User support for patient data analysis

Cluitmans, L.J.M.

DOI:
10.6100/IR527559

Published: 01/01/1999

Citation for published version (APA):
DOI: 10.6100/IR527559

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
• You may freely distribute the URL identifying the publication in the public portal

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.
User Support for Patient Data Analysis

Luc Cluitmans
User Support for Patient Data Analysis

Proefontwerp

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op maandag 29 November 1999 om 16.00 uur

door

Lucas Joseph Maria Cluitmans

geboren te Swalmen.
Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr.ir. A. Hasman
en
prof.dr.ir. P.P.J. van den Bosch

Copromotor:

dr.ir. J.A. Blom

© copyright 1999 L.J.M. Cluitmans
All rights reserved. No parts of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright owner.

Printed by: Eindhoven University Press Facilities

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Cluitmans, Lucas J.M.

User Support for Patient Data Analysis / by Lucas J.M. Cluitmans.
NUGI 832
Subject headings: patient monitoring / data analysis / software tools / data acquisition.
Summary

In a hospital’s intensive care unit many variables are monitored for every patient. This designer’s PhD thesis describes the design of software for the acquisition and analysis of such signals. The work described in this thesis comprises two parts.

The first part is a data acquisition system called HPDATA, that I designed for the intensive care unit (ICU) of the Catharina hospital in Eindhoven, the Netherlands. The HPDATA system continuously acquires data from all patient monitors in the ICU and makes the acquired data available in two different forms: as network packets and as files. Software running on computers connected to the ICU’s local area network can use the network packets as input for online patient monitoring software. The data files can be used directly by software that aids medical staff in assessment of the health of patients. The data files can also be used offline, as a record of a patient’s stay in the ICU or for medical research.

As an example of using the acquired data, I also developed HPView, a health assessment application for showing graphs of the variables measured. This application has been in use in the daily routine of the Catharina hospital’s ICU since September 1996.

Both types of acquired data can be used for a variety of computer aided patient monitoring (and related) applications. The second part of this thesis describes the software I developed to facilitate the design and implementation of such applications. This software has two major components: the ‘data processing core’ COPDA and the Data Wand program. Though originally conceived as tools for processing the data acquired by HPDATA, COPDA and Data Wand are not limited to this data, but can use other sources of data as well.

Data Wand is a program that allows its users to design and test data processing algorithms graphically. Data Wand is used for implementing and running data processing algorithms on offline data (data stored in files). The output of data processing performed using Data Wand can include graphs of processed signals, messages generated during processing and output data files. For medical research, this functionality in itself is already useful. In a wider perspective, Data Wand can also be seen as a data processing algorithm prototyping and development tool.

COPDA is the ‘data processing core’ that provides Data Wand with its data processing capabilities. The primary function of COPDA is to act as an interpreter for data processing algorithms described in the CPMDL language. Such descriptions can be developed using Data Wand. COPDA is implemented as a COM component that can be used in programs that need data processing capabilities. The two main guidelines used in the design of COPDA were maximum flexibility and minimum user interface involvement.

COPDA itself does not provide any kind of user interface, but leaves the presentation of its functionality to the user as a task for the program that uses COPDA. The minimum user interface involvement ensures that the development of the user interface of programs that use COPDA is unrestricted by COPDA.

Maximising flexibility influences many aspects of COPDA (and hence indirectly also Data Wand). Data input and output is not restricted to any particular file or network stream format, but uses abstractions called data sources and data destinations. Programs using
COPDA should implement these abstractions to write or read their own data formats (or they can use standard implementations for some well-known data formats). Another example of COPDA's flexibility is the extensibility of the set of functions and classes provided by the CPMDL language. This set can be extended by loading extension libraries, which are implemented as DLLs written in a conventional programming language (such as C or Object Pascal).

Some of the distinguishing features of COPDA are:

- Inherent support for processing ‘bad data’: missing samples or samples marked as ‘invalid’.
- Uniform treatment of different kinds of input data, being it data read from an (offline) file or data entering the data processing program as a (real-time) network stream.
- Synchronised processing of many signals, making multi-signal processing and data fusion easier.

Some of the distinguishing features of Data Wand are:

- Two seamlessly integrated styles of user interface for access to the data processing algorithm descriptions: a user interface for building algorithms by graphically connecting data processing primitives (modules) and a user interface for textually modifying the functionality of these primitives (by editing their CPMDL source).
- A powerful built-in tool for presenting input and output signals graphically.
- As Data Wand is based on COPDA, algorithms designed in Data Wand can be used in other COPDA-based applications without requiring any form of recompilation (provided they do not use Data Wand-specific features).

Data Wand has been used by students in the Medical Electrical Engineering group at Eindhoven University of Technology as a tool for analysing data recorded by the HPDATA system in a project investigating early predictions of myocardial infarcts. Suitability of Data Wand as an educational tool to introduce data processing to non-experts is currently under investigation.

Data Wand and COPDA are currently used in the European IBIS project. Data Wand is used as a tool to develop new medical data processing algorithms (called biosignal interpretation methods in this context). There are many types of algorithms to be developed in this project, ranging from signal validation to pattern recognition. The final goal of the IBIS project is to deliver an IBIS workstation, a system that actually runs these algorithms in real time. This system will be based on COPDA.
# Contents

Summary

Glossary

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Intensive care</td>
</tr>
<tr>
<td>1.2</td>
<td>Problem statement</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Data Acquisition</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Data analysis</td>
</tr>
<tr>
<td>1.2.3</td>
<td>DataWand and COPDA</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Design background</td>
</tr>
<tr>
<td>1.3</td>
<td>Reading guide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2.2</td>
<td>Application areas</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Target applications</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Delivering data to applications</td>
</tr>
<tr>
<td>2.3</td>
<td>Sample rates</td>
</tr>
<tr>
<td>2.4</td>
<td>Existing acquisition systems</td>
</tr>
<tr>
<td>2.5</td>
<td>Interface standards</td>
</tr>
<tr>
<td>2.6</td>
<td>Data analysis systems</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Literature</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Commercial data processing tools</td>
</tr>
<tr>
<td>2.7</td>
<td>Development methods</td>
</tr>
<tr>
<td>2.8</td>
<td>Component based design</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Reusability of software code</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Object oriented design</td>
</tr>
<tr>
<td>2.8.3</td>
<td>The Component Object Model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>The HPDATA System</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Designing the system</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Location of use</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Network infrastructure</td>
</tr>
<tr>
<td>3.3</td>
<td>SDN and the SDN adapter</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Overview of SDN</td>
</tr>
<tr>
<td>3.3.2</td>
<td>SDN data concepts</td>
</tr>
<tr>
<td>3.3.3</td>
<td>SDN PC interface adapter</td>
</tr>
</tbody>
</table>
4.9.2 Extending CPMDL

4.8.6 Module networks in the

4.8.4 An introduction to

4.8.3 CPMDL design

4.8.2 Introduction to

4.7.3 Data transfer model

4.7.1 Introducing the COPDA input/output model

4.6 Two views

4.5 Two views of COPDA

4.4 Defining 'software foundation'

4.4.2 Commercial tools

4.4.1 Function library

4.4 Initial requirements

4.3 Initial requirements

4.2 Interfaces and components

4.1 Summary

Chapter 4 Data Processing Core

4.7.4 Data sequencing model

4.7.3 Data transfer model

4.7.2 Signal model

4.7.1 Introducing the COPDA input/output model

4.7 Data interface: input and output

4.6 Software model

4.5 Two views of COPDA

4.4.4 Conclusion

4.4.3 Data processing core

4.4.2 Commercial tools

4.4.1 Function library

4.4 Defining 'software foundation'

4.3 Initial requirements

4.2 Interfaces and components

4.1 Summary
## Contents

4.10 Conclusions................................................................. 86
4.10.1 summary................................................................. 86
4.10.2 Requirement evaluation........................................ 87

### Chapter 5 Using COPDA

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 DataWand</td>
<td>89</td>
</tr>
<tr>
<td>5.1.1 Design goals</td>
<td>89</td>
</tr>
<tr>
<td>5.1.2 Design process of DataWand</td>
<td>90</td>
</tr>
<tr>
<td>5.1.3 Final design specifications</td>
<td>93</td>
</tr>
<tr>
<td>5.1.4 User interface design</td>
<td>95</td>
</tr>
<tr>
<td>5.1.5 Input and output specifications</td>
<td>97</td>
</tr>
<tr>
<td>5.1.6 Operation specification</td>
<td>100</td>
</tr>
<tr>
<td>5.1.7 CPMDL development</td>
<td>104</td>
</tr>
<tr>
<td>5.1.8 Evaluation of the usability of DataWand</td>
<td>105</td>
</tr>
<tr>
<td>5.1.9 Summary</td>
<td>107</td>
</tr>
<tr>
<td>5.1.10 Requirement evaluation</td>
<td>108</td>
</tr>
<tr>
<td>5.2 Example of using DataWand</td>
<td>108</td>
</tr>
<tr>
<td>5.3 Real time COPDA programs</td>
<td>111</td>
</tr>
</tbody>
</table>

### Chapter 6 Conclusions and Discussion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Data acquisition system</td>
<td>113</td>
</tr>
<tr>
<td>6.1.1 Recommendations</td>
<td>114</td>
</tr>
<tr>
<td>6.2 Data Processing Core</td>
<td>115</td>
</tr>
<tr>
<td>6.2.1 A summary of COPDA</td>
<td>115</td>
</tr>
<tr>
<td>6.2.2 The goals of COPDA</td>
<td>116</td>
</tr>
<tr>
<td>6.2.3 Modifications</td>
<td>119</td>
</tr>
<tr>
<td>6.3 DataWand</td>
<td>120</td>
</tr>
<tr>
<td>6.3.1 Modifications</td>
<td>121</td>
</tr>
<tr>
<td>6.3.2 User satisfaction</td>
<td>121</td>
</tr>
<tr>
<td>6.4 The future of COPDA and DataWand</td>
<td>122</td>
</tr>
</tbody>
</table>

### Appendix A: SBF files

<table>
<thead>
<tr>
<th>Appendix A: SBF files</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix B: Interfaces</td>
<td>127</td>
</tr>
<tr>
<td>Appendix C: CPMDL syntax</td>
<td>135</td>
</tr>
<tr>
<td>Appendix D: Gap-friendly first order lowpass filter</td>
<td>141</td>
</tr>
<tr>
<td>Appendix E: Configuration Specifications</td>
<td>143</td>
</tr>
</tbody>
</table>

### Bibliography

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samenvatting</td>
<td>149</td>
</tr>
</tbody>
</table>

### Curriculum Vitae

<table>
<thead>
<tr>
<th>Curriculum Vitae</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>151</td>
</tr>
</tbody>
</table>
Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABP</td>
<td>Arterial blood pressure</td>
</tr>
<tr>
<td>application</td>
<td>See the definition of program and application in section 4.2.2</td>
</tr>
<tr>
<td>COM</td>
<td>Component object model (see section 2.8.3).</td>
</tr>
<tr>
<td>component</td>
<td>(in chapters 2, 4, 5 and 6) An object implementing one or more interfaces.</td>
</tr>
<tr>
<td>COPDA</td>
<td>Core components for patient data analysis (the main topic of chapter 4).</td>
</tr>
<tr>
<td>CPMDL</td>
<td>COPDA module description language (the language interpreted by COPDA, see section 4.8)</td>
</tr>
<tr>
<td>data destination</td>
<td>A component implementing the output related interfaces for COPDA.</td>
</tr>
<tr>
<td>data source</td>
<td>A component implementing the input related interfaces for COPDA.</td>
</tr>
<tr>
<td>ECG</td>
<td>Electrocardiogram (measurement of electrical heart activity).</td>
</tr>
<tr>
<td>EEG</td>
<td>Electroencephalogram (measurement of electrical brain activity).</td>
</tr>
<tr>
<td>HPDATA</td>
<td>A system to acquire data from Hewlett Packard patient monitors: the data acquisition system whose design is described in chapter 3. Also used to indicate the format used for the files generated by this system, see SBF.</td>
</tr>
<tr>
<td>ICIS</td>
<td>Intensive care information system.</td>
</tr>
<tr>
<td>ICU</td>
<td>Intensive care unit</td>
</tr>
<tr>
<td>interface</td>
<td>(in chapters 2, 4, 5 and 6) A pure abstract class (section 2.8.2.5)</td>
</tr>
<tr>
<td>IP</td>
<td>Internet protocol, used as the base network protocol for TCP and UDP.</td>
</tr>
<tr>
<td>IPX</td>
<td>A local area network protocol, especially popular during the late 80's and most of the 90's. Its role is slowly being taken over by IP-based protocols</td>
</tr>
<tr>
<td>LAN</td>
<td>Local area network.</td>
</tr>
<tr>
<td>MIB</td>
<td>Medical information bus (section 2.5)</td>
</tr>
<tr>
<td>module</td>
<td>A primitive building block for describing data processing algorithms for COPDA. Modules for COPDA are described in the CPMDL language.</td>
</tr>
<tr>
<td>OOP</td>
<td>Object oriented programming</td>
</tr>
<tr>
<td>PDMS</td>
<td>Patient data management system.</td>
</tr>
<tr>
<td>program</td>
<td>See the definition of program and application in section 4.2.2</td>
</tr>
<tr>
<td>SBF</td>
<td>Sequential block file. See sections 3.6.3 and appendix A.</td>
</tr>
<tr>
<td>SDN</td>
<td>Serial data network (see section 3.3)</td>
</tr>
<tr>
<td>Speedy</td>
<td>The PC plug-in board for communicating with the SDN network. (section 3.3.3)</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission control protocol. A stream-based network protocol normally used on the internet (also referred to as TCP/IP).</td>
</tr>
<tr>
<td>UDP</td>
<td>User datagram protocol. A datagram-based network protocol normally used on the internet (also referred to as UDP/IP).</td>
</tr>
</tbody>
</table>

vi
Chapter 1
Introduction

1.1 Intensive care

A prominent feature of modern hospitals is a dedicated intensive care unit (abbreviated as ICU), in which patients with serious organ malfunction can be monitored and treated. Typically, the occupants of this unit are patients who have recently undergone major surgery, patients being treated for severely inflamed or otherwise damaged tissues, and the comatose. The intensive care personnel's task is to maintain the patient's metabolism in adequate condition until the organs concerned have stabilised sufficiently to resume their normal function as nearly as possible.

Technology assists intensive care staff members in two aspects of their task. First, database systems store clinical data such as patient information and drug prescriptions. In an intensive care environment, such database systems are typically referred to as Patient Data Management Systems (PDMS). Second, several sensors and measuring devices, integrated in a so-called patient monitor, monitor the patient's vital signs such as heart rate and blood pressure. All major patient monitoring equipment manufacturers also sell PDMS systems, conceived as extensions to the manufacturer's monitoring equipment.

In the Catharina Hospital in Eindhoven a patient data management system called ICIS (Intensive Care Information System) is used. This system is not a major monitoring equipment manufacturer's product but was developed by a local software company in collaboration with the intensive care staff members. ICIS started its development as a database for storing drug prescriptions and patient reports, to assist communication between intensive care staff members. It matured into a full intensive care information system as users provided more suggestions for possible extensions.

ICIS is however not connected to the patient monitoring system installed in the Catharina Hospital's ICU. A summary of parameters such as heart rate or blood pressure is transcribed onto a paper form from the monitoring system's screen and subsequently entered manually into the PDMS.

1.2 Problem statement

1.2.1 Data Acquisition

At the end of 1994 the advantages of coupling the intensive care's patient monitoring system to ICIS became obvious. At that time I was looking for a design assignment for a two-year designer's course at the Stan Ackermans Institute at Eindhoven University of Technology. Through dr. Blom of the Medical Electrical Engineering section of the department of electrical engineering of the University I secured an introduction to dr. Korsten, the chief
intensivist of the Catharina Hospital’s intensive care unit. The idea was to develop a system that would allow the collection of data from the intensive care’s monitoring system and using those data in implementing a *critiquing* system for tracking the condition of patients and warning against conflicting therapeutic actions. Additionally this would allow ICIS to access the monitoring data directly, making the manual transcription of the data unnecessary.

However, during the initial phases of the project it became clear that accessing the monitoring data and making it available to other software was more complex than initially assumed. It was therefore decided to limit the assignment to designing (and preferably also implementing) a *data acquisition system* that would allow computer applications to access the monitoring data. The implementation of a critiquing system became a separate PhD project, to be performed by P.A. de Clercq (see [de Clercq et al, 99a] and [de Clercq et al, 99b] for some results of that project).

The requirements of that data acquisition system are threefold.

1. The data acquisition system should *collect* data that is measured by the patient data monitors in the ICU of the Catharina Hospital.
2. The data acquisition system should make those data *available for applications running on other computers* in the ICU’s computer network, such a ICIS.
3. The data acquisition system should make those data *available for being viewed graphically* by intensive care staff.

In the two years of the designer course I designed and implemented a data acquisition system (the *HPDATA* data acquisition system) and the software to view graphs of the collected data. Chapter 3 of this thesis describes the design of that system. A more detailed description can be found in [Cluitmans, 96]. The implemented system has been in use since September 1996.

### 1.2.2 Data analysis

ICU staff using the graph viewer of the HPDATA system got the impression that certain *patterns* in the graphs indicated symptoms that, when recognised early by an automated pattern recognition system, would assist them in the treatment of patients. For example, some occurrences of ischemia and myocardial infarction were very recognisable in the graph views as concurrent drops in ST segment values and rises in heart rate and blood pressure.

Applications that can perform automatic detection of this and similar patterns are likely a useful addition to the set of diagnostic tools available to intensive care staff.

It was suggested by the doctors using the system that I should investigate the possibilities for designing such pattern recognition applications as a continuation of my work on the data acquisition system. Designing an application that can detect one particular pattern has its uses. However, designing a generic and flexible system that can be used to recognise several patterns and in which new pattern descriptions can be added dynamically would definitely be more useful.

To design such a pattern recognition application, an accurate description of the pattern to be recognised is required. To detect such patterns in a signal, a *pattern recognition algorithm* is needed. To assist in developing such recognition algorithms, software that provides tools for ‘processing’ (analysing, filtering, manipulating, viewing, etc.) the acquired data is needed. One goal of such a *data processing toolbox application* is to support the designing, testing and evaluation of pattern recognition algorithms. The tested pattern recognition algorithm can then be implemented in a pattern recognition application. When testing such a pattern recognition algorithm one may conclude that reliably recognising that pattern is not viable; in
this case the data processing toolbox application was used to test the viability of recognising
the pattern, which is another goal of a data processing toolbox application.

I introduced the concept of a data processing toolbox application that processes medical data
from the viewpoint of using it as a tool in the design and evaluation of pattern recognition
algorithms. The application of such a toolbox is not limited to support for pattern recognition
tasks. Such a toolbox can be used design of other ‘real-time’ data processing tasks. It also is
useful outside the field of algorithm design and validation. An example of another application
is using the toolbox for testing medical research hypotheses.

It is worth noting that the two kinds of applications mentioned above (pattern recognition
applications and data processing toolbox software) share several tasks. Both application types
need to access patient data and apply similar data processing algorithms. There are obvious
differences as well: for example, pattern recognition software typically processes a
continuous stream of data (it is a real-time application) while a data processing toolbox
application typically processes batches of data, applying a candidate pattern recognition
algorithm to a batch of data to evaluate that algorithm (it is an offline application). The
distinction in different application types will be discussed in section 2.2. Despite such
differences, designers of both application types may benefit from having access to a common
data processing software core that provides data access and data processing functionality.

