS_DEACCESS) and
SYSSDASSIGN() routines

int main()
{
  short channel;
  int status;
  short iosb[4];

  status = syssqiow(3,
    channel,
    IOS_DEACCESS|IOSM_SHUTDOWN,
    iosb,
    0, 0,
    0, 0,
    0,
    "UCX$C_DSC_ALL", /* p4 */
    0, 0);

  status = syssqiow(3,
    channel,
    IOS_DEACCESS,
    iosb,
    0, 0,
    0, 0,
A.14 Spawn a Task

A.14.1 the taskSpawn() routine

```c
#define SERVERWORK_PRIORITY 100
#define SERVER_STACK_SIZE 32768

int tcpServerWorkTask (int sFd, char * address, u_short port);

int main()
{
    int sFd, newFd;
    struct sockaddr_in remotetAddr;
    char workName[16];
    int ix = 0;

    for (;;)
    {
        /* Accept new connection requests my means of the accept() routine. */
        sprintf (workName, "TcpWork%d", ix++);
        taskSpawn(workName, SERVERWORK_PRIORITY, 0, SERVER_STACK_SIZE,
            (FUNCPTR) tcpServerWorkTask,
            newFd,
            (int) inet_ntoa (remotetAddr.sin_addr),
            ntohs (remotetAddr.sin_port),
            0, 0, 0, 0, 0, 0);
    }

    return (0);
}

int tcpServerWorkTask(int sFd, char * address, u_short port)
{
    /* This routine contains the routines for further communication. */

    return (0);
}
```
A.15 An application with a Graphical User Interface

#define ADD_INPUT_EFN 6

void SockInCallback();
void Micro_system_quit(Widget, XtPointer, XmPushButtonCallbackStruct *);
void InitServer();
void QIOAccept();
void AstCompletion();

XtAppContext app_context;
short channel, newchannel;
short IOSB[4];

int main()
{
    /\ A few X Window routines to initialise the graphical user interface, generated by X-designer */
    XtAppAddInput (app_context, ADD_INPUT_EFN, 0, SockInCallback, 0);
    InitServer();
    XtAppMainLoop (app_context);
    exit (0);
}

/* Callback Functions */

void SockInCallback();
{
    if (!((IOSB[0] & 1))
    {
        /* Take the proper actions upon an error in the completion of the IOS_ACCESSIOMS_ACCEPT function queued with the SYSSQIO() system service routine in the QIOAccept() routine. These actions depend on the purpose of the communication between the local and the remote application. For example, the application accepts the next connection request. Because there is no connection established the application can not notify the remote application of this error. */
    }

    /* routines that perform the actual communication between the client and the server application after the connection is established and routines to close the newly created device socket when communication between the two application is finished. These routines are the same as discussed in section 4.1 */
    QIOAccept();
}

void Micro_system_quit (w, client_data, call_data)
{
    /* routines that close the socket the server uses to receive connection requests when the user pushes the quit button to exit the application program. These routines are the same as described in section 4.1 */
    exit(0);
}
/** Additional Routines */

void JInitServer()
{
/* routines that provide the functionality from creating a device socket to accepting a connection request from the queue
associated to this device socket. These routines are identical to the routines described in section 4.1. Actual accepting the
connection request is performed by the next routine QIOAccept */
QIOAccept();
}

void QIOAccept()
{
sys$clref(ADD_INPUT_EFN);

sys$sassign(&inet_dev, &newchannel, 0, 0);

sys$sqio(3,
    channel,
    IOS$ACCESSIOM_ACCEPT,
    IOSB,
    AstCompletion,    /**< AST completion routine */
    0,
    0, 0,
    &rhs t_adsr,
    &newchannel,
    0, 0);
}

void AstCompletion()
{
    sys$ssetef(ADD_INPUT_EFN);
}