1.2.3 DataWand and COPDA

I continued the two-year designer’s course for two more years to design such a software core
(which I have named COPDA) and to use that core in the design of a data processing
toolbox application (which I have named DataWand). Both have their role in the context of
pattern recognition applications: DataWand is needed to validate candidate pattern
recognition algorithms, while COPDA provides a base for building pattern recognition
applications. The scope of both sub-projects is wider than just support for pattern recognition
application designers though. DataWand can be used as part of any research based on the
medical data acquired with the HPDATA data acquisition system I implemented during my
two-year designer’s course. Actually it is not even limited to this data, and has been in use for
some time in a project involving data acquired using a different data acquisition system (see
[IBIS]). COPDA has already proven its use for other purposes than being a base for pattern
recognition applications, as the DataWand program itself was built using COPDA.

1.2.3.1 COPDA requirements

The requirements for the data processing software core (COPDA) are as follows. These
requirements will be evaluated in section 6.2.2.

1. It should provide designers of various kinds of data processing applications with a set of
reusable software building blocks (the core) for performing typical data processing tasks.
In the rest of this thesis, any programs that use COPDA as their data processing core are
referred to as COPDA programs.

---

1 I intentionally use the rather vague term software core here, leaving open the question what form it takes. Some of the
options may be: a function library, an object library, a component library, a single component, an application skeleton.
Most of these options may be implemented either statically linked (the core becomes part of the application) or
dynamically linked (the core is a separate DLL).

2 COPDA is the acronym for Core components for Patient Data Analysis
2. A mechanism should be defined to extend the core with new building blocks. As the individual building blocks in the core (including future extensions) probably depend on each other, such extensibility has to be taken into account during the design phase.

3. **The core should be usable for both data processing applications using batches of data (files) as input (non-real-time data processing), as well as applications that use continuous streams of data as input (real-time data processing).**

4. The core should be able to use the features peculiar to the data acquired with the HPDATA system (support for non-numerical data, including support for ‘invalid’ values).

5. The core should not be restricted to processing data in a particular format, but data processing application designers (users of the core) should be free to use whatever data format they like to use.

6. COPDA should not put restrictions that are not relevant to its functionality on COPDA applications. This implies that it should not put unnecessary restrictions on any COPDA-based application’s user interface.

7. The core should provide features that allow data processing algorithms developed for one COPDA application to be usable for other COPDA applications with minimal or no adaptations.

### 1.2.3.2 DataWand requirements

The requirements for the **data processing toolbox program** (DataWand) are as follows. These requirements will be evaluated in section 6.3.

1. DataWand should provide medical researchers with an environment to develop data processing algorithms for and test hypotheses related to the data acquired using the HPDATA system.

2. DataWand should provide a prototyping environment for developing data processing algorithms for use in real-time applications built using COPDA.

3. DataWand should provide a test environment for COPDA itself and for extensions to COPDA.

### 1.2.4 Design background

I designed COPDA and DataWand as a designer's PhD project at Eindhoven University of Technology. A designer's PhD is a post-graduate doctorate program intended for design course graduates. The primary goal of a designer's PhD project is to prove one's designing capabilities. When comparing designer's PhD with the more conventional researcher’s PhD, a designer's PhD project puts more emphasis on synthesis while a researcher's PhD project puts more emphasis on analysis. Of course, despite this difference in emphasis, analysis is an important aspect of design. Both types of PhD projects rely on the PhD student's creativity, originality and the use of scientifically sound methods for success.

### 1.3 Reading guide

The following chapters are structured as follows:

Chapter 2 provides backgrounds and introductions to issues, terminology and concepts used the chapters 3, 4 and 5.
1.3 Reading guide

Chapter 3 describes the design and implementation of HPDATA, the data acquisition system that collects the patient data. It is a summary of the work I performed during my two-year designer's course, from November 1994 to November 1996.

Chapter 4 describes the design of and rationale behind the data processing software core (COPDA). This chapter uses concepts introduced in chapter 2, but can be read independently from chapter 3.

Chapter 5 describes and evaluates the use of COPDA in various applications. This includes the design and evaluation of DataWand. This chapter uses concepts introduced in chapters 2 and 4, but can be read independently from chapter 3.

Chapter 6 provides discussions of the results and conclusions for the systems designed in chapters 3, 4 and 5.
Chapter 2
Background

2.1 Introduction

The project leading to this thesis comprises two major goals.

- The first goal is to design an intensive care data acquisition system. Though designed for use in the intensive care department of the Catharina hospital in Eindhoven, the design uses principles that apply to intensive care data acquisition systems that are more general. This system shall collect the vital signs that are measured by the monitoring equipment attached to the patients and make these data available for use by applications running on PCs connected to the ICU's local area network.

- The second goal is to create a software toolbox that provides programmers with easy access to the acquired data and provides a set of standard solutions to commonly encountered problems when using this type of data.

This chapter provides background information on several of the topics encountered in the following two chapters. The topics covered are:

- Definition of terms and concepts used throughout this thesis (sections 2.2 and 2.8)
- Literature study results (sections 2.3, 2.4, 2.5 and 2.6)
- Discussions of design methodology and design philosophy (2.7 and 2.8)

2.2 Application areas

When starting the design of a system, it is necessary to investigate what the system is going to be used for, that is: what the system requirements are. This subsection investigates and classifies the kind of applications that use the acquired data and that may be built using the data processing software core. This investigation will provide guidelines to be followed when designing the core and provides guidelines for the data acquisition system as well.

2.2.1 Target applications

Currently, in many ICU departments *vital signs* data collected by patient data monitors is not recorded by a data acquisition system. Doctors and nurses use the data provided on the monitor screen directly in the patient health assessment. Those monitors provide alarm limit systems that alert ICU staff when any measured parameter reaches a value that is possibly dangerous to the patient. In recent years it has become clear that computer aided analysis of the data collected by the monitors can provide extra information that aids ICU staff in their task. This section presents a classification of computer applications for this kind of data processing.
Vital signs measured in an ICU department can be used in two distinct application classes: *online* and *offline*. *Online* applications process data of patients currently residing in the ICU. *Offline* applications are used to analyse the vital signs of patients that no longer reside in the ICU (using archived data).

![Data processing in the ICU](image)

*Offline* applications are not used for the benefit of a single patient but they are research tools to study cases to derive or refine medical knowledge, benefiting future patients. A typical use of an offline application is the derivation of the knowledge that is required to build a real-time vital signs processing system. Another use is evaluation of real-time applications and the algorithms used in them.

Online applications can be further subdivided in two groups: *real-time systems* and *health-assessment systems*.

*Real-time systems* process vital signs data as soon as it is acquired. An example of a real-time system is a warning system that analyses the vital signs and provides a warning when a potentially dangerous situation occurs. Another example of a real-time vital signs processing application is a system that performs artefact detection and removal as a pre-processing step for data archival.

*Health-assessment* applications aid ICU staff in assessing a patient's health situation by providing access to and visualising recently recorded vital signs. *Patient Data Management Systems* (PDMS) can be considered to be examples of health assessment applications. A typical PDMS however has a broader scope; providing vital signs information is only a part of it. A better example of a health assessment application is an application that allows ICU staff to browse through recent vital signs measurements and to view vital signs graphs (such an application could be a part of a complete PDMS system).

The two subtypes of online vital signs processing applications are sufficiently different to justify classification of vital signs processing applications into three major groups: *real-time* applications, *health-assessment* (non-real-time online) applications and *offline* applications (figure 2.1). This classification is used extensively in this thesis.

Similar classifications have been proposed in literature. The nomenclature of these application types is not standardised, so different writers tend to use different names for similar concepts. To prevent confusion with similar, yet not exactly equivalent concepts I use my own classification as depicted in figure 2.1. In [Saranummi et al, 96] a schematic arrangement of functions in a patient monitoring environment is given that can be mapped to my classification. Their definition of *off-line* application is virtually the same as my definition. Their definition of *front-end* closely resembles my definition of real-time.
applications, except that their *front-end* also includes the actual data acquisition software. Their *back-end* is a PDMS, one of the tasks of it being the provision of health assessment functionality.

### 2.2.2 Delivering data to applications

In one aspect, health assessment applications are more like off-line applications than like real-time applications: both health assessment applications and off-line applications perform random access on the data they use, and they typically access data in batches of finite size. Real-time applications on the other hand sequentially process a continuous stream of data that conceptually is never-ending.

The natural way of health assessment applications and offline applications to access data is from random-access media, such as files or a database. Those applications use a *pull* style data access: the storage medium should provide the data when the application requests it. The storage medium acts as a 'slave' to the application. The natural way for real-time applications to access data however is a data stream, e.g. a network link. They use a *push* style data access: the application processes data as soon as the medium supplies new data. The application acts as a 'slave' to the incoming data stream.

Summarising, the data (communication) medium (symbolised by the grey arrow in figure 2.1) is different in nature for real-time applications on one hand and for health-assessment and off-line applications on the other hand. When all three types of applications are to be supported, this fact may influence the design of the data acquisition module. (I write 'may influence', because streamed data can be simulated using a combination of files and some triggering mechanism.) As will be shown in the next chapter, the data acquisition system I designed provides both file-based output (*data-pull*) as well as network-based output (*data-push*) to cater for both groups of applications.

A short comment on the term *real-time*. In some branches of science, using of the term *real-time* implies guaranteeing certain response times and certain data throughput. Often such *strict* real-time systems require specialised hardware or operating system features. In this thesis I do not use the term *real-time* in this strict sense. I use the term *real-time* to indicate any systems that process data in a *push* style: data is delivered to such systems as a continuous and theoretically infinite stream.

### 2.3 Sample rates

Selecting a good sample rate is an important issue when designing any data acquisition system. If the data are not sampled frequently enough, it is possible to loose relevant information. If data are sampled far more often then required to follow a signal, valuable network capacity or disk space is wasted when transporting or storing the data. Of course it is better to sample too fast than to sample too slow.

This subsection is about selecting a sample rate for slowly varying vital signs, such as heart rate or systolic blood pressure. As will be shown in the next chapter, sample rates for quickly varying signals, such as ECGs, are not an issue in my design.

There is not much literature on the topic of selecting sample rates for slowly varying vitals signs. It looks as if many designers of medical data acquisition systems do not give much consideration to the subject of selecting a good sample rate. The only reference I found, [Gravenstein et al, 89], analysed some of the signals measured in operating rooms. The
authors try to derive guidelines for sample rates for heart rate, systolic blood pressure, oxygen saturation and end-tidal carbon dioxide. They defined several critical situations that an anaesthetist should be able to detect quickly using these signals and performed an experiment on a laboratory dog invoking these situations while measuring these parameters at a high sample rate. Then they analysed the measured vital signs using three methods: moving a sample grid, using a frequency (Nyquist criterion) analysis and using a slope analysis.

The first method showed that setting the sample rate to five minute intervals leaves a large chance that important events are left undetected (because they last less then five minutes). A frequency analysis of the measured data and applying Nyquist's theorem showed that using a sample rate of once per minute (1/60 Hz) is sufficient to reconstruct the four signals. Analysis of the slopes showed that to detect a relevant change accurately when it occurs requires a sample rate of twice per minute (1/30 Hz) for most of the analysed changes. Recovery from hypoxemia occurred even faster: the authors calculated a sample rate of once per eleven seconds (1/11 Hz) would be required to accurately time the occurrence of this event.

The difference in results from the Nyquist method and the slope method may look unexpected at first sight. They can be explained by considering what the meaning of the results is. The Nyquist method determines the sample frequency required to reconstruct the entire signal from all recorded samples: locating a change using a Nyquist analysis uses knowledge of samples taken after the change occurs. The slope method derives a sample rate that allows detecting a slope when it occurs: it only uses samples taken during and before the occurrence of the change. The result from the slope method is relevant when designing a data acquisition system that should provide data that is to be analysed immediately (without knowledge of future samples); this applies to real-time applications. The result from the Nyquist method is relevant when designing a data acquisition system that collects data that is used later, when samples measured after an event are also available; this applies to health-assessment applications and offline applications.

The results of this analysis will be used in chapter 3 when choosing the sample rate to be used by HPDATA (see section 3.5.1 for the real-time case and section 3.6.4 for the offline case).

### 2.4 Existing acquisition systems

Medical data acquisition systems are not often described as such in literature. Researchers designing systems to acquire medical data tend to be more interested in the data that is to be acquired using their system than in the system itself. All modern patient monitoring equipment contains some kind of electrical interface that provides the measured parameters in analogue or digital form. For many researchers it is sufficient to use these built-in capabilities of monitors to hook up a computer to the monitor to acquire the data. Such a set-up does usually not justify much attention in articles. Articles that do describe data acquisition systems usually do so from the perspective of a complete PDMS system that does not only acquire vital signs data but does collect other patient information as well.

In [Imhoff, 92] a PDMS system is described consisting of several modules. An interesting aspect of this article is the analysis of the computing resources required for various aspects of data acquisition and representation. When classifying measured vital signs into two classes, slowly varying parameters (such as heart rate) and real-time waveforms (such as ECG), the authors estimate a data flow of at most 10 Mbytes per day per patient resulting from the first kind of data and a data flow of about 100 Mbytes per day per patient for the second kind of data. The two main factors involved in calculating the data volume are: number of signals (there are typically more parameters being measured) and sample rate (waveform data is
2.6 Data analysis systems

sampled at far higher rates). The important point is the difference in the amount of data resulting from these two kinds of data measured by vital signs monitors.

2.5 Interface standards

Another point observed in [Imhoff, 92] is the lack of standardisation of monitor interfaces. Several attempts have been started in the past to define a standard for communication between bedside monitoring equipment and computers (both for hardware and for software). The most promising of these standards is probably the IEEE Medical Information Bus (MIB). The hardware part of this standard is finished now, but the definition of the behaviour of the software parts is not yet complete at the time of writing of this thesis. See the MIB project's www page ([MIB]) for the latest news on MIB. Kennelly ([Kennelly et al, 97]) gives a good overview of the history of MIB, the problems with the first version and its future perspective.

Taboada ([Taboada et al I, 97], [Taboada et al II, 97]) describes a data acquisition system that uses this medical information bus. The authors do note some drawbacks of the MIB. They remark on their usage of a separate analogue connection to acquire certain parameters. I quote: "This direct link not only makes up for the MIB's lack of analogue capacity, but can also be used to obtain information if the MIB suffers some kind of failure". Clearly, fifteen years after its conception in 1982, the MIB is still not full-grown if designs using it incorporate a back-up data transport system "in case the MIB suffers some kind of failure".

There are a few important problems when using the MIB. First, most monitoring equipment currently in use does not yet support MIB directly. To use MIB now requires expensive hardware that converts signals from a monitor to the MIB standard. Secondly, using MIB requires installing cabling in ICUs. This requires the ICU to be temporarily unavailable. These reasons lead to a chicken-and-egg problem: ICUs will not use MIB unless it is standard available on all monitoring equipment, while monitoring equipment manufacturers will show reluctance in adding MIB interfaces to their equipment if those interfaces are not going to be used.

I did not consider using MIB in my data acquisition system design because at the time of designing that system (1994) MIB was not an accepted standard. Rumours were that it would never be. Nowadays (1998) it seems MIB has risen from its grave and will be accepted as a standard (not just by the committee defining the standard, but by manufacturers and users as well). For new medical data acquisition system designs I would recommend studying the status of MIB ([MIB]) and consider it as a communications standard.

2.6 Data analysis systems

2.6.1 Literature

While literature on vital signs acquisition systems is hard to find, literature on general medical data analysis systems is even harder to find. Articles on vital signs analysis tend to be articles about algorithms for a specialised subtask, not about general data analysis frameworks or toolboxes. An article related to the topic of data analysis frameworks, [Saranummi et al, 96], was already discussed in section 2.2.
2.6.2 Commercial data processing tools

Several commercial tools exist to aid in data analysis and processing. At first sight, using such a tool may seem useful when designing a data analysis software core. Listing all such existing tools is not a feasible task. I'll describe two examples of such commercial tools (MatLab and LabView) and describe a few problems that are common to these and other tools.

2.6.2.1 MatLab

MatLab is a tool that allows researchers to process numerical data and signals. Its power lies in operating on vectors and matrices (the name MatLab is derived from Matrix Laboratory). A vector can be used to represent part of a signal when each element in the vector contains one sample. MatLab provides many filters that can be applied to these vectors. Its interface is textual: the user types the formulas he wants to evaluate and MatLab performs those calculations. MatLab can also be used without this textual interface as a function library that provides a large amount of standard filtering and other data processing functionality to external programs. Another interface exists to MatLab that allows designing signal processing operations using a graphical user interface: SimuLink.

The nature of MatLab makes it suitable for batch-oriented processing where a number of samples is processed simultaneously. When I evaluated MatLab, my impression was that it is less suitable for sequential processing of data, where a small operation is performed often, such as is typical for real-time data processing.

Only much later I learned that packages exist that aid in using MatLab and SimuLink in a real-time environment (such packages are not included in MatLab but must be purchased separately). The real-time workshop package allows designing a real-time application in MatLab/SimuLink, and compile that application for use on dedicated (Digital Signal Processor) hardware. The real-time toolbox allows designing and running real-time applications in SimuLink under Windows 95/98, providing access to a range of PC data acquisition boards.

MatLab has no explicit provisions for processing non-numerical data, such as a signal comprising textual status messages, or built-in support for handling 'invalid' samples.

2.6.2.2 LabView

LabView is created by a company (National Instruments) that specialises in producing data acquisition hardware. LabView allows building a virtual instrument panel including various kinds of filters. The system is primarily intended for real-time data acquisition and processing.

LabView allows the user to connect various modules graphically to define a data acquisition system, data analysis system, or simulation. Those modules provide an interface to data acquisition hardware, various filtering functions, various storage functions, various display functions, and various input controls (such as knobs and sliders). The 'data acquisition hardware' can also be a disk-based file to allow offline data processing. The user of a LabView virtual instrument is shown a graphical interface that is used to control the data acquisition or control system.

LabView is very useful to design industrial control applications. Unlike MatLab, LabView excels at processing real-time data. Like MatLab however, LabView only processes numerical data. Another drawback of LabView is the graphical user interface that inherently
comes with each LabView application. Though customisable, it is reminiscent of an industrial control panel, which would look rather strange for a real-time medical data processing application running as a background process.

2.6.2.3 Features of commercial tools

The two commercial tools for data acquisition, processing and analysis I have investigated, have several drawbacks. Extrapolating these drawbacks to other similar systems is dangerous, but I am confident that drawbacks of these systems are common to other such systems as well.

One drawback is the lack of support for non-numerical data. Lack of support for processing non-numerical data disallows certain potentially interesting forms of data processing. It disallows processing textual signals, such as may result from an intelligent monitor that performs a rhythm analysis on an ECG signal. Another kind of non-numerical signal are signals whose values can be either a number or a status message such as 'measurement invalid'. Such signals are useful when using and keeping track of measurement errors (instead of ignoring or circumventing them) is relevant. This is often the case when processing medical signals (section 4.7.2.3 further investigates this type of signal and the use of 'invalid' markers).

Another problem I encountered is the licensing problem. The designer of a data analysis system of course must buy a version of any tool he wants to use. However, the user of the designed system also must buy a (run-time) license for this tool. If the designer works in a commercial environment, this is usually not a real problem: the designed system is sold together with the run-time license for the used tool. In a non-profit environment however (academic or hospital), the high cost of such a license is often considered a problem.

It is often believed that using an existing tool saves system development time because many of the parts of the system (those derived from the tool) already have been designed and tested. A cause of problems when re-using an existing tool is the fact that it does not just save development time, but it also adds development time because the tool parts have to be made to fit into the complete system. The time saved by re-using existing and well tested system parts may be completely lost in the efforts required to make these parts fit the system. This effect is especially true in software design. The effect is often underestimated by managers trying to make their designers use standard components that require much effort to adapt to other parts of the system. Section 2.8.1 further investigates this effect.

These issues with existing commercial tools were some of the reasons that led me to design COPDA as a new system, instead of basing COPDA on such commercial tools (details follow in section 4.4).

2.7 Development methods

This section gives a gentle introduction to development and design methods in general. No specific design or development methods are described: there are many good books on this topic.

---

3 Note that support for non-numerical signals is not the same as allowing 'symbolic calculations', such as available in MatLab. Symbolic calculations are specialised operations such as calculating a formula for the derivative of another formula; they are not related to processing symbolic signals.
To help an engineer design a system in a structured fashion various design methods exist. Reduced to their simplest form, design methods contain the following steps:

1. **Analysis**: The designer asks the client for a list of requirements of the system.
2. **Design**: The designer makes a design for the system.
3. **Implementation**: The implementer (for small systems often the same person as the designer) implements the system.
4. **Deployment**: The client gets his system from the implementer and is happy with it, as it does exactly what he required.

In this simplified scheme, the true text of the fourth step should read: "The client gets his system from the implementer and is unhappy with it as it doesn't do what he wanted". Practice has shown the key word in this sentence is the word wanted. What the user brought as requirements in the first step quite often does not completely describe what he actually wants (or the user doesn’t know what he wants). Often the requirements are incomplete. Often their interpretation is ambiguous. An additional source of deviation from the original idea occurs when an implementer interprets the design in his own way if the design specification is ambiguous. The primary goal of true design methods is to make sure that the client gets what he wants.

In general, these methods take extra time since the general steps above have to be iterated several times in order to get a better quality of the design.

In *Structured Analysis and Design* (see for instance [Stevens et al]), the design consists of two models, the essential model and the implementation model, which are designed and refined simultaneously using reiterations of the basic design steps above; the essential model leads to the final description of the requirements, while the implementation model leads to the final design.

In *Rapid Prototyping* the goal is to provide the client with a prototype implementing (parts of) the requirements as soon as possible. This allows the client to evaluate consequences of his choices early and to refine his requirements and add missing specifications. Rapid prototyping is especially useful when specifications are very fuzzy, e.g. when designing user interfaces.

Some development methods are intimately related to *project management methods*. Project management methods are used to guide the aspects of the management of the design process that are not related to the actual content of the design. These aspects include various resources such as time (deadlines), money and manpower.

The goals of a project manager sometimes conflict with the goals of a designer. As an example, consider the design of a software application with a graphical user interface. At the deadline for delivering the application, it can perform all tasks it should be able to, but some tasks can only be performed in a clumsy way. The user interface could be improved to make these tasks more fluent, but it would take another month before the system would be ready for deployment. The goal of the project management method is to deliver the application at the deadline. The focus of most design methods is to deliver the best application, which would take another month.

One of the tasks of a designer is to balance those goals. In software design it is important to know that *software can always be improved*. This implies that without deadlines or cost limits a software project will never be finished. To use the words used in the designer course I followed at the start of my project, "the biggest enemy of a good design is a better design": striving for a better design may take more time (and other resources) than can be justified by the improvement or the client is willing to pay for.
In my own experience as a designer I have found that this balance between the result of an improvement and the cost of arriving at that improvement is important. I encountered the following situation several times. During the design process many sub-problems have to be solved. Any design method will prescribe that to find a good solution to a (sub-)problem, the designer should evaluate several alternative solutions and choose the best solution. This automatically ensures that when the designer is asked why he chose a certain solution over another one, he can point to the results of his evaluation of the alternatives. The problem occurs when the designer starts out by evaluating the first alternative and qualifies it as a sufficient solution that is unlikely to be grossly outperformed by other solutions. Why should he go on and spend his time evaluating other alternatives, when his first attempt is functioning perfectly? To arrive at the best solution, or make plausible he already has arrived at the best solution, he has to evaluate more alternatives, even when experience or intuition tells him that chances of finding a significantly better solution are slim. Skipping such an evaluation very likely saves time, but makes explaining why the choice was made more difficult. This situation, when evaluation (or even proposal) of alternatives is unlikely to result in a better solution according to the experience or intuition of the designer, occurs quite often in software design. (The opposite situation, where there are multiple paths that may be taken, each of which needs a good evaluation before a good choice can be made, occurs frequently as well.) An experienced designer can save a lot of design time by skipping the evaluation of alternatives when his experience or intuition tells him he already has arrived at a sufficient solution. There is always the possibility that a much better solution may be found later. A careful analysis of alternative solutions reduces the chance that this happens, but does not prevent it.

2.8 Component based design

Many design methods are used in modern software engineering practice. Most software design methods focus on the reusability and maintainability of the produced code. This section introduces some of the principles behind component based design, the design methodology to be used in the design described in chapter 4.

2.8.1 Reusability of software code

Problems to be solved when designing new software (further referred to as 'software problems') are often similar to problems that have already been solved in the past. Creating reusable code, that is trying to implement a solution to a software problem in a way that can be useful as solutions to future software problems as well, always has been a major goal in software engineering.

Creating reusable (sub-) designs is a goal of many design methods, not just software design methods. Not reusing an existing solution to a design problem is usually referred to as 'trying to reinvent the wheel'. Reinventing wheels costs time in design projects - at least that is what many project management theories try to make you believe. However, reusing an existing 'wheel' (solution to a design problem) quite often costs time as well. First, to reuse an existing solution, a designer has to be aware of its existence. Next, a designer has to find out how to use it. In software design this boils down to reading the manual, which often is lacking or even is non-existent for software modules that were not explicitly designed for re-use. Quite often, the designer then discovers that the 'wheel' is a solution to a problem that is slightly different from the problem he has to solve, and has to adapt the 'wheel' to make it fit his problem. Sometimes, such an adaptation is well worth the effort. However, not seldom is re-
implementing the ‘wheel’ in a form that is more suitable to the problem at hand a better idea. Re-inventing the wheel is not necessarily bad.

As the previous paragraph shows, reusing a ‘solution to a software problem’ (reusing existing software code) puts requirements on that solution:

- The solution’s existence must be known to designers that may want to reuse it
- The solution must be well documented
- The solution must fit the context the designer wants to use it in.

The third requirement has several implications. The probability that this requirement is met seriously decreases if the solution is not self-contained but depends on solutions to other problems. In software design, an additional implication is the solution and the project in which it is to be used must be software-compatible, often implying they must be implemented in the same programming language.

2.8.2 Object oriented design

A popular solution in software design to improve reusability of code is object oriented design. Object oriented design tries to improve reusability of software by partitioning a design in more or less self-contained objects. A complete introduction to object oriented design is outside the scope of this text.

As basic knowledge of object oriented design is crucial to understand chapter 4, I will introduce some of concepts used in object oriented design here: object, method, class, instantiating, overriding, abstract methods, pure abstract classes. Eventually I will introduce the concepts of interface and component, that will play an important role in chapter 4.

2.8.2.1 Basic OOP concepts

Any text describing OOP (object oriented programming) will heavily use the three terms object, class, and method.

An object is a software entity, a ‘thing’ that exists in the computer’s memory. An object stores data and provides a set of functions to access, process or modify that data. In essence an object is like a record (or struct) in a conventional programming language (such as C or Pascal), that in addition to having data fields also has one or more fields that are functions (more precisely: function pointers). These functions, that are part of the object and operate on the object, are called the methods of the object. Using objects allows a programmer to keep data structures and functions operating on those data structures together. This greatly improves opportunities for code reuse.

The definition of what data fields and what methods are present in an object is a type definition. In object oriented programming languages such as C++ and Object Pascal, such a type definition is similar to the definition of an ordinary record (or struct) type. However, in addition to normal data fields it includes the definitions of the methods as well. These take the form of function prototype definitions appearing inside the ‘record’ definition. The resulting ‘record’ data type is called a class. An object of such a data type is called an instance of that class, and creating an object of a class is called instantiating a class.

In OOP, a programmer can assume two different roles: the programmer who programs an object and a programmer who uses an object (these ‘two’ programmers need not be different persons, but can be different roles of the same programmer). Both programmers work
2.8 Component based design

together in defining what features are to be provided by a new object\(^4\). Ideally, once this
definition is complete, both programmers can work independently. For any non-trivial type of
object it is advisable for the 'object user' not to access the data fields directly, but only use
the methods to operate on the object's data. This gives the object programmer freedom in
how he implements the object and often allows changing the object implementation without
having to change the code that uses the object. Most object oriented programming languages
provide a mechanism that allows the object programmer to indicate which methods the object
user is supposed to use to access the object and which methods and data fields are to be
considered 'private' to the object implementer.

2.8.2.2 Polymorphism and the derivation of classes

One of the features provided by object-oriented languages is that new object classes can be
declared as an extension of an existing class. The new class is said to be derived from a base
class (or parent class). In this case new data fields and new methods are added to the ones
already existing in the base class.

![Example class hierarchy](image)

*figure 2.2 Example class hierarchy.*

To introduce the terminology used when describing the relations between derived classes,
consider the situation where classes named Z1 and Z2 are derived from a class named Y
which in turn was derived from a class named X (figure 2.2). Both X and Y are referred to as
base classes of classes Z1 and Z2. The parent class of class Z1 is class Y, the parent class
of class Y is class X. Class Z1 is a child class of class Y. Class X is the grandparent of class Z2.
The fact that class Y is derived from class X means that class Y defines all data fields and
methods that class X defines, plus some more.

Objects of the derived class can often be treated as if they were objects of a base class.
Because class Y is derived from class X in the example above, any method (or data field)
declared by class X is also defined by class Y. For example, if class X defines a method named
A, class Y also defines a method named A. A function that expects to be given an instance of
class X as one of its arguments can be given an instance of class Y (or Z1 or Z2) instead. That
function can call any of class X's methods on that object.

When deriving a new class it is possible to replace the implementation of a method of the
base class with a new implementation in the derived class. When an object of the derived
class is used as if it were an object of the base class, it now uses the new implementation of
the method. This mechanism is called overriding a method. For example (referring to the

---

\(^4\) In some circumstances the definition of the features of an object is solely at the discretion of the object programmer - the
object user has to live with whatever the object programmer came up with. This situation may occur in the case of object
re-use, where the object already has been implemented before the object user has thought of using it.
example above), class Z2 can override the definition of method A it inherited from class X. When a function's prototype declares its one argument should be an instance of class X, that function can call the A method of that object. When the caller of that function passes a Z2 object as the argument, that function actually calls the A method implemented by the Z2 class instead of the A method implemented by the X class. The power of this mechanism lies in the fact that the function doesn’t need to ‘know’ that the Z2 class exists at all, it only needs to know that the X class defines a method named A. Overriding methods enables the use of the OOP concept of polymorphism: different objects can be treated as if they have the same type, but still exhibit different behaviour.

As another example (figure 2.3), consider a class Animal that is used to describe animals. Instances of this class describe individual animals. Assume the Animal class defines a method CanFly that returns true if the animal can fly and returns false when the animal cannot fly. As most animals in the world cannot fly, the implementation of CanFly in the Animal class always returns false. To describe birds, a subclass Bird of the Animal class is derived (Animal is a base class of Bird). As birds typically can fly, the bird class overrides the CanFly method of the Animal class. The implementation of CanFly in the Bird class always returns true. To describe an ostrich (a flightless bird), a subclass Ostrich is derived from the Bird class (both Bird and Animal are base classes of Ostrich), which once more overrides the CanFly method, to return false. An application can treat instances of the class Ostrich as instances of the class Animal. That is: methods defined by the Animal class, such as CanFly, are available in the Ostrich class as well (though the behaviour of these methods may be different).

2.8.2.3 Abstract methods and classes
The polymorphism concept proves to be the key to many of the advantages provided by object oriented programming. Practice has shown that when structuring an application it is useful to have some classes that are only used as base classes for other classes. Objects of any class derived from such a class can then be treated as being of this base class. Such classes will often have methods that exist solely to be overridden in subclasses. Such a method does not need an implementation in the base class. Object oriented programming languages such as C++, Java, or Object Pascal provide a mechanism that allows omitting the implementation of those methods in the base class. Such a method without an implementation is commonly called an abstract (or pure virtual) method. Classes with one or more abstract methods are called abstract classes. Classes that contain nothing but abstract methods are called pure.

---

5 The reader should be aware that the adjectives abstract and pure virtual to the nouns class and method are not used consistently across literature, and my definitions may differ slightly from what the reader is familiar with.
abstract classes. As abstract methods miss an implementation, it is not possible to instantiate objects of abstract classes (because abstract methods cannot be executed). Objects of classes that are derived from an abstract class can however be used as if they were objects of the abstract class.

To extend the previous example: in general not much can be said about the flying abilities of animals, unless more is known about what particular animal it is. Being able to ask the question 'can this animal fly?' is useful when discussing animals. So, it is useful to define a CanFly method in the Animal class. But implementing it and let it return either true or false is less useful, as the answer will often be wrong. Defining the CanFly method of the Animal class as an abstract method reflects this line of thought. Doing so has a side-effect: the Animal class becomes an abstract class, and hence the Animal class cannot be instantiated. A subclass of Animal that overrides the CanFly class (e.g. the Ostrich class) can be instantiated though (provided the Animal class does not define other abstract methods). Instances of this subclass can then be treated as instances of the Animal class.

The fact that the Animal class cannot be instantiated directly can be interpreted as: there can be instances of Ostrich, Cat, Fly, Horse, etc, all of them being Animals, but there cannot be objects that are just 'Animals' and nothing else.

2.8.2.4 Multiple inheritance

The first generation of object oriented programming languages usually only supported single inheritance: each class can have only one parent class (as is the case with all examples above). It is sometimes useful to have multiple base classes for a derived class: this allows an object to be treated as one object out of a set of unrelated classes. Implementing such a multiple inheritance mechanism comes at a cost though. Without digressing too much on this topic (see standard OOP literature for details), I name some of the problems (some of which can be solved by extending the syntax of the OOP language):

- The parent classes may be incompatible (for instance when different parent classes define a method of the same name with a different prototype).
- In the case where some of the parent classes themselves are inherited from a common base class a new problem arises: do these classes share the same base object or do they have separate base objects. In the latter case, the derived class has a case of 'schizophrenia': when asked to act as that common base class, it doesn’t know which of the two to use.
- Multiple base classes may define a method of the same name and the same prototype. When the derived class wants to override one of those methods, it is ambiguous which method actually will be overridden.

2.8.2.5 Interfaces

There is a very elegant solution that solves most of the problems associated with multiple inheritance and en passant provides some extra benefits as well. The basic idea is to put some extra restrictions on the classes that can be used in multiple inheritance and to put restrictions on the use of objects of the derived class.

- All but one of the parent classes of the new class must be pure abstract classes.
- Once instantiated, each object of the derived class can only be accessed as being one of these pure abstract classes (not as the one non-abstract parent class or as the new class itself). New methods defined by the derived class itself are invisible to the user of the instance. Such new methods can only be used as internal methods, that are called by the implementation of the methods that override methods in one of the abstract parent classes.
This enforces that software using the object does not use any knowledge it may have about the details of the implementation of the object. This in turn makes software maintenance easier, as it allows these details to be changed without changing the use of the object.

When talking about this paradigm the following terms are used extensively. The pure abstract parent classes are called interfaces. Objects following this paradigm are called components. When a certain interface is a parent class of a component class, that component is said to support that interface.

The example of modelling animals in an object-oriented is approached differently when using interfaces. Assume in this case there is a Nightingale component, which supports the Animal and FlyingObject interfaces. Via the Animal interface the Nightingale component exposes information about the animal-specific parts of the nightingale, such as the favourite food or the life span of the animal. Via the FlyingObject interface the Nightingale component exposes information on the flying behaviour of the nightingale, such as average cruising speed and average flying altitude. There is absolutely no relationship between the Animal and FlyingObject interfaces. An Aeroplane component can support the FlyingObject as well to provide information on its flying characteristics. Components that support the Animal interface have no obligation to support the FlyingObject interface. Those components can indicate that they do not represent flying animals by not supporting the FlyingObject interface.

This interface based system is very useful when object oriented programming is combined with dynamic linking. An application can access a component that is defined in a DLL 6 once it knows how to instantiate the component and how to access the interface it wants to use. When the DLL is replaced with a newer version, the application still works as expected under the following restrictions: the component is still instantiated in the same manner as before, the interface used by the application has not changed, and the implementation of the interface’s methods (overridden by the component class) is compatible with its previous behaviour. The new component version can implement new functionality by implementing new interfaces. These new interfaces will not influence the ‘old’ application, because that application doesn’t know about their existence and hence cannot access them. When the new component version wants to augment an existing interface with new methods it should not change the existing interface but it should add a new interface to the component. This mechanism that ensures that ‘old’ applications can use ‘new’ components only works when the application only accesses interfaces of the component, and never accesses the component class itself. This is because the component class necessarily changes whenever changes to the component are made. Because the component class is not an abstract class this means that to use the modified component class the application using it must be recompiled.

This system of combining interface based design with dynamic linking provides two interesting advantages. First, a newer version of a DLL can provide new features to old applications when the implementation of an existing interface of the component is modernised. Only the implementation of the methods can be changed; changing the interface itself by adding new methods is not possible without causing compatibility problems. Second, an updated component can provide completely new features to new applications while still being compatible with old applications. For these reasons components implemented in DLLs

---

* A DLL is a Dynamically Linkable Library. A DLL is a pseudo-program: it can provide most functionality that can be provided by a normal program, though it cannot run stand-alone. A DLL can implement functions and make those available for calling by programs or other DLLs. This is called exporting a function. Programs can use DLLs by ‘loading’ the DLL and then use the functions implemented in the DLL. This is called importing a function from a DLL.
are very heavily used in Microsoft’s operating systems: it allows applications to use operating system features provided by these components while Microsoft can still improve these components without fear of breaking existing applications.

2.8.3 The Component Object Model

Microsoft’s interface based system is called the Component Object Model (abbreviated as COM). It has a few improvements on the basic interface based design system introduced in the previous section. COM concepts will be used heavily in chapter 4. This section will briefly introduce some of the key features and properties of COM; for a complete introduction I refer to the book Inside COM [Inside COM].

In COM there are some important further restrictions to the interface based design model described in the previous paragraph:

- Each interface must be derived from a ‘root interface’ (called IUnknown).
- Each interface must have a unique identification code.
- COM Components are reference counted.
- The details of how a compiler should implement interfaces are standardised at the binary level. This allows applications to use components from DLLs implemented using a different compiler or implemented using a different programming language.

Interfaces are pure abstract classes. Just like ordinary classes, a pure abstract class can be derived from another class, as long as that other class is pure abstract as well. So, an interface can be inherited from another interface. For implementation reasons, this interface inheritance is restricted to single inheritance.

One of the problems of the interface based design model described above is that once an application has acquired an interface of a component, there is no standard way to access other interfaces supported by the component. In COM this problem is solved using the unique names that each component must have and the QueryInterface method, which is one of the three methods of the IUnknown interface. This method checks whether the component supports the interface whose unique identification code is given as one of the method’s arguments. If the component supports the interface with this identification code, the method returns this interface of the component. As each COM interface is derived from IUnknown, each COM interface provides this capability to access other interfaces supported by the component.

One of the frequently occurring problems in programming is to keep track of what blocks of memory are in use by an application and when these blocks can be freed. Failing to free a block of memory that is no longer in use causes memory leaks. Worse, when a block of memory is accessed after it has been freed, all kinds of difficult to track bugs can occur.

Several systems to aid keeping track of what objects are in use and what objects can be freed have been developed in the history of computer science. COM requires objects to use a reference counting mechanism: each component keeps track of the number of times other objects (or variables) reference it. When this count reaches zero, the component destroys itself (releasing any memory or other resources it used). To maintain this reference count, a component needs the help of the software that uses the component. Each time software stores

---

7 In COM, the convention for interface names is to be a descriptive name prefixed with a capital I.

8 This code is a 128-bit number, called a IID (Interface Identifier). IIDs are an example of GUIDs (Globally Unique Identifiers), which are 128-bit numbers that are used as identification codes for various entities in COM.
a reference to the component, it should call that component’s `AddRef` method (`AddRef` is one of the `IUnknown` methods). When the reference is no longer in use, the software should call the `Release` method. Software ‘stores a reference’ to a component when it assigns an interface to a variable (e.g. when creating a component or when copying another variable). That reference is ‘no longer in use’ when the variable goes out of scope or when the data structure containing the variable is destroyed.

Reference counting makes a number of programming tasks easier. However, it has one major drawback: circular references. Consider the following situation: there are two components, named A and B. Component A needs to access component B, so it stores a reference to component B. When the last reference to component A is removed, it will destroy itself and thereby release its reference to component B. If this was the last reference to component B, that component will self-destruct as well. A problem occurs when component B needs to access component A as well. When component B stores a reference to component A, component A will only be destroyed when component B is destroyed and vice versa. In other words: they will never be destroyed at all and a memory/resource leak is born. A few standard solutions exist to solve this problem.

One solution is to let both components store a reference to each other but let component B ‘forget’ to call the `AddRef` call when it stores the reference to component A. The ‘forgotten’ `AddRef` allows component A to be freed without component B `Releasing` its reference to A first. For correct behaviour, a mechanism should exist that prevents component B from calling the component A when component A has been freed.

Another solution is to let component B not store a reference to component A at all, but provide a reference to component A as an argument to each method of component B that needs to access component A. I used this principle in the system that will be described in chapter 4: in that system a component exists (the COPDA engine) that stores a reference to most other components in the system. Methods of those other components that need to access the engine component have an argument called `context` that is an interface of the engine.
Chapter 3
The HPDATA System

3.1 Introduction

In the intensive care department of the Catharina Hospital in Eindhoven two systems are used to keep track of patient information. First, the patient data management system (PDMS) called ICIS keeps track of general patient information (such as patient name and date of birth), and treatment information (such as prescribed drugs). Second, a system of patient monitors (Hewlett-Packard Merlin monitors) measures patient vital signs and displays them.

Back in 1994 those two systems were not coupled: measured vital signs had to be transferred from the monitors and entered manually into the PDMS. The reason for this was the lack of a system to make the vital signs measured by the monitors available electronically. Therefore there was a need to design a data acquisition system to make the vital signs measured by the Hewlett-Packard monitors available for processing by PCs.

This chapter describes the results of the design process for such a system: the HPDATA data acquisition system. I designed this system in the context of my two-year designer course from November 1994 to November 1996 (the technical choices described in this chapter mostly reflect the technology of that era). A more complete description may be found in my designer course report [Cluitmans, 96]. The data acquired with this system inspired the work that will be described in chapters 4 and 5, but those chapters can be read separately from the current chapter.

3.2 Designing the system

3.2.1 Location of use

The first step in designing a data acquisition system is to identify where the system is to be used. From that follows what hardware can be assumed to be available for use in the system. As already stated the system is intended for use in the intensive care department of the Catharina hospital. The question is whether the system should be usable in intensive care departments of other hospitals as well. In other words, there is a choice between a specific approach, in which the system is designed for the Catharina hospital only, and a generic approach, in which the system should be usable for as many hospitals as possible.

On one hand, the generic approach has the obvious advantage that it is commercially more interesting: the system can be sold to other hospitals than the Catharina hospital. However, it requires the near impossible task of defining a generic interface to patient vital signs monitors of various manufacturers. Several groups of researchers have tried to define such a generic interface standard in the last two decades. Up to now none of these standards has been widely accepted (see also section 2.5). The standards that are flexible enough to be fit for the task
(HL7 [HL7], IEEE Medical Bus [MIB]) have the problem of being too flexible: often one piece of data can be represented in more than one way, which makes interpreting this data difficult.

An additional problem is that manufacturers of monitoring equipment are ambivalent with regards to these attempts towards standardisation. Existence of such a standard would improve interoperability of hardware of different manufacturers. Currently, once a hospital chooses for patient monitors of a certain manufacturer, it is stuck with that manufacturer. Most equipment from other manufacturers will not co-operate with these monitors. This is an advantage for the manufacturer who supplied the monitors, and a disadvantage for other manufacturers. Emergence of a well-accepted standard can change this situation. It seems that manufacturers try to get involved in most of the groups that are developing such interoperability standards, both to closely monitor development of these standards as well as to make sure all features of their equipment are supported by them.

On the other hand, the specific approach has the advantage that it is clear what hardware infrastructure is available. Custom communication protocols can be designed for the specific features of this hardware. The drawback is that the system only works in intensive care departments with the same hardware infrastructure as the intensive care the system was designed for. However, the experience gained in designing such a specific system may be applicable to more generic designs as well.

Because of the problems associated with the generic approach, I chose to use the specific approach for this project: the system was designed specifically to target the Hewlett-Packard Merlin monitors.

### 3.2.2 Network infrastructure

The next step in the design is to decide how to access the patient monitors. The Hewlett-Packard monitors support two ways of accessing the data they gather. Each monitor contains a standard serial (RS232) port that can be connected to the serial port of a PC. Additionally all monitors are connected using a proprietary Hewlett-Packard network called the Serial Data Network (usually referenced by its acronym SDN). Hewlett-Packard sells a plug-in board that allows connecting a PC to the SDN. A PC equipped with such a board can read the data gathered by any monitor connected to the SDN. To access the data in the monitors a choice has to be made between using either the serial port or the SDN.

When using the serial port of the monitors there are two ways to collect the data at one point. One may choose to connect all monitors to one PC equipped with a large number of serial ports. Alternatively, one may choose to use a separate PC for each monitor. Those PCs can be interconnected using a standard (Ethernet) network. As there are about 20 monitors in the intensive care department, this is a choice between either a large number of cables or a large number of PCs. For maintenance and cost reasons the latter doesn't seem to be a good solution. Both of these solutions involve installing a new data infrastructure in the intensive care, which in itself is a problem, as the normal activities of the intensive care preferably should not be disturbed by installing a new data acquisition system. In this respect using the SDN has a strong advantage: the required network infrastructure is already present; to complete the infrastructure requires only connecting a PC containing the adapter board to the SDN.

Using the serial ports of the monitors has one big advantage over using the SDN. Most of the medical equipment used in the intensive care has a serial port. Using a serial port as the basic source for data for the data acquisition system allows coupling new equipment to the system...
at a later stage, decreasing the dependency of the system on Hewlett-Packard's products. Choosing SDN as the means to access the monitor data restricts the data sources to Hewlett-Packard compatible\(^9\) products.

A topic that requires special consideration in all applications of electronics in a medical environment is electrical safety. There are strict rules about the isolation requirements of electrical equipment that is connected to a patient. A standard RS232 serial port on a PC is typically insufficiently isolated to comply to these rules. This implies that electrical safety has a large impact on the hardware used in a serial port based approach. On the other hand, SDN already includes the required electrical isolation hardware (in the SDN connectors), which makes electrical safety less of a problem.

Because of the advantages the SDN approach provides over the serial port approach (cabling already present, no special electrical safety considerations required), I chose to use the SDN as the infrastructure used in the data acquisition system. The only reasonable way to access the SDN by a PC is using the Hewlett-Packard SDN adapter PC plug-in board.

Before continuing with the next step of the design process it is necessary to know more about SDN, the SDN adapter board and the patient data the board can retrieve from the SDN. The next section provides a background on these topics.

### 3.3 SDN and the SDN adapter

#### 3.3.1 Overview of SDN

A diagram of a SDN network is shown in figure 3.1. The SDN system connects up to 24 patient-monitoring stations (bedside monitors) to one or more patient information centres. The bedside monitors provide the vital signs they measure to the SDN network. The patient information centres provide auxiliary patient data (e.g. the patient's name and identification code) to the network.

Physically, a bedside monitor is a device that is mounted close to a bed incorporating a screen displaying the parameters currently being measured and a keyboard to operate it. Intensive care technicians can install up to 16 measuring modules in a bedside monitor, each of which can supply one or more types of data to the monitor. One example is the ECG module, which measures ECG waves and additionally calculates the heart rate. Intensive care staff (nurses and doctors) can define alarm limits for the measured parameters using the monitor's keyboard. When necessary they can (temporarily) disable certain alarms.

A patient information centre comprises a large screen, a keyboard and an embedded computer. It displays data for a small group of patients, usually all patients in a ward. The patient information centres are also used for entering and storing generic data such as the patient's name and identification code. SDN allows the patient information centres to display the data measured by the bedside monitors and allows the bedside monitors to access the patient's name from the information centre. The SDN also allows any bedside monitor to display data measured by another bedside monitor linked to the network.

---

\(^9\) This is not necessarily the same as Hewlett-Packard products. For instance, an adapter exists for the Siemens servoventilators used in the Catharina hospital that allows providing the parameters measured by this Siemens equipment on the SDN.
This section introduces a few important concepts that will be used frequently throughout the rest of this thesis.

The data provided by the bedside monitors for a vital sign can be of one of two classes: it can be provided as a parameter or as a waveform. These are the terms that are used in Hewlett-Packard's documentation ([HP78360A], [HP78360B]) and I will use these terms throughout this chapter. Data is transmitted on the SDN as a parameter in case of vital signs that are measured at a low frequency (one Hz or slower), such as heart rate and systolic blood pressure. Waveforms are used for vital signs that are measured at a higher frequency (usually a few hundred Hz), such as ECG and blood pressure curves. A parameter is transmitted value by value. A waveform is transmitted via the SDN as a block of 128, 256 or 512 samples.

There is a difference between the way parameters and waveforms are used in medical practice: typically parameters are treated as numbers ('the patient has a heart rate of 72'), and of waveforms as graphs ('there is an extra peak in this ECG').

---

When an important concept is defined in this thesis, it is printed in a bold italic font. An entry for this term will be present in the index of this thesis, referring to this definition.
The parameters or waveforms are identified by their Medical Function Code (MFC). MFCs are numbers in the range between 1 and 63, each of which uniquely defines a parameter or waveform. For instance, the MFC for heart rate is 10. Not all of the possible 63 MFCs have yet been defined by HP, so HP can assign new MFCs when new measuring devices become available.

Each parameter and waveform can have one of three possible parameter types: numeric, stored or textual. The type of the parameter depends on the nature of the parameter. Most parameters and all waveforms are of the numeric type: their value comprises one numeric value (varying in time). Some numeric parameters - blood pressures - actually comprise three numbers (systolic, diastolic and mean). Apart from the numeric parameters, two other types are defined. The value of textual parameters is not a number but a text-string. An example of this type is the parameter generated by the ECG waveform analyser: its value is an empty string if an ECG signal has a normal waveform or a description of the irregularity when an abnormal waveform is detected (for example ‘arrhythmia’ or ‘VES 4’). The third type of parameter is called a stored parameter. Its contents comprise a numeric value and the timestamp (hours and minutes) at which it was measured. This type of parameter is used for measurements performed only a few times a day, such as cardiac output.

3.3.3 SDN PC interface adapter

Hewlett-Packard sells a PC plug-in board that allows data provided to the SDN by bedside monitors and patient information centres to be transferred into a personal computer. This board (called HP78360B or ‘Speedy’ by Hewlett-Packard) can only read data from the SDN; it cannot write to the SDN or modify existing data in bedside monitors or patient information centres.

The following subsections give a concise description of the hardware and the software interface of Speedy. For a more complete description, refer to [Cluitmans, 96], [HP78360A] and [HP78360B].

3.3.3.1 Speedy hardware

This subsection gives an introduction to the hardware of Speedy and the implications for its use.

Speedy is a board that fits into the ISA bus of a standard PC. It contains its own Motorola 68000 processor that controls the reception of data from the SDN. The board and its host PC communicate by reading and writing speedy messages in a part of the host PC’s memory that is set aside as shared memory.

The term speedy message is used in this section for the communication blocks that are used in the communication between Speedy and its host PC. The reader should be aware of the possible confusion with SDN messages (see section 3.3.4), that are the communication data blocks that are read from (never written to) the SDN. A speedy message may contain a SDN message, but may also convey a command from the host PC to Speedy or a status message from Speedy to the host.

The PC can easily access the shared memory because it is part of its own main memory; Speedy accesses the shared memory through the computer’s ISA bus DMA controller, which must be specially reprogrammed to allow this.

As far as the software is concerned, the shared-memory block comprises three elements: one small block of control information and two communication channels (see figure 3.2). One of
these channels (the transmitter channel) is used to send *speedy messages* from the host to Speedy and the other (receiver channel) is used to send *speedy messages* from Speedy to the host. Full details of the communication between Speedy and its host PC can be found in [HP’78360A] and [HP’78360B].

![Speedy shared memory layout](image)

**Figure 3.2** Speedy shared memory layout. Only the transmitter channel is zoomed in, the receiver buffer uses exactly the same data structures.

### 3.3.3.2 Using Speedy

Before Speedy can operate, a shared-memory buffer has to be set up. Setting up a shared-memory buffer consists of:

- allocating a block of memory to be used as shared memory and initialising the required data structures;
- setting up the host PC’s DMA controller in a special mode that allows Speedy to access the host PC’s memory; and
- telling Speedy where in the host PC’s memory it can find the shared memory block.

Allocating the memory buffer is not as simple a task as it may seem at first. The memory buffer will be accessed by Speedy via the (ISA-bus) DMA controller of the host PC. Because of the design of the ISA-bus DMA, this memory buffer can only be allocated at a few special places in the memory: the physical memory address of the start of the buffer must be a multiple of 128 Kbytes. In addition to this, Speedy also requires the buffer to be allocated within the first (physical) megabyte of the host PC’s memory.

Restricting the buffer’s start address to a multiple of 128 Kbytes and locating the buffer in the first megabyte of the computer’s memory requires special precautions. The worst problem is that these requirements involve physical rather than logical memory addresses. Many software-drivers or operating-system parts that are nowadays found in normally functioning PCs change the way memory addresses are mapped from logical addresses to physical memory locations. These tools allow accessing more than 640 Kbytes of memory under
3.3 SDN and the SDN adapter

DOS, they allow multiple programs to run at the same time or they allow the use of disk space as virtual memory. These so-called DPMI\(^{11}\) provider programs include all versions of MS-Windows, and EMM386.EXE or other memory-optimisation tools. (HPDATA was designed in 1995. At that time ‘MS-Windows’ was equivalent to ‘Windows 3.11’. The same holds however for newer versions of MS-Windows.) The memory-mapping actions performed by these tools are usually completely hidden from programmers, except when needing memory addresses that can be accessed by other devices in the PC. The problem is that most of these tools do not provide a way to reach physical memory addresses.

There are two ways to get round the DPMI provider tool problem:

1. using no DPMI provider tool at all. This implies that DOS’s 640 Kbytes memory limit applies and cannot easily be circumvented. It also implies that many of the software tools available today to make networking and database access easier cannot be used;
2. using a DPMI provider tool that allows access to physical memory addresses. In MS-Windows (both Windows 3.1 and Windows 95), low-level drivers can be written that allow access to physical memory addresses. These drivers are called VxDs. Programming such a driver is no easy task: it involves assembly language programming and a fairly deep understanding of the inner workings of Windows. Debugging VxDs is difficult: programming errors in VxDs usually crash the operating system before the cause of the error can be traced. However, using Windows as a DPMI provider tool allows the use of several existing database and network tools that simplify programming the rest of the SDN bridge software.

Before a choice can be made between these two approaches, one other feature of Speedy has to be taken into account: several PC-Speedy combinations do not work well. Most modern PCs do not support using the special mode of the ISA bus DMA controller that is required by Speedy. More specifically: most modern PCs only have an ISA bus as an appendage to their main (PCI) bus and their PCI-to-ISA bridge does not support this ISA chaining mode, as only a very few PC plug-in boards use this mode. Unfortunately, Speedy is such a plug-in board. It should be noted that when I was designing this system (1995) PCI bus PCs were not yet common and I discovered this problem only late in 1996.

Even if the PC motherboard supports the Speedy hardware, conflicts may arise with other plug-in boards or system software. I investigated option 2 above (writing a VxD) by building a prototype system that accessed Speedy in a Windows 3.1 environment. This system seemed to run fine for a while but would lock up every few days. I also built a prototype system that used option 1 (a bare DOS system). This system worked fine and never crashed. A variant of this system (described in section 3.5) has been running uninterrupted for more than a year, and was still running at the time of writing this thesis.

However, a DOS-based approach has severe limitations regarding memory use. Doing anything but actually acquiring patient data from Speedy on this machine would impair the stability of the system. The safest way to use this system in a data acquisition system is to get the acquired data out of this system onto a network as soon as possible with as little processing as necessary. Other systems on that network can then perform more memory-consuming tasks.

\(^{11}\) DPMI = ‘DOS Protected Mode Interface’
3.3.4 Logical sources

This subsection describes the software aspects of the communication between Speedy and its host PC.

Acquiring data from the SDN via Speedy requires two steps. First the software requests Speedy to start sending data of a specified type and at a specified rate. Next the host starts receiving the requested data and uses it.

For each class of data (see section 3.3.5) the host sets up a logical source in Speedy by sending it a logical source connect speedy message. This speedy message acts as a command that causes Speedy to send a speedy message with data of the desired SDN message class at regular intervals to the host\(^\text{12}\). The data in the incoming speedy message contains the SDN message that was sent by a bedside monitor or patient information centre. To avoid confusion, I will explicitly use either speedy message or SDN message when talking about messages in this subsection. After this subsection when I talk about a message I always mean an SDN message.

A few examples will give an impression of what a logical source definition may be (these examples are translated in human-readable sentences; the actual logical source definitions are binary data blocks):

- “send an SDN message containing all patient names and identifications as stored in all patient information centres in the SDN once every minute”;
- “send an SDN message containing all parameter values for bed number 4 every five seconds”; and
- “send an SDN message containing the value for ‘Heart Rate’ for the patient on bed 11 every second”.

Each logical source definition relates to a SDN message class, as will be described in section 3.3.5. For each logical source definition the beds which should provide the data are specified. A bed in this context is the combination of both the bedside monitor aside a physical bed and the patient information centre that stores the auxiliary data related to that bedside monitor.

There are a few options for the timing of the messages generated by the logical source. A logical source can generate its output once or repeatedly. When repeating, the interval at which a message is generated by the logical source must be a multiple of 1024 milliseconds.

For full information on logical sources, see HP's programmer’s documentation for Speedy ([HP78360A], [HP78360B]).

3.3.5 SDN message classes

This section typifies the kind of data that can be retrieved from the SDN network by a Speedy board. Each SDN message received by Speedy is of one SDN message class. The SDN specification (see [HP78360A] and [HP78360B] for details) defines the following 11 SDN message classes:

- **Bed Label**, which retrieves the list of bed names from a patient information center. This provides a way of mapping abstract bed numbers used in the other messages to

\(^{12}\) Actually, sending at regular intervals is only one of three possible ways of defining the Speedy-to-host PC communication. But this scheduled automatic mode of operation is the only mode I use. See the HP documentation for Speedy ([HP78360A], [HP78360B]) for more details.

30
names used for the beds by the intensive care staff (e.g. 'Room2, bed 3' for bed number '7');

- **Patient Name Identification**, which provides the name and identification code as stored in a patient information center. The patient identification code can be used to determine which patient is connected to which bedside monitor;

- **Parameter Values**, i.e. the values of all the parameters measured by a specified monitor, or the value of one selected parameter measured by the specified monitor (there is no way to select a subset of parameters);

- **Parameter Support Data**, which provide additional information about the parameters measured by a monitor. This includes: a textual representation of the value, alarm limits for a parameter, the label to be used for a parameter and value interpretation information. The value interpretation information describes how the 10-bit integer values received from a *parameter value* message are to be interpreted. It defines whether a parameter is signed or unsigned and it defines the scaling factor (1, 0.1 or 0.01) to apply;

- **Instrument Status**, which provides information on the operating status of each bedside monitor connected to SDN. This includes a flag that determines whether the bedside monitor is on-line or off-line (generated by its associated patient information center) and a list of the parameters that are measured by the monitor and those which are not;

- **Wave Values**, i.e. blocks of high-frequency parameter data. Using this message-class puts a very high load on Speedy; retrieving more than a few waveform parameters (let alone all the waveform parameters for all beds) will cause data traffic that exceeds Speedy's transport capacity. Some signals that are derived from the waveform parameters are available as normal parameters. For example, the patient monitors calculate the heart rate from the ECG waveform signals;

- **Wave Support Data**, which provide additional information on parameters that are provided as waveform;

- **Alert status**, which provides information about the alarms currently being triggered at each bedside monitor. Because of the way this message-class is handled by Speedy, requesting all *alert status* messages from all beds would exceed Speedy's data transfer capacity. Some alarm information is available in the *parameter value* and *parameter support data* messages: i.e. the alarm limits and a flag indicating whether an alarm has been triggered;

- **Alarm Text**, which provides the text for current alarms from a bedside monitor. Requesting this message class causes the same problems as requesting the *alert status* messages;

- **Inop Text**, which provides the status text for inoperative modules or sub-modules. Requesting this message class causes the same problems as requesting the *alert status* or *alarm text* messages;

- **Time of Day**, which provides a time reference. This would be very useful for providing standardised time stamps for the data measured. However, *time of day* messages are only available on the SDN when this is connected to a time source, which was not the case.
3.4 Designing the SDN data retrieval

This section continues the line of thought started in section 3.2. To continue the design of the data acquisition system several interrelated questions have to be answered. This section will answer questions about what the acquired data is going to be used for. The answer to these questions deeply influences the design of the data acquisition system.

3.4.1 Target applications

In section 3.1 the overall goal of the data acquisition system was defined as *make the vital signs measured in the intensive care unit available for processing by PCs.* Though the goal of the system is to provide data for applications that have yet to be designed, a few target application areas were defined in section 2.2:

- **Alarming/monitoring** systems. This type of application continuously monitors incoming data and can provide users with warnings if anomalies are detected.

- **Patient health assessment** applications. This type of application assists medical staff in assessing a patient's health status using recently measured data (up to a few days ago) for a patient. A typical example is an application that shows graphs of the parameters measured for a patient, allowing medical staff to analyse trends. Health assessment applications may be designed to work with data about patients that are currently in the ICU, but also to inspect data about patients that are no longer in the ICU (for instance to assist in analysing a death cause).

- **Offline** applications. I use this name for the broad class of applications that analyse a large body of patient data. The goal of such applications is to obtain new knowledge, such as finding patterns in measured parameters. This knowledge can for instance be used in monitoring applications. The primary goal of this type of application is not in specifically helping the individual patients whose data are analysed, but to gain new medical insights that may benefit all patients.

These application areas put different requirements on the way they expect data to be delivered to them: On one hand, health assessment applications and offline applications are *data-pull* based applications that expect data samples that were measured at different times to be accessible at the same time. They will require the data to be *stored* in files or in a database. On the other hand, monitoring applications usually are *data-push* based applications: they process the data that is available at a certain moment in time and then wait for the next sample to arrive. The native way for this type of application to access data is to wait for a packet of data to arrive via a *network* and then process it.

Thus the desired types of target applications would require the data to be available in two different ways: both as packets on a network and stored in files. Access via a network has the drawback of being volatile (data is sent once, and then it is gone), but an application can use data as soon as it becomes available. Access via files has the drawback that it is less suited for real-time applications, but data in a file is non-volatile. I decided to let the data acquisition system support both ways of providing data.

3.4.2 Basic system layout

The data acquisition system will basically look as depicted in figure 3.3.a: it receives data from the SDN network using a Speedy board, it produces data packets appearing on a Local
Area Network and it stores output on a storage system (data files and/or a database). Two choices exist for implementing such a system. The first is to implement the complete system in one PC (figure 3.3.b). This PC should be equipped with a Speedy board and should both send incoming data on the outgoing LAN and store them. The other option is to use two PCs (figure 3.3.c). The first PC is equipped with a Speedy board and acts as a bridge that transfers data from the SDN to the LAN. The second PC reads the data sent on the LAN by the first PC and stores it.

![Data acquisition system diagram](image)

Figure 3.3 Data acquisition layout and implementation options. The thick black lines depict the incoming (SDN) and outgoing (LAN) networks.

The obvious advantage of the first solution is that only one PC is needed for the data acquisition system. It should be able to run fairly complex software to perform the tasks of receiving SDN data, sending network packets on the LAN and maintaining the data storage. The latter task requires a decent amount of memory in the PC for successful operation. At this point it is necessary to review the notes about the use of the Speedy board that were made in section 3.3.3.2: essentially Speedy can only be used in an older model computer running plain DOS without any DPMI provider programs. This implies that memory on the PC containing Speedy is limited to 640 Kbytes, which is probably not sufficient for the three tasks. The second solution (using two PCs) circumvents this problem: the two tasks requiring access to Speedy (reading from the SDN and sending the data on the LAN) run on a different computer than the task requiring a large amount of memory (data storage). For this reason I chose to design the data acquisition system to consist of two separate parts (see also figure 3.4):

- a bridge PC, equipped with a Speedy board and an ethernet card that reads data from the SDN and copies this data to the Local Area Network connected to the ethernet card.
- an archiver PC, that receives data sent by the bridge on the LAN and stores it, for use by other PCs on the LAN or for use by programs running on the archiver PC.
3.5 The SDN bridge

3.5.1 Data retrieval

The next issue to address is what kind of data to retrieve from the SDN. The goal of the system is to make data present on the SDN as much as possible available to the outputs of the data acquisition system. To simplify both the design of the data acquisition system as well as to simplify using its output, I chose to make all data to be provided by HPDATA available at the same rate. The choice of a suitable rate will be discussed at the end of this subsection.

The amount of data that can be requested simultaneously is limited by Speedy's transport capacity (the maximum amount of data that can be transferred from the SDN to Speedy). This transport capacity is too small to request all data that is available on the SDN, so a subset of the available data has to be chosen. The amount of data transferred from the SDN to Speedy by setting up a logical source (see section 3.3.5) only depends on the message class for that logical source and on the number of beds it applies to. It does not depend on the rate at which the data is requested to be sent from Speedy to its host PC. Internally Speedy always acquires data at its maximum speed (one cycle processing all logical sources once per 1024 milliseconds). When requesting data at a slower rate, Speedy discards the data it receives from the SDN for the intermediate cycles. Requesting data at a slower rate does not allow Speedy to provide data for more logical sources.

Some of the message classes carry redundant information (Instrument status) or information for which a summary is available in other messages (a bit indicating an alarm status is available in Parameter Values, removing the primary reason for acquiring Alert status, Alarm text and Inop text messages).
Some message classes require much more of Speedy's transport capacity than others. For instance, one single *Wave Values* message, containing data for one wave source from one bed, comprises slightly more than 1 kilobyte of data. On the other hand, a *Parameter Values* message containing all parameters for all beds (usually around 200 parameters in total for a full ICU) comprises about 2 kilobyte of data. So, the transport capacity required for one single *Wave Value* is about the same as the capacity required for 100 *Parameter Values* (see also [Imhoff, 92]). Requesting more than a few *Wave Values* for one or more beds is already more than Speedy's transport capacity can handle. For this reason I decided that *Wave Values* would not be provided by the data acquisition system. This decision also makes retrieving *Wave Support Data* unnecessary.

The transport capacity required by some message classes does not depend on how much memory they actually use in Speedy, but on the amount of memory they would use in a worst case situation. This is of importance for the *Alert status*, *Alarm text* and *Inop text* message classes. For example, the amount of transport capacity required for an *Alarm Text* message for one bed is the space required to store one alarm string (20 bytes) for each parameter measured by that bed's monitor (up to 16), even if there are no alarms active. This effect is so strong that requesting the *Alarm Text* of all monitors surpasses Speedy's transport capacity, despite the fact that that message's data usually is empty. For this reason it is infeasible to retrieve messages of these three classes.

The *Time of Day* message class would be a useful message to retrieve for purposes of standardising the time references (a PC's clock tends to drift surprisingly fast when the PC is permanently powered-up). However, experiments showed that this message was not available on the SDN of the Catharina Hospital.

What is left are *Bed Label* messages, *Patient Name Identification* messages, *Parameter Value* messages and *Parameter Support Data* messages. Requesting all data available for all beds for these four message classes does not overflow Speedy's transport capacity. To make available as much SDN data as technically possible, I decided to use all four message classes.

The SDN bridge thus collects all *Bed Labels*, all *Patient Name Identifications*, and *Parameter Values* and *Parameter Support Data* for all parameters of all beds. Once again, *Wave Values* such as ECG and blood pressure curves are not transmitted by the bridge. However, parameters derived from these curves (such as systolic, diastolic and mean blood pressures, heart rate, parameters derived from ST segments, etc.) are available in the *Parameter Values*. Though not needed, the bridge additionally transmits *Instrument status* messages for historic reasons (this text was written two years after designing the bridge system; I would not include *Instrument status* messages if I redesigned the system now as they are not used).

The rate at which data is retrieved by the bridge PC from the SDN using Speedy is influenced by several factors. The technical limit imposed by Speedy (see section 3.3.4) is that the rate must be a multiple of 1024 milliseconds. For alarming or monitoring applications, this retrieval rate is the main factor that determines how fast it can react to changes, as this is the rate at which it is supplied with data. Using a rate less then the maximum rate causes less network traffic on the LAN. However, using a slow rate decreases the speed at which an alarming application can react to changes in parameters and may cause effects related to undersampling a signal. A trade-off has to be made between decreasing network traffic and increasing response speed for alarming and monitoring applications.

Section 2.3 discussed the topic of choosing sample rates for medical signals (see also [Gravenstein et al, 89]). A rate that allows sufficiently fast reactions to changes in the parameters without causing more network traffic than necessary is once per five times 1024
milliseconds. This is the rate at which the SDN bridge retrieves data from the SDN and sends it to the LAN. In the following sections I will often refer to this rate as once per five seconds, though this is strictly speaking not correct.

Having answered the questions of what data to retrieve from the SDN and at what rate, the part of the bridge PC design applying to its input is done. Before continuing with the output part I have to introduce the options available for LAN networking.

3.5.2 Networking in the bridge PC

The goal of the bridge PC is to retrieve data from the SDN and transmit the received data on a local area network. There are several options for the network protocol to be used on the LAN. This section introduces the options that were relevant at the time of designing the networking part of the data acquisition system (1996).

The networking system should allow communication between the bridge PC and other PCs (client PCs) connected to the network (including the archiver PC). Most client PCs will be modern PCs running the Windows 95 operating system (or a successor of Windows 95). Due to the reasons given in section 3.3.3.2 however, the bridge PC will be running the DOS operating system without any DPMI provider programs. This implies that the software running on the bridge PC must be very conservative in its memory usage, as there is only 640 Kbytes available. This memory restriction should be kept in mind when reviewing networking choices.

3.5.2.1 Classification of networking protocols

This section introduces some topics related to networking protocols. For an introduction to computer networking I refer to [Tanenbaum]. Networking protocols in general can be divided in several groups, and some networking protocols can be used in more than one way. Two ways to classify networking protocols are:

- Connection-based protocols vs. Connectionless protocols.
- Streaming-based protocols vs. Datagram-based protocols

A connection-based protocol is used to communicate between two computers, one called the client and the other called the server. When the client computer wants to communicate with the server computer, it sets up a connection to the server and uses this 'channel' to send data to the server or receive data from the server. Both sides maintain information about this connection used in maintaining the connection.

In a connectionless protocol, when one computer wants to send data to another computer, it sends a packet containing the data and some destination information on the network. A receiving computer continuously scans the network for packets and processes packets that are intended for itself (according to the destination information). The drawback of this system is that there is no way for the sender to know whether a packet it sent did arrive at all, because computers that receive the packet do not need to acknowledge reception of the packet. There are two advantages however. Firstly, the computers do not need to maintain information about currently active connections, which saves both memory space and CPU time. Secondly, a connectionless protocol allows for a sender to broadcast packets: in that case the sender sends a data packet with a blank destination information, meaning that the packet is intended for all computers on the network. Each receiving computer can then decide whether it wants to use that packet or ignore it.
3.5 The SDN bridge

The difference between datagram protocols and streaming protocols is that datagram protocols send data in chunks called datagrams, while streaming protocols transfer data as if it were one big continuous stream of data. On the lowest level of most networks, data can only be sent as datagrams of a small size, so a streaming protocols using such networks must break the data stream into separate packets at the sender’s side and reconstruct the stream from these packets at the receiver’s side. To be able to do this without errors requires a connection, so streaming protocols will always be connection-based. Datagram based protocols do not need a connection, so in principle they are connectionless. However, datagram protocols may present a programming interface in a pseudo-connection style: the sender of the datagrams can specify a default destination that will be used for all datagrams (and hence no explicit destination needs to be specified for each datagram).

3.5.2.2 Protocols for the data acquisition system

There were only three feasible options available for the networking protocol at the time of designing the data acquisition system:

- **IPX.** This is the basic networking protocol used by Novell NetWare. It is however also supported in Microsoft’s Windows 95 networking drivers and often a royalty-free IPX driver for DOS is included in the software that comes with an Ethernet card. IPX is a connectionless datagram protocol that allows both broadcasting and specifying a destination.

- **SPX.** This protocol is in the same family as IPX, but uses a different paradigm to transport data over a network. It actually uses IPX as an underlying protocol to implement streaming. SPX is a connection-based protocol that can be used both as a datagram protocol as well as a streaming protocol. As SPX is connection-based, it does not support broadcasting.

- **NetBEUI/NetBIOS** (NetBIOS is an older version of NetBEUI). This is the basic networking protocol used in Microsoft’s networking products, such as Windows for Workgroups and the networking software built into Windows 95. Like IPX, NetBEUI is a connectionless datagram protocol that allows both broadcasting as well as specifying a destination.

If I were designing the system at the time of writing this thesis (1998), I would surely have included the TCP/IP (streaming, see [RFC 793]) and UDP/IP (datagram, see [RFC 768]) protocols in my evaluation. Though these protocols are intended for use on Wide Area Networks (the internet), they can be used on a Local Area Network as well (intranet). However, at the time of designing the system these protocols were not yet really an option. Additionally, it is hard to get the IP protocols working on a computer that has the memory restrictions of the bridge PC.

3.5.2.3 IPX and NetBEUI

IPX and NetBEUI are very similar protocols, though IPX has an important advantage over NetBEUI. To clarify this difference I have to explain how these protocols are used. Assume there are two applications, each running on a different PC: an application running on the first PC transmitting blocks of data, and another application, running on a different PC, that can receive the transmitted data blocks. In figure 3.5 these are called the transmitting application (running on the transmitting PC) and the receiving application (running on the receiving PC). When the transmitting application wants to transmit a block of data, it starts by packaging this data block in a larger data block called a data packet, which contains the information about the destination of the data in addition to the data block. The transmitting application
then gives this data block to the network driver (the IPX driver or NetBEUI driver), which will send it on the network. Meanwhile, the receiving application has notified the network driver of the receiving PC that it is interested in receiving packets by supplying the network driver with a block of memory that in essence is an empty data packet, to be filled by the network driver. When the network driver receives a packet intended for the receiving application it will copy the data from the network into the empty data packet and notify the receiving application that a new packet has arrived. The receiving application can now extract the block of data from the packet and use it. When there was no empty packet waiting to be filled at the receiver's network driver, the packet will be lost.

Transmitting PC

\[\begin{array}{c}
\text{Transmitting Application} \\
\text{Network Driver}
\end{array}\]

Receiving PC

\[\begin{array}{c}
\text{Receiving Application} \\
\text{Network Driver}
\end{array}\]

Network

*figure 3.5 Network data communication via IPX or NetBEUI*

The most important difference between NetBEUI and IPX is the way in which empty packets are provided to the receiver's network driver by the receiving application. The NetBEUI driver can only have one empty packet waiting at a time. The IPX driver however uses a pool of empty packets. Each time a new packet arrives, the IPX driver removes a packet from the empty packet pool and fills it with the incoming data. After the receiving application is finished with the packet, it can send it back to the empty packet pool. When packets arrive in rapid succession, an application using NetBEUI has to process the incoming packets immediately and provide the driver with an empty packet again as soon as possible, otherwise packets will be lost. An application using IPX however only has to make sure the empty packet pool never empties to prevent this kind of packet loss. For this reason the choice between IPX and NetBEUI will nearly always be in favour of IPX.

The actual situation is slightly more complex, to cater for the case where multiple applications want to transmit or receive data independently from each other on the same computer. This is solved by including in the destination information not only an address of the computer for which a packet is intended, but an 'address' of the application running on that computer as well. Such an application address is known as a socket or port. Where I said in the previous paragraph that IPX maintains an empty packet pool, it would be more correct to say that IPX maintains a empty packet pool for each socket. For efficiency reasons, a socket is identified by a 16 bit number in most network protocols. The only protocol I know of that does not use a 16 bit integer as a socket identifier is NetBEUI, which uses a short string instead. This should be considered an advantage of the NetBEUI protocol, as this

\[\text{13 This does not prevent other forms of packet loss, for instance caused by network collisions, in which case a packet that was sent never reaches the receiving PC at all.}\]
3.5 The SDN bridge

decreases the chance of accidentally using the same socket identifier for two different applications.

To be able to communicate, a transmitting application and a receiving application must agree on the socket number that is to be used. Additionally, the transmitter also must know the network address of the receiver, unless the transmitter is broadcasting. Even when a transmitter broadcasts network packets (that is: packets are sent to all computers in a network) it is still required to specify a socket identifier to identify the intended application on the receiving computers.

3.5.3 Design of the networking interface

For the networking interface of the bridge PC first the choice between a connection-based protocol or a connectionless protocol has to be made. As explained in section 3.5.2.1, a connectionless protocol has the disadvantage that a transmitter cannot detect if a packet it sent did arrive at all and a receiver cannot directly detect that it didn't receive a packet it should have received (there are some methods that allow a receiver to detect indirectly that it did miss one or more packets, for instance including a sequence number in each packet). On the other hand, a connection-based protocol puts higher requirements on both sides of the connection (especially on the server side), and it does not support broadcasting, which implies that the network load grows linearly with the number of clients, instead of being constant.

The main two issues here are the CPU and memory load on the server and an increased program complexity for a connection-based protocol and the possibility of missing data packets for a connectionless protocol. As memory load is a very important issue for the bridge PC and packets do not drop very often on the not heavily loaded LAN in the ICU, I chose to use a connectionless protocol.

The choice between the two available connectionless protocols, IPX and NetBEUI was easier. I chose IPX, as the probability of in-driver packet loss is smaller than in NetBEUI, as explained in section 3.5.2.3.

The main advantage provided by using a connectionless protocol is that it allows the sender (the bridge PC) to be kept as simple as possible. By using broadcasting, it doesn't need to know which PCs are using the data it sends. In that case it can use the network as a 'write only device', further simplifying the bridge PC software. For this reason I chose to use IPX broadcasting to produce the bridge's output.

Only one major part of the bridge PC is now left to be designed: the coupling between its SDN input and its IPX output. I chose to use the most simple option here. As both input and output are in the form of blocks of data (the SDN messages and the IPX packet contents) I chose to simply copy each incoming SDN message as an IPX packet. Only a small header is prefixed to each SDN message. This ten byte header contains three fields: a four-byte signature (containing the letters 'HPDT'), a four-byte sequence number and a two-byte version number (always zero). The sequence number is increased by one for each packet sent, which allows a receiving application to detect that it has missed packets. The signature and version allow the receiving application to be sure that the packet is indeed a packet that came from the bridge PC, and was not produced by another application accidentally using the same socket number (34408) as the data acquisition system.
3.5.4 SDN-to-IPX bridge summary

![SDN-to-IPX bridge diagram]

Concluding, the SDN to IPX bridge is designed as depicted in figure 3.6. It retrieves parameter values messages, parameter support data messages, bed label messages and patient name identification messages from the SDN for all beds connected to the SDN and for all parameters. It does not retrieve wave value messages, so the bridge offers no support for applications like ECG analysis. The retrieved SDN messages are broadcast on the local area network connected to the bridge as IPX packets on socket 34408.

3.6 The patient data archiver

As stated in section 3.4.2, the patient data archiver PC's task is to store data that was transmitted by the SDN bridge PC. The main decisions to be taken in the design of this archiver are how, where, at what rate, and for how long to store the data.

3.6.1 Input to the archiver

The archiver PC receives the packets that are sent by the bridge PC as its input. These packets contain parameter information (values and support data), bed label information and patient identification information. The data that is going to be stored by the archiver will be used by applications (see section 3.4.1) that need to know what the values of the measured parameters at a certain time for a given patient are. This requires storage of patient identification together with parameter information.

Bed label information is pseudo-static information: the names that are used for the beds by the staff of the ICU (e.g. 'room 1, bed 1') only change when the SDN is reconfigured, for instance when new beds are installed in the ICU. In the one and a half years the system is running now in the Catharina hospital's ICU, this has happened once. The bed label information is mainly of importance for real-time applications to identify the bed a patient is currently in, with a name that is meaningful to the ICU staff. As the archiver's output is used for non-real-time applications, I decided not to store bed label information.
The 'parameter information' entering the archiver consists of packets of data, each of which contains both the parameter values SDN message for all parameters of one bed as well as the parameter support data for that bed. One such packet is sent every five seconds for each bed currently in use. When a bed is not in use (that is: no parameters are measured), no data packets are sent for that bed. The parameter values SDN message in each packet contains a set of sub-packets (called data elements in HP's Speedy documentation), each describing one parameter and its value (or values; some parameters, e.g. blood pressures, contain three values). The parameter support data message has a similar structure. Its elements contain 'support data' for each parameter in the parameter values message. This support data includes information on how to interpret the numerical values in the parameter values message (whether the values are signed or unsigned, and what scaling factor is to be applied), and it contains a textual representation of the parameter value. Some parameters only have 'support data', and have no element in the parameter values message; These are parameters whose value is not a number but a text string, such as the result of HP's ECG rhythm analyser.

So, the parameter information enters the archiver sorted by bed. Most applications however would like to see their data sorted by patient. For real-time applications this is not a real problem because at each moment in time each bed is used by exactly one patient (if it is used at all) and each patient is in exactly one bed, so there is a one-on-one relationship between beds and patients. On a larger time scale however, there are some potential problems. During a given period a bed can be used by multiple patients, and a patient can be relocated to a different bed. The goal of the archiver should be to allow applications to access stored patient data for a given patient, not just for a given bed. To accomplish this, the archiver can use the patient identification messages to find out what patient is in what bed.

One should be aware of a possible problem here though: using the patient identification messages to detect what data belongs to what patient does not work correctly when the patient identification messages are not correct. The patient identification that is provided by the SDN consists of two strings that must be entered in a bedside monitor by intensive care staff when a patient is admitted to the ICU. The first string is the name of the patient and the second string is the identification number that is used in the hospital for the patient. One problem is that this information is often entered only after the patient has been in the ICU for a while. Until that time there is no way to identify a patient via the SDN. A second problem is that a patient's identification is sometimes entered incorrectly and is corrected later. As will be shown later, these problems with the patient identification can have implications on what is the best way to store data.

### 3.6.2 Storage options

There are several options available for the archiver PC to store the patient data. The main design aspect to take into account while choosing is the ever present space versus time trade-off. Data can be stored in a complex, space consuming structure that allows finding data fast or in a simple structure for which finding information takes more time. The main choice to be made for an implementation is between a database-based approach and a file-based approach. By database-based approach I mean using a standard relational database tool (e.g. Oracle, Access, Paradox) to store the data. By file-based approach I mean using a custom file-format to store the data.

---

14 At the time of writing, object-oriented database tools (in contrast to relational databases) were not yet commonly available.
Using a standard database tool has the obvious advantages of using a standard tool and provides good searching options. Devising a custom file format allows for storing the data in a non-standard way that allows using the innate structure of the data to store it in a more compact way than a database can, and unlike standard database tools, doesn't involve any licensing costs.

3.6.2.1 Databases

Most commonly used database tools are based on relational databases. See [Paredaens et al] for an introduction to relational databases. In the description of database options below I assume the database used is a relational database (which was the standard type of database on PC-based systems at the time HPDATA was designed).

When using a database, several options exist for the layout of the database tables. The design of the database tables depends mainly on how the data is going to be queried. Typical queries on a patient data database are:

- From what time to what time is data available for patient X?
- What blood pressures were measured for patient X on April 1st 1997?
- What patient was in a certain bed around 15:00 on April 18th 1997?
- What patients were in the ICU on April 1st 1997?
- What parameters were measured for patient X during his stay in the ICU?

Several items that need to be stored in the database can be derived from these queries: patient identification, bed numbers, time information, parameter availability and parameter values.

Two ways of storing these data in a relational database table are (many more ways are possible, but these two are the simplest examples of two broad categories):

- Use one main table, where each record contains: a timestamp, patient identification, the bed number and a value for each possible parameter (these values can be empty if the parameter is not measured for that patient at that time). In this set-up one record is added for each patient every time new data becomes available. In this case parameters that are not measured still take up space in the database.
- Use one main table, where each record contains: a timestamp, patient identification, the bed number, a parameter identification and a value for that parameter. In this set-up one record is added for each parameter known for each patient every time new data becomes available. In this case, parameters that are not measured are not stored in the database. However, the timestamp, patient identification and bed number information is replicated for each parameter (this can be improved by splitting the table into two tables).

The answers to the queries given above can be found by searching the database (using standard SQL calls). The space efficiency of the second database layout is better than that of the first one because of the 63 possible parameters usually only 10 are used at one time (on average). The exact efficiency depends on both the database tool used and what data is stored for each parameter: the examples above simplified the situation a bit because the values of different parameters can be different in nature. The value of some parameters is one number, for others it consists of three numbers (i.e. systolic, diastolic and mean blood pressures) and for yet others it is a text string. In addition to the raw value of a parameter additional information is available that can be stored: a measure for the validity of a value (valid,
questionable or invalid), a flag indicating an alarm condition is currently triggered for that parameter, etc.

The conclusion of this all is that trying to store all data in a relational database will easily lead to inefficient data storage.

Using a database to store the patient data also has another drawback: most database tools (such as Oracle or MS SQL server) require the end-user to purchase an expensive license to use that database. Other database tools (such as Paradox) require no licenses or only a small license fee, but are not able to support multi-user access or require frequent database maintenance.

These reasons caused me to investigate an alternative to using a standard database tool to store the patient data.

3.6.2.2 Files

An alternative for using databases is storing the data directly in files. Of course, a database stores its data in files as well. When I talk about storing data in files instead of databases, I mean sacrificing fast search capabilities to attain a smaller storage size.

Programs using the data stored in files want to be able to 'query' the files for the same queries as given in the previous section. However, it is important to know how these data are commonly used.

The intended applications for these data are health assessment applications and offline applications, as described in section 3.4.1. The main observation to make is that all health assessment applications and most offline applications will use the data for one patient at a time in a sequential fashion. The main required 'search' functions are finding a list of available patients and finding out from what time to what time data is available for each patient. Given this information an application can then process the sequence of data available for a patient. Finding out what a specific parameter value is at a specific moment is only a secondary aspect. A good way of matching this approach in the way the data is stored in files is to store data for only one patient per file, to store it sequentially and to have a mechanism to find the files containing the data for a patient. Note that I didn't say store all data for a patient in one file, just that one file does not contain data for more than one patient. Storing all data for one patient in one file may seem to be a good idea as well but causes a practical problem.

The main problem was mentioned at the end of section 3.6.1: it is not always known to which patient incoming data belongs because the patient identification information stored in the bedside monitors is not always correct. A solution to this problem may be found in storing data from one bed in one file as long as the patient identification for that bed is not changed and data is coming from that bed. I call such a set of data an episode. The data file containing an episode is closed as soon as a bed no longer provides data (meaning the patient has been dismissed from the ICU) or the patient identification string changes (meaning the patient was dismissed and a different patient was admitted to the bed, or a patient's identification was corrected). When the patient identification changes, a new episode data file is started in addition to the closing of the previous episode data file. This approach makes sure that each data file contains data for only one patient and for one bed and that data is available for each sample in the time interval stored in the data file (that is: there are no 'gaps' in the data). However, data for one patient may be split up in different files, each containing data from a different non-overlapping interval. The problems caused by incorrect or missing patient identification information are still not solved by this approach, but periods for which no
patient identification is known or for which patient identification may be incorrect are isolated now to separate files.

This approach in itself provides no implicit searching capability for finding out for what patients data is available and for finding the data files belonging to one patient. A way of providing this searching capability is to use a database table to store information on each episode (making the data storage a hybrid database/data file approach). Each record in that episode database contains information for one episode: the file name of the file storing the data for that episode, the name of the patient (if known), the identification code for the patient (if known), the bed providing the data, the start time and the end time of the episode. Standard SQL queries can be used to find a list of data files for a certain patient (given the patient's name or identification code).

The combination of such an episode database together with the episode data files requires less disk space than a full database storing all data, because a custom designed data file can store patient data more efficiently than a general purpose database tool. The episode database is very small in comparison to a full database because it stores only one (small) record for each data file. The approach described here still does not define what the layout of the data files should be. The design of these data files can still provide some search capabilities, such as providing a fast way to discover what parameters were measured for a patient. I will address this topic in the next section.

This episode approach is not the only way to store patient data in files. However, the episode approach provides a compact way to store the data (dependent on the implementation of the episode data files), while still providing the most important search functions. Other approaches providing more advanced search functions tend to become in essence a development of a custom database, which is a topic that deviates too far from my basic assignment, and is probably not worth the effort.

For this reason and the compactness of the output generated by the episode approach when related to a full database approach I chose to use an episode approach for storing the data by the archiver. The episode database was implemented as a single Paradox table.

3.6.3 Data file layout

Several aspects are important when defining how patient data is to be stored in the episode data files:

- Shared reading and writing by different applications.
- Search capabilities.
- Flexibility of the file format.
- Summary information.
- Space efficiency.
- Data compression.
- Sample rate.
- File size limiting.

3.6.3.1 Consequences of shared file access

Some of the applications that will use the output of the archiver (health assessment applications) would like to access the patient data as soon as possible. For instance, an application showing graphs for the parameters measured for a patient probably wants to
include the most recent data. When the archiver keeps its episode data files open for writing until the episode is finished, the graphing application must be able to read the data from a file that is being written at that moment by the archiver. This means that the archiver must allow the files it is writing to be read by other applications. This does not only imply that both applications must open the file in a share-aware fashion, but also implies that the problems related to resource sharing in a multiprocessing environment may occur. This type of problem occurs when the reading application tries to read sections of the data file that the writing application was modifying at the same moment: the reading application may read a mix of modified and unmodified data. When the data file organisation is not a linear sequence of data records, but a more complex file structure, this means the reading application essentially reads incorrect data. Complex file structures, such as for instance B-trees, allow for fast searching of data in the file. A file structure that is a linear sequence of data does not allow for fast searching of data \footnote{Fast searching in sequential files is possible when the data is written as records of a fixed size. But not being restricted to fixed-size records was one of the reasons to store data in files instead of in databases, as explained in section 3.6.2.1.}, but it circumvents the reader-writer sharing problem because data in the file is never modified once it is written.

There may seem to be a way to prevent reading data from a file that is open for writing by another application: the archiver can opt to not keep its output files open permanently, but only to open the output file when it wants to write some data, write that data, and then close the file as soon as possible. If the writing application disallows sharing of its output file, and the reading application disallows sharing of its input file by the writer, conflicts as mentioned in the previous paragraph cannot occur. However, the situation in this case is even worse for two reasons: if either the reading application or the writing application tries to open the file while the other application opened the file, a sharing violation is generated: the operating system refuses the second application to access the file opened by the first application. That application has to retry opening the file until the other application closes the file. This situation is not acceptable. An other negative effect is that opening and closing a file for writing under Windows 3.1 (for which the archiver system was originally designed) causes the application to wait until the modifications actually have been written to the hard disk. This causes extra strain for the hard disk, which would not occur when the file being written would not be closed, because the disk caching mechanism then can delay the actual write to a more convenient moment. Another related solution would be to prevent common access to a file by a reader program and a writer program by restricting access to parts of a file using record locking. However, a few experiments showed that using the record locking functionality in Windows 3.1 does not work as expected. Concluding, the only viable alternative to possible resource conflicts is disallowing reading the data until the episode is finished. This is not acceptable as well however.

Some way of handling the problems caused by shared reading and writing must be devised. A simple but effective solution becomes apparent when realising that those problems only occur when the writing application modifies an existing part of the file. The solution is never to modify any part of the file, but only to append data at the end of the file. The drawback of this approach is that advanced file formats, that allow searching capabilities other than linearly scanning a data file, are not possible. Despite this disadvantage, I chose to use this approach (only appending data at the end of the data file) for the data files produced by the archiver.

### 3.6.3.2 Sequential block files

This section introduces the layout of the data files that are written by the archiver software. A more detailed description is given in Appendix A. I designed the file format primarily based
on the sequential write requirement mentioned above and on the desired flexibility of the file format. The data files consist of a sequence of variably-sized records (see figure 3.7). Each record consists of two parts. It starts with an eight-byte header part, followed by the header-dependent data part. The header part contains three fields: the size (in bytes) of the record (at most 65535 bytes), the time the record was created (as a standard UNIX-style timestamp: the number of seconds elapsed since January 1st 1970) and a record type field. The type is a two-byte integer defining the function and interpretation of the data part of the record. Several types are used in the patient data files, e.g. there is a type indicating that the data part contains patient data and a type indicating the data part is a file-header containing summary information of the entire data file. This file format, that I will further refer to as Sequential Block File or SBF, is flexible in the sense that new record types can be defined later, while existing software is still able to use the new files by just ignoring the records with types it does not know.

```
figure 3.7: SBF file structure
```

Defining a new record type consists of two parts: a two-byte integer must be chosen to uniquely identify the new record type and the layout of the data part must be defined. A full definition of the SBF file format and the record types that I used in the archiver is given in appendix A.

3.6.3.3 SBF record type changes

A problem may occur when a flaw in the data part layout is discovered after a record type has been in use for a while. Changing the definition of a record type will break existing applications and new applications will not work with old data files.

The SBF file structure has the flexibility to solve this problem in several ways. Below I present a few example solutions to this problem. In this discussion I assume that the layout is missing some useful fields that need to be added, not that some superfluous fields can be removed (which is not possible without breaking existing applications).

The first solution is keeping the existing definition and add a record of a new type containing the extra information immediately after the 'old-style' record in new data files. The problem
3.6 The patient data archiver

with this approach is that new applications must look ahead in the file to determine if the new information is present.

The second solution is based on the fact that extending the data part for an existing record type does not necessarily break old applications. New applications can discriminate between the old or new version of a record type by checking the size of the record (as stored in the record header). Data parts of existing record types can be safely extended with new fields if the following two conditions are met:

- The existing record type has a fixed size. This allows new applications to differentiate different versions of the record type by checking the record size. It also guarantees old applications will continue working because they only use the first part of the record. This is not necessarily true when the data part can vary in size. In that case the layout of the old-style data often depends on the data size.

- No application should assume a record of a certain type has a certain size, but it should always check the size as recorded in the record header. All applications should be able to handle records of up to 32 kilobytes (the maximum size of any record).

A third way to extend record types only applies when it is known in advance that the record type may need to be extended later. In that case one may add a link field that is the offset in the data file of a record that contains the extended information. This is the file-equivalent of using pointers in memory-based data structures. This can for instance be used in a file header record type to 'link' to a record at the end of the file containing summary information or an index for the file. Just before the writer of the file closes the file it can add such a summary record and modify the link in the header record, so the summary record can be found by an application reading the file without having to read all records in the file. This third solution also allows existing data files to be updated to a new version by appending the extension data and updating the link field. A warning should be given here though: as existing data in the file is modified a small possibility of conflicts arising from simultaneous reads and writes in the same file exists. However, such summary information is intended to speed up offline style applications, which usually do not use recently recorded data, in other words they do not read data at the same time that data is being written.

When the situation arises that an existing record type needs to be modified, the programmer may choose any of these three methods, provided the necessary requirements are met: to be able to use the third method, the record type to be modified should already have a link field.

3.6.4 Stored data

Up to now I only described the tools to be used for storing the data (the episode database and the SBF patient data files), but not what data is actually stored.

Data for each logical source defined by the bridge PC is arriving at the archiver at a rate of once per five seconds. There are 25 logical sources that can send data: one for all patient names SDN messages and one for each of 24 possible beds containing the parameter data and parameter support data. For real-time applications once per five seconds is a good rate that allows reasonably fast response to sudden changes in any parameter. For storage purposes once per five seconds is overkill however. That rate generates a tremendous amount of data to store, while most signals do not change significantly in between samples. To reduce the amount of data to be stored I decided to sample the incoming signals down to once per thirty seconds before storing them (see section 2.3 for a discussion of sample rate issues).
The HPDATA System

This reduces the volume of data to be stored by a factor of six, without throwing away too much information (see [Gravenstein et al, 89]).

To perform the sampling down I used a simple approach: the archiver stores the last incoming data value for each bed and for each possible parameter. A timer triggers every thirty seconds a writing action, that collects the most recent data for each patient in a record and writes it to a file. After collecting the data the timer action clears all stored data fields. This is necessary to determine whether a parameter is no longer measured. As SDN does not provide information when a parameter measurement is discontinued, failing to clear the parameters stored in memory after writing them to a file would cause the last measured value to be repeated over and over again.

Concluding, the data stored in the data files are stored as one record per thirty seconds per bed. Each record contains data for every available valid parameter as well as the patient identification information. The exact layout of the records as stored in SBF files (see previous section) is described in appendix A.

3.6.5 Implementation of the archiver

I implemented the archiver system for use on a PC running the Windows 3.1 operating system, which was the standard operating system at the time (first half of 1996). At that time my main experience was in programming language C, though I already discovered that building Windows user interfaces was far easier using Delphi (a rapid application development tool that uses Pascal as its basic language). Delphi has the additional advantage of having built-in support for database operations. I decided to implement the system using a mix of both languages: the basic Windows framework and the user interface was implemented in Delphi, while the data processing functions were implemented in a DLL written in C.

A second DLL written in C was used to allow multiple applications running on one computer to use the data coming from the bridge PC. This circumvents a problem with the IPX protocol that I discovered only after I already had committed to using IPX: on one PC there can be only one application that uses a given IPX port at a time, so only one application can use the data sent by the bridge PC. If a second application tries to use the same port, the call in the IPX driver used to open that port will fail. The winipx2 DLL I wrote bypasses this problem: it allows multiple applications to receive IPX packets from the same IPX port. The first application to use this DLL to open an IPX port will cause the DLL to actually open the port. A second application trying to open the same port using this DLL will not cause a new attempt to really open the IPX port, but will just receive copies of the packets that are received on this port. This works because in Windows 3.1 DLLs can be used to share memory.

A problem with this approach occurs however when trying to translate the software to a native Win32 (Windows 95/98/NT/2000) application (which was never done, but may be done in the future). In Win32, DLLs do not share memory, and each time a DLL is loaded by a program it will behave as if it was the first time the DLL was loaded. Implementing the same functionality in a Win32 environment would require some changes in the way winipx2.dll works. After a few experiments with using IPX on a Win32 I discovered that the winipx2 DLL is not needed at all in a Win32 environment, because unlike the Win16 (Windows 3.1) IPX driver, the Win32 IPX driver does allow a port to be opened more than once on one machine.
3.7 Discussion

3.7.1 Summary

I implemented the HPDATA data acquisition system as depicted in figure 3.4. The Bridge PC uses the Speedy board to acquire data from the SDN and broadcasts a burst of IPX packets containing patient parameter data once per five seconds (once per 5120 milliseconds to be more precise). PCs on the IPX network can pick-up these network packets for use by real-time applications. These packets are also the input for the Archiver PC. The archiver PC keeps track of the most recent data received for each parameter from each patient by storing each parameter from each patient in a temporary memory buffer. Once per thirty seconds the archiver collects the most recent sample for all parameters for each patient and stores the collected data as variably sized records in an SBF file (a separate file for each patient). Each SBF file contains data for one uninterrupted sequence of data for a single patient. When the sequence is interrupted (because the patient was moved to another bed for instance), a new file is started. This means that data for a single patient may be stored in several files, each describing data from a separate time-frame. The archiver maintains an episode database to keep track of what data files belong to what patient.

The HPDATA system has been in use since September 1996. On average there are about twenty patients concurrently in the ICU. For each patient about sixteen parameters are measured.

The main application that uses the data stored by the archiver is a health assessment style application. It is used to view graphs of the patients for which data is available. This application is used extensively during the briefings of the intensive care staff.

3.7.2 Suggestions and problems

The system of storing the most recent samples once per thirty seconds works well, though after designing it I had thoughts about improving it. The quality of the data would probably improve by not storing just the last sample of a parameter before the thirty second timer triggers, but to use the median of all six measurements. As this idea only grew long after I finished implementing the archiver system, I never implemented it.

No computer system is perfect, and HPDATA is no exception. After more than two years of use I can summarise the problems that occurred as follows:

- The patient identification strings used by HPDATA are the strings that are entered by intensive care staff at the bedside monitors. These strings are quite often not entered at all or sometimes entered incorrectly. I made a tool that allows changing the identification strings in the episode database, which solves most of the resulting problems. These changes are only recorded in the episode database, not in the data files themselves.

- The Paradox table storing the episode database sometimes becomes corrupt, for instance when someone switches off the computer without terminating the archiver application. I never noticed that this caused the table to become unusable, though some records may be lost and some fields may contain invalid data (especially the start and end times of an episode contain incorrect values). I made a tool that can reconstruct a table with incorrect data from the raw data files, which also checks the
The HPDATA System

data files for errors (they too can become corrupted in case of a sudden power-down).

- No standard back-up tool is present. To prevent the hard disk from filling up, data files have to be removed every now and then (once every three or four months) and the episode database has to be updated to remove the corresponding records. The removed data files can be archived somewhere else, together with their associated episode database, for instance on a recordable CD. Such a CDR can be used for offline applications or to review past patient data.

- Currently, freshly archived data can only be accessed on the archiver PC. (Once data is archived on a CD-ROM or data files been transported manually to another PC these data can be used elsewhere). There are two reasons for this limitation: the episode database is implemented as a Paradox table; Paradox does not support access from several computers in the network simultaneously. The other reason is that network file sharing (needed to access the data files) is not working very well in Windows 3.1, the operating system the archiver is running on. When enabling network file sharing, Windows effectively disables the write caching for the hard disk. As the archiver writes records to all its open data files (usually 16-20 data files are open at any time) every thirty seconds, this causes a very high load on the hard disk. In essence, the computer locks up for five seconds every thirty seconds trying to write all data, when network file sharing is enabled.

Solving the latter problem requires redesigning the way archived data is accessed. Currently, the data is accessed by using the episode database directly and by reading the data files directly. Hiding both these accesses behind a standardised networked interface probably would solve the problem. Using the current state of the art (1998 technology) this would at least require to abandon Windows 3.1 as a host operating system and for instance switch to Windows NT or Windows 95. Continuing to use such a Windows environment (instead of switching to a radically different environment like a Novell or Unix server) ensures that existing software does not need to be changed very much. The only problem to expect was already mentioned in section 3.6.5: the way the data for the archiver is retrieved from the IPX network works different in a Win32 environment.

The problem may be a good candidate to be solved using an Intranet style approach: the patient data can be served by a HTTP server (see [RFC 2068] for an introduction to HTTP) that uses queries provided as URLs to send the desired patient data files to the client. Using HTTP has the advantage that it is a very flexible standard protocol and that standard solutions exist to cache data received by the client to reduce network traffic. I did not further investigate this topic.
Chapter 4  
Data Processing Core

4.1 Summary
This chapter describes the design of COPDA (Core Components for Patient Data Analysis). COPDA provides a common software foundation for a class of patient data processing and analysis applications. It consists of a set of COM components that can be used by programs as providers of patient data processing functionality. The reader is assumed to have at least a basic understanding of basic COM concepts: components and interfaces. See sections 2.8.2 and 2.8.3 for an introduction to these topics or [Inside COM] for a detailed discussion.

There are two viewpoints to the features provided by COPDA: the viewpoint of the programmer creating a program that uses the COPDA components (a program developer) and the viewpoint of the user of such a program (an application developer).

From the viewpoint of a programmer using COPDA as a foundation for his program, COPDA is a set of (COM) components. COPDA defines several (COM) interfaces required for using these components. Some of these interfaces are supported by the COPDA components themselves (that is: the COPDA components can be accessed via these interfaces). Other interfaces are interfaces that COPDA expects to be supported by other components in the program (allowing the COPDA components to make use of those other components).

From the viewpoint of the user of a COPDA-based program, COPDA provides a custom language (CPMDL - COPDA Module Description Language) for specifying data processing operations. The structure of CPMDL is such that a COPDA-based program may present this language as 'graphical language' to the user. In this case data processing operations are not specified in a textual fashion, but by graphically building a network of standard data processing modules.

4.2 Definitions
Before analysing the requirements for COPDA I will introduce a few concepts that are used extensively in this chapter (and the following chapters).

4.2.1 Interfaces and components
The concepts of interface and component were introduced in section 2.8.2. An interface is a pure abstract class, a class that that comprises only abstract methods. A component is an object that supports one or more interfaces. Besides viewing an interface as a pure abstract class, interfaces can also be viewed from a few other perspectives. Viewing interfaces from a different viewpoint may aid in better understanding the concept of interfaces. Two of these alternate perspectives are:
• An interface can be viewed as a group of related method prototypes. Using interfaces is a convenient way to define more structure in a class that has a large number of methods: related methods can be grouped together into an interface.

• An interface can be viewed as a contract or specification. The programmer who uses a component that supports a certain interface knows how to use that component (via that interface). The programmer who implements a component that supports that interface knows what functionality has to be implemented and how that functionality is to be made available to the component user. If the implementer wants to build more functionality in the component than can be accessed via this interface, he should do so by defining a new interface and let the component support that new interface (in addition to supporting the old one). The previously agreed upon interface should never be changed though. See [Inside COM] for more a more detailed treatment of this viewpoint.

4.2.2 Applications and programs

In this chapter the words application and program will be used in a specific sense that the reader should be aware of. In normal computer-related English these two terms are often used as synonyms. I will define them more strictly here. In previous chapters the need for distinction between these two concepts was not as urgent, and in most cases I used the word application to indicate either of them.

A program is an executable file that can be run on a computer. In this thesis I only consider programs running on an MS-Windows operating system, so a program is a file with the .EXE file extension.

An application is a task that can be performed using a program.

Many programs can be used for only one application. In this case the distinction between program and application is usually not relevant. For example, a few years ago a typical word processing program could only be used as a word processing application.

Other programs combine a few applications. For example, modern word processing programs can often be used as a drawing application in addition to being used as a word processing application.

Yet other programs can run a virtually unlimited number of applications. The DataWand program that will be described in the next chapter is itself an application that can be used to design data processing algorithms. It can also run these data processing algorithms. Each data processing algorithm itself can be considered a separate application, though these data processing algorithms aren't programs themselves but need a 'host program' (such as DataWand) to run.

Often the distinction between program and application is fuzzy. In general, a program defines how it looks (the user interface) and what it can do, while the application defines what it does.

4.3 Initial requirements

The primary objective of COPDA is to provide data processing functionality common to the three categories of applications mentioned in section 2.2.1: real-time applications, health assessment applications and offline applications. Note that these three application types normally correspond to three different program types as well. A program that can run both
real-time applications and offline applications is conceivable, but the rather different requirements that these application types put on the program's user interface make usefulness of such a combined real-time and offline program unlikely.

COPDA is a *software foundation* that is to be compiled into data processing programs. (A better explanation of what COPDA is - better than 'a software foundation' - will be provided in section 4.4.) Programs using the COPDA technology will be referred to as COPDA programs. Data processing applications executed by COPDA programs will be referred to as COPDA applications. Usually usage of these two terms in this thesis also indicates that the distinction between program and application is relevant.

Designing COPDA starts by identifying the common functionality found in applications in these categories. To arrive at design specifications for COPDA it is also helpful to identify application features that are different amongst the applications and hence should not be included in the COPDA functionality.

*Real-time* applications monitor an incoming *stream* of patient vital signs data. Typically, their function is to recognise patterns in the incoming data that indicate a dangerous situation for a patient. Real-time programs usually run unobtrusively in the background. User interactions required by real-time applications are limited: the main user interaction required by real-time applications is providing a way to display warning messages.

*Health assessment* applications use recently measured patient data for assisting medical staff in assessing a patient's health situation. Typically health assessment programs display vital signs graphically and provide functionality to manipulate this graphical display. They may also include annotation functions, and functions for data validation. Health assessment applications require a user interface for selecting what data to process (e.g. selecting a patient whose data is to be viewed) and for manipulating graphical data. Data access by health assessment applications should be designed carefully, as the data is often provided by a medium (file or database) that is shared with other applications (such as the system acquiring the data).

*Offline* applications process stored patient data that typically was *not* measured recently. Typically offline applications are used for analysing data, for instance to discover and define patterns whose recognition is suitable for implementation in a real-time application (data mining). “Analysing” a patient's data covers a broad area ranging from applying data processing algorithms to statistical analysis and requires a user interface to perform these kind of operations. Unlike health assessment program design, there is usually no need to design data access share-aware in offline programs.

The following major differences between the application categories can be identified:

- **Input paradigm.** Real-time applications use a *data-push* style input paradigm: processing a data sample is triggered by the arrival of a new sample in the input stream. Health-assessment and offline applications use a *data-pull* style input paradigm: processing a data sample is not triggered; a new sample is processed after the processing of the previous sample is finished.

- **Data interface.** The three application categories access input data differently and produce different types of output. Not only is the input paradigm different for different application categories, but even applications within the same category can use different kinds of input, such as reading different file formats. The output of real-time applications typically is an occasional warning message; the output of health-assessment applications is a set of output graphs or a set of numbers; the
output of offline applications can be graphs and numbers as well, but also messages displaying processing results and data files containing intermediate processing results to be used with other tools.

- **User interface:** The three application categories have vastly different user interface requirements. The main function of the user interface of a real-time program is to display messages. The user interface of a health assessment program should be easy to use by the medical staff using it in their daily practice. The user interface of offline programs is complex in the sense that it should make a multitude of data processing functionalities available to the user.

The similar part in the application categories can be summarised as follows:

- **Processing template:** 'Processing' data follows a similar template for all applications: first the application (or user) specifies what input is to be processed and what to do with the output. Next the application applies an application-dependent algorithm to the input samples and produces its output. Often, the application-dependent algorithm can be described by a combination of application-independent data processing primitives, such as standard filtering algorithms, basic arithmetic functions, checking a signal against limits, etc.

Several requirements for COPDA can be derived from these points:

- **Functionality provided by COPDA** should allow both input paradigms: data-push and data-pull.

- COPDA should be flexible in the way it treats inputs and outputs. It should not be limited to a restricted set of input sources and output destinations. Instead, in the ideal case COPDA should be able to use any type of input or output that a COPDA application would like to use. This means that a program that uses COPDA must implement 'new' input and output functionality itself, in a way that can be understood by COPDA. This is where COM interfaces come in handy, as COPDA can define the interfaces that the program-defined input and output components should adhere to.

- COPDA should not put any restrictions on the user interface of a COPDA program. It should neither require any particular element in the user interface to be present, nor prevent any kind of user interface technology from being used, nor require any user interface element that is present in the program to be implemented in a particular way.

- COPDA should provide a set of data processing primitives and a means to combine these primitives. As such a set can never be complete, COPDA should provide an extension mechanism to add new primitives to this set 'dynamically'. 'Dynamically' adding new primitives means that primitives defined in a new DLL can be used without having to recompile COPDA (or the COPDA program). In a stricter sense 'dynamically' could mean that COPDA applications are able to use such new primitives without even having to restart the application. Such 'hot-pluggable' extensions may be very useful in real-time applications: new features can be added without having to shut down a running application.
4.4 Defining 'software foundation'

There is one additional point that influences the decision for a basic structure of COPDA:

- As COPDA is originally designed for use with HPDATA (the data acquisition system presented in chapter 3), it should at least be able to use the features of the data collected with that system. The relevant features of these data acquired with that system are:
  - All values in these data are annotated with a validity flag.
    COPDA should provide a system to use this validity information.
  - Not all signals in these data are numerical in nature. For example, some 'signals' are not sequences of numbers, but are sequences of text strings.
    COPDA should be able to process such non-numeric signals.
  - All signals provided by the HPDATA system are synchronised to a single rate: all signals have the same sample rate (once per five seconds for real-time data and twice per minute for stored data). Signals that are measured at irregular intervals (e.g. Cardiac Output) are treated in a 'sample and hold' fashion: such signals are still synchronised to the common sample rate, providing the most recent measurement instead of the current value (this is the way SDN provides these signals to HPDATA).
    To be able to work with these data, COPDA does not need to be able to work with asynchronous or multi-rate signals, but a capability to work with synchronous data is sufficient.
  - Some of the signals comprise multiple sub-signals (e.g. most blood pressure signals comprise separate values for systolic, diastolic and mean pressures). In most cases it is useful to treat such sub-signals as separate values, though they may share some properties (such as validity).
    COPDA should provide a model to access these sub-signals as separate values.

4.4 Defining 'software foundation'

COPDA provides a 'foundation' for data processing programs/applications. The concept of a 'software foundation' was kept abstract in earlier sections because there are several radically different approaches available for implementing such a system. This section will explain the approach that was taken with COPDA.

Three approaches for a software foundation are (other approaches exist, though those are often mixtures of the approaches mentioned below):

- A function library
- Using an existing commercial tool
- Using a data processing core

4.4.1 Function library

A function library provides pre-programmed data processing functions. A programmer using a conventional programming language can use the library functions to perform complex data
processing tasks; using such a library saves much programming work. Such data processing libraries have been in existence for nearly as long as computers exist.

The main drawback of a function library is that it only helps the programmer with a few of his tedious tasks. The programmer still has both the task of performing the 'standard' programming work (creating the program) as well as programming the data processing task (creating the application). Also, to users of the application, the library provides no benefits, as it is invisibly linked into the program. For real-time applications and health-assessment applications this may be of little relevance, but for offline applications this means that the programmer has much more work to do than for instance with a data processing core based system (see below).

4.4.2 Commercial tools

Some data processing applications may be designed as an application purely built with existing commercial data processing programs (such as MatLab or LabView). Other applications can be designed basically as an application built with such a tool, with some custom extensions. Yet other applications can be designed as programs that use such tools as a function library, providing standard data processing functionality. These three variants may be viable approaches for some data processing tasks but have certain drawbacks, as already mentioned in section 2.6.2:

- Some of these tools (e.g. LabView) put restrictions on the user interface of applications built with them (they enforce certain user interface choices).
- All of the tools I investigated only process numerical data (with support for symbolic manipulation of formulae in some cases). I did not find any support for processing textual signals, nor support for processing signals with additional validity information.
- All of the tools I investigated required not only a one-time purchase cost, but also a licensing cost for users of applications built using that tool. This hampers distribution of such applications.
- For some of the tools the basic concepts they use must be stretched to their limits or beyond to fit a real-time data processing paradigm.

4.4.3 Data processing core

A different approach is taken when the task of a data processing programmer is analysed further. This task can be separated in two (more or less) independent sub-tasks: specifying how to process data (designing the data processing algorithm) and designing the rest of the program (including the user interface, data access system etc.). This is essentially the distinction between designing the application (what to do) and designing the program (what is available to do it), as indicated in section 4.2.2

Such a separation can be achieved by using a data processing core that, on one side, provides a capability to specify data processing operations in a generic fashion and, on the other side makes itself available as a component that can be incorporated into a standard Windows program. As such the data processing core separates the two sub-tasks of a data processing programmer.
4.4 Defining 'software foundation'

Such a data processing core is a ‘software module’ designed to become a part of any program that has need of its functionality. For now I leave the form this ‘software module’ takes unspecified. It can take any form that is used in software engineering for achieving modularity. For example it could take the form of a collection of related functions and data structures, it could take the form of a set of related objects, it could take the form of a single object, it could take the form of a set of components and interfaces or it could take the form of a single component (with one or more interfaces). What form is chosen is a design choice.

One of the two tasks of using the data processing core is incorporating it into a Windows program. This is easiest done when the core can be treated as some kind of object in a conventional programming language, e.g. by making the core a COM component (or a set of co-operating COM components).

Specifying the data processing algorithm on the other hand is a task that is more easily done using a dedicated data processing language, instead of using a conventional language. Using a dedicated language for this purpose has the following advantages:

- A dedicated language can be interpreted instead of being compiled, allowing the entire program to be built without fixing its data processing algorithm. For offline programs this is more or less a necessity to be usable at all. For real-time programs this makes application maintenance easier.

- A dedicated signal processing language typically allows describing signal processing algorithms much more compactly (and hence less error prone) than is possible with a conventional programming language.

There are also drawbacks in using a dedicated signal processing language:

- A dedicated language is, well, dedicated. For most users it will be a new language, that has to be learned.

- An interpreted dedicated language executes slower than a compiled language.

However, the first drawback is less important then it may appear at first. The problem with learning any new programming language (especially non-general-purpose languages) is not so much learning the 'grammar' of the language (the syntax and the concepts provided by the language), but much more the learning of the 'idiom' of the language (what functions are available and what they can be used for). When using a system that requires the use of a 'new' language, the time spent learning the 'grammar' of that language is typically short in relation to the time spent learning the 'idiom' of that system. When using a system that only requires use of a conventional language (e.g. a function library), the 'grammar' may be familiar to the programmer, but learning the 'idiom' of that system still takes as much time as learning the idiom of a dedicated language.

The second drawback implies that running an algorithm in an interpreter instead of compiling it into a program will incur a certain overhead. This overhead can be small when the interpreter is designed smart. For a typical interpreter this overhead is linear: when a compiled algorithm would execute in a time $t$, the same algorithm can be executed by an interpreter in a time shorter than $K \cdot t$ (with $K$ being an interpreter dependent constant).
4.4.4  Conclusion

Reviewing these three possible directions, a choice has to be made in what style to develop COPDA.

Styling COPDA as a function library is not a reasonable option. There is not much that can be done on this front that hasn't been done already. Also, a function library style may have some uses for real-time applications and health-assessment applications, but it does not really help development of offline applications very much.

Designing COPDA using existing commercial tools has the drawbacks mentioned above: data processing is limited to numerical data processing, freedom of the application user interface may be limited, there are licensing cost issues and fitting both real-time data processing and stored data processing in the same processing model may be difficult in some tools.

Styling COPDA as a data processing core on the other hand looks like an ideal choice. This comes probably not as a surprise to readers remembering that the acronym COPDA stands for Core Components for Patient Data Analysis; COPDA got its name only after I decided to shape it as a data processing core.

4.5  Two views of COPDA

A data processing algorithm (application) developer's view of COPDA (High-Level)

---

COPDA language (CPMDL)

---

COPDA

---

COPDA components and interfaces

A program developer's view of COPDA (Low-Level)

The services provided by COPDA can be viewed from two sides, depending on who is using COPDA and for what purpose. The distinction is parallel to the distinction between programs and applications as defined in section 4.2.2.

- **Low Level**: To a program developer, COPDA provides a basic foundation for data processing. The program developer however does not need to be concerned with the actual data processing algorithm to be executed (the application): the goal of using COPDA in a program is to provide the user of that program with a set of data processing capabilities. The program developer can choose to hide the data processing capabilities of COPDA and let COPDA execute a fixed algorithm (the COPDA program can only run one COPDA application in that case). This may be a viable approach for real-time applications.
4.6 Software model

- **High Level**: To a data processing algorithm designer (the *application* programmer), COPDA provides a system to specify data processing algorithms that - in principle - are *program*-independent. An algorithm (COPDA application) for use with a real-time COPDA program can be designed using an offline COPDA program (assuming the real-time data themselves can be simulated adequately in the offline system). This specification system, centred around the *CPMDL* language and its *module* concept, will be explained in later sections in this chapter.

In the following sections, it is important to keep the distinction between these two levels in mind. Some COPDA features are strictly related to one of those two sides. In descriptions pertaining to the *low level view*, COM concepts such as *interface* and *component* are frequently used. The context in the *low-level view* is always the design of programs (in the strict sense of section 4.2.2). In descriptions pertaining to the *high level view*, CPDML related concepts such as *modules* and the name CPDML itself are frequently used. The context in the *high-level view* is always the design of *applications* (in the strict sense of section 4.2.2). Though many COPDA features are mainly associated with one of the two levels, they tend to have implications in the other level as well.

In the rest of this chapter the various aspects of COPDA are introduced, starting with concepts related to the *low level view* and gradually working upward to pure *high level view* topics.

4.6 **Software model**

I decided to take an interface oriented design approach using COM to design COPDA. Section 2.8 introduced this model. For the rest of the chapter the reader is assumed to understand the basic concepts used in COM: *interfaces* and *components*.

The main advantages of such an approach over a 'normal' object oriented design are:

- *enforces a better abstraction* in the API by hiding data fields of objects.
- *provides the advantages* of multiple inheritance without most of the disadvantages. This greatly improves extensibility of COPDA and usability of such 'extended' COPDA features in existing programs.
- *defines a standard* for extending programs, using components provided in separate DLLs. This extension method solves several problems related to 'version management' and communication of software parts written in different programming languages. As extensions are DLLs, COPDA does not need to be recompiled to use such extensions.
- *provides automatic memory management* that is less bug-prone by using reference counting.

There are some disadvantages as well though:

- *some of the features* of COM (reference counting and interface querying) come at the cost of extra execution time.
- *the drawback of reference counting*: the designer should make sure there are no circular references.

These drawbacks are not problematic though. Circular references can be prevented by careful design. The small loss in execution speed should be viewed as the result of balancing flexibility against execution speed.
4.7 Data interface: input and output

Before designing a structure for the way COPDA processes data internally, I will first focus on the structure of input and output of data. This structure has two aspects: first, the model used by COPDA to define its relation to data inputs and outputs (the input/output model) and second, the assumptions made on the signals that are to be processed (the signal model). Because these aspects are intertwined, I will first introduce the input/output model at a macroscopic scale, then describe the signal model, and finally provide details of the input/output model.

Some of the analyses of the following subsections will lead to the definition of COM interfaces. To keep the text more readable, the details of these definitions are not presented in this chapter, but in appendix B.

4.7.1 Introducing the COPDA input/output model

COPDA’s requirement to be independent of any particular input or output format (see sections 1.2.3.1 and 4.3) is realised by keeping the implementation of the actual access of input and output outside the implementation of COPDA itself. The actual (program-dependent) data access should be performed by program-defined components (called data sources and data destinations). COPDA defines the interfaces that these components must support to allow COPDA to access them.
COPDA's *input/output model* defines what parts play a role in the communication between COPDA, inputs, outputs and the program using COPDA and it defines the interfaces these parts use for communication. The components and their relations are depicted abstractly in figure 4.2.

### 4.7.1.1 COPDA engine

The COPDA engine is the component that provides the actual data processing functionality. To allow treating different kinds of data equally, it uses *data sources* and *data destinations* to convert data to and from a standard representation.

A short note on the number of components provided by COPDA is in place here. In a certain sense the COPDA engine is COPDA and that engine is the sole component that is accessed by any COPDA application. In this sense it is valid to talk about the *one COPDA component* (being the COPDA engine). However, the COPDA engine is not monolithic: it contains a myriad of other components that can be accessed indirectly via the engine component, as will be explained in sections 4.8 and later. In this latter sense, it is equally valid to talk about the *set of COPDA components*, one of them being the COPDA engine (albeit a special one).

In addition to *implementing* the COPDA engine (and its subcomponents), the COPDA system also *defines* a large number of interfaces. Some of these interfaces are *supported* (implemented) by one or more of the COPDA components and are used by COPDA programs to access the COPDA components. Other interfaces are the vehicle used by COPDA components to access program-defined components (such as *data sources* and *data destinations*). These are not *supported* by the COPDA components, but are intended to be *supported* by components implemented by the program.

### 4.7.1.2 Data sources

*Data Sources* are components that supply data to the COPDA engine. A Data Source reads data and presents this data in a standardised fashion for use by COPDA (possibly first converting the format of the data). This *standardised fashion* is the *data transfer model* that will be introduced in section 4.7.3. In a real-time program, the Data Source is the component that actually receives the data from the input stream. In an offline program (or health assessment program) the Data Source is the component that accesses stored data (a file or a database) and converts the data for use by COPDA.

COPDA uses only one data source at a time (see section 4.7.3). This does not mean that COPDA can only process one signal at a time or can only process data from one resource simultaneously. It does mean however that combining and synchronising data from multiple resources (e.g. multiple files using different file formats) is a task that is not performed by the COPDA engine, but must be done by a data source component.

### 4.7.1.3 Data destinations

*Data Destinations* are components that use the data that was processed by COPDA. For example, a component in an offline program that stores processed data in a file is a data destination (it is the task of that data destination component to convert the data produced by COPDA to the desired file format). A component that collects processed data to show it in a graph viewer is a data destination as well.

A COPDA program can use multiple data destinations simultaneously without any restriction. For example, an offline program may want to store data in a file using one data destination
component and collect the same data for viewing in a 'graph viewer’ data destination component.

In section 4.7.5 two different categories of data destinations will be identified.

4.7.1.4 First steps in using COPDA to build a program

Some of the tasks of a programmer using COPDA in his program can be derived from figure 4.2.

The programmer must instantiate the COPDA engine component, instantiate a data source component and instantiate the desired data destination components. This means establishing the top three bullet-arrow lines in the figure. How each component should be instantiated depends on the component.

If no data source component exists for the resource the program wants to use for reading its data (e.g. no data source component exists supporting a certain file format), the programmer must first program a new data source component (the same holds for data destinations).

After instantiating the data source and data destination components, the programmer must register these components with the COPDA engine, so the engine can access them. This means establishing the bottom two bullet-arrow lines in the figure.

4.7.2 Signal model

The way COPDA treats values and signals is derived from the properties of the data acquired with the HPDATA system, as processing these data was the initial goal of COPDA. The properties of these data were summarised at the end of section 4.2.

4.7.2.1 Limitation to synchronous signals

For now, I assume that all signals to be processed are sampled at the same sample rate, which is true for the data acquired with the HPDATA data acquisition system. This assumption makes an elegant data processing paradigm possible: sequential data processing.

In sequential data processing, the same data processing operation is repeated at each sample time. At each sample time the following steps happen in this paradigm:

- The data source component is instructed to retrieve all sample values for that sample time (for real time data sources this means waiting until the data arrives).
- A data processing algorithm is applied to the samples. Such an algorithm can have access to a storage mechanism that allows accessing previous samples or previous results, such as is required for filtering operations.
- The results of the data processing algorithm are sent to the data destination component(s).

This paradigm is simple yet powerful. It has no need to make any distinction between data-push and data-pull style programs/applications; the task of making that distinction is delegated to the data source component.

The major drawback is that the sequential data processing paradigm fails when data in different signals is sampled at different rates or when a signal is present that is sampled at irregular intervals. However, a data source may be able to convert the multi-rate data or asynchronous data to a synchronously sampled model. For multi-rate data, signals may be
oversampled to a common data rate. This comes at a performance cost and is not possible if there is no least common multiplier of all sample rates (when the ratios between sample rates are not rational numbers). Alternatively, in some situations using a mixture of subsampling some signals and oversampling other signals can be acceptable. For asynchronous input signals, a sample and hold system can be used to synchronise the signal.

As COPDA initially was intended for processing the data acquired with HPDATA, in which all signals are synchronised to the same rate, I started evaluating the usability of the sequential data processing paradigm. As this proved to be successful, I did not evaluate other data processing paradigms.

4.7.2.2 Non-numerical data and the COPDA data type

Some of the signals acquired from the SDN system (chapter 3) carry string values instead of numerical values. In all signals (numerical signals and string signals) values can be an invalid marker instead of a number or string. It would be nice if COPDA were able to process such signals containing non-numerical values.

One way of supporting various signal types is defining separate operations, tailored for each of these signal types. A more powerful method is to define a unified data type that can carry an integer, a string or anything else that may be of use and treat every signal as a sequence of such values. A unified data type also makes defining the language for specifying COPDA data processing algorithms simpler, as will be shown in section 4.8.4.

For these reasons I chose to define a unified data type ('the COPDA data type'). Values of this unified data type can carry a single value of one of the following seven data types.

- invalid markers
- Integers
- Floating point numbers
- Text strings
- Boolean values
- Dates / timestamps
- (COM) Interfaces (IUnknown)

Of these seven values, the latter three types were included in the COPDA data type as they are useful as variable values in a language to specify data processing algorithms. Allowing the values to carry COM interfaces allows a COPDA value to carry complex data structures, such as components acting as 'arrays' or 'records'.

4.7.2.3 Data validity

Data samples recorded by the HPDATA system can have the value invalid. The interpretation of this information may be different on different occasions.

Several of the sensors used for measuring the patient's vital signs do not produce valid output under certain circumstances. For instance, the patient should not be moving too much for a reliable SpO₂ (blood oxygenation) measurement. Some of these sensors can detect these circumstances and output a sample marked as 'invalid' instead of outputting a number. In figure 4.3 a blood oxygen saturation signal (SpO₂, expressed in percents) that contains some 'invalid' samples is shown. Depending on the circumstances, this value can be interpreted as being truly invalid, or as unknown. In this thesis I refer to this special value as invalid.
Standard signal processing algorithms expect an uninterrupted stream of valid numeric samples and cannot process such signals with 'gaps' in them. (A 'gap' is a sequence of one or more invalid samples.) There are two solutions to work around this problem. Either one can try to fill in the gaps so standard signal processing algorithms can be applied or one can try to modify standard algorithms so they can work when samples are missing.

The first solution is to fill the gaps before applying algorithms (modifying the data). To do this some kind of interpolation of the missing samples has to be performed. Many methods exist to perform such an interpolation. These methods take one or more valid samples before and/or after the gap and generate a series of probable intermediate points. There is a caveat though. The gaps occur when the 'signal quality' drops below a sensor-dependent limit. Interpolating algorithms use the 'border' points around the gaps to predict substitute values for the missing samples. These border points generally are of lower signal quality than the rest of the signal, as they were measured when the 'signal quality' was already degrading, but still above the sensor threshold. The interpolated samples are thus based on lower quality data.

By their very nature, gap-interpolating methods erase the evidence that an input signal contained gaps (and hence erases evidence that the signal quality was lower for some samples). This complicates assessing the reliability of conclusions drawn on these interpolated signals. The data processing engineer who is designing data processing algorithms should be aware of this effect.

The second solution is to modify the algorithms in a way that allows them to process data with 'invalid' samples. Processing signals containing gaps with algorithms that 'understand' gaps is preferable over interpolating methods, as the information about the measurement quality is not destroyed. Not all standard algorithms can easily be converted to a 'gap-friendly' form. On the other hand, supporting 'invalid' samples in a signal makes some algorithms not commonly used in processing of one dimensional signals useful (e.g. 'erosion' of gaps: marking valid samples immediately before or after an invalid sample as invalid). As an example of a non-trivial conversion, appendix D contains a description of a first-order IIR low-pass filter that can handle gaps.

Simple rules can be stated for normal arithmetic operations for which one of the arguments is invalid. For many purposes simply defining the result of any operation involving invalid as invalid gives good results. In other occasions it is useful to be more careful. For example,
4.7 Data interface: input and output

Some algorithms can benefit from treating the result of multiplying zero with invalid as zero instead of invalid.

An example of a group of algorithms that can easily be converted to handle gaps in a signal are the order statistics filters (median filter, minimum filter, maximum filter). For each sample these algorithms build an array of the most recent $n$ input samples ($n$ is an algorithm parameter), sort this array and return a value from this sorted array as the filter's output (a median filter returns the median value of this array, a minimum filter returns the minimum, a maximum filter returns the maximum). Extending these algorithms to support invalid values in the incoming signal is easy: just skip adding a value to the array when building the array when that value is invalid. If the resulting array is empty, the filter returns invalid as its output, else the median of the stored values is returned. The result of applying such a four sample median filter is shown in figure 4.4. It is clearly visible that a side-effect of this filter is that smaller gaps are effectively interpolated away. Whether this effect is allowable or not is a choice of the user of the algorithm, not a choice enforced by COPDA. If this effect is undesirable, the user can choose not to use this filter at all, or to modify the filter by changing the interpretation of invalid in its filter.

4.7.3 Data transfer model

This subsection describes how the data transfer between I/O devices and the COPDA engine is modelled (the fat grey arrows in figure 4.2). I use the term I/O device to collectively address data sources and data destinations. Despite the differences between the two, there is a common base in the data transfer model that can be used by both.

In the sequential data processing paradigm, input signals provide a new sample synchronously and output signals transfer their sample values synchronously. This implies that at any moment during the processing of the data, it is valid to talk about the current sample value for each signal in an I/O device. Processing algorithms in the sequential data access paradigm in principle only need to access (read or write) these current sample values. If previous values are required in the algorithm (as is the case for nearly every conceivable filtering algorithm), that algorithm can store those values in its own variables when they were the current sample, and retrieve them during the processing of the later sample. In other words: when during the processing of a sample a sequential data processing algorithm needs
to access previous samples, making sure those previous values were stored somewhere is a task of the COPDA engine, not a task of the I/O device.

This means that the main task of an I/O device with respect to providing data is to provide access to the current sample of every signal. For data sources this means providing read access, for data destinations this means providing write access. For components that act as both a data source as well as a data destination, this may mean providing both read and write access.

Data sources and data destinations have other related tasks as well: preparing samples for use or finishing their use. A data source should have a method to prepare itself for each sample time (e.g., for file based data sources, this probably means loading the signal samples from the file and caching them). This method is used by COPDA programs before the data processing algorithm starts using the data for that sample time, reading the cached data from the data source. After the data processing algorithm finishes processing the data for one sample time, the data destinations are instructed to 'use' the data they buffered for each signal (e.g., for file based data destinations, this probably means writing the buffered signal values to the file).

For reading or writing data values, an I/O device can be viewed as a set of current data values that can be read from or written to. This is the essence of the COPDA data transfer model that is used to derive the COM interfaces used in the communication between the COPDA engine and I/O devices. Such a set can be implemented in different ways.

### 4.7.3.1 Data transfer model implementation options

![Figure 4.5 Data as a time sequence of abstract one-dimensional arrays.](image)

A possible implementation is to treat each I/O device as an abstract array of current sample values, each sample value being of the COPDA data type introduced in section 4.7.2.2. Each array entry contains the current sample value of one signal. See figure 4.5.

I use the word abstract array because an I/O device is accessed as an interface by COPDA, not as a conventional array. It is the design of that interface (how to map array indices to signals) that is the topic of this subsection. For an abstract array this interface should implement an array-like behaviour, that is: there is a number of entries in the 'array', indexed by numbers in a restricted range. The 'array' is accessed by Read and Write methods that are
4.7 Data interface: input and output

given the 'array index' as one of their arguments. Some object-oriented programming
languages (C++ and Delphi) actually allow using the conventional 'array' syntax in the use of
such interfaces (using operator overloading; in Delphi this is the only form of operator
overloading allowed).

For the data produced by the HPDATA system, this approach works quite well. Each signal
that can possibly be provided by that system already has a numerical code identifying that
signal that can be used as the array index: its MFC, see section 3.3.2. For example, SDN
defines the MFC of Heart Rate as 10; Using this system, 10 becomes the index of the heart
rate signal in the abstract signal array.

The only problem with this is the existence of signals that comprise multiple atomic values.
This occurs for some blood pressure signals. These contain separate values for systolic,
diastolic and mean blood pressures. There are several ways to support these triple-valued
signals (or more generally: multi-valued signals). Three of the options are:

- Change the COPDA data type to include support for triple values. The triple values
  then would become an atomic data type.
- Use the normal MFC as the index of the first value of the signal, use values outside
  the normal MFC range (1-63) as indices for the other two values.
- Do not use a one-dimensional abstract array to present the data values, but use a
  two-dimensional abstract array. Use the MFC of a signal as the first index and a code
  identifying which of the three values of that signal (systolic, diastolic or mean) to
  use as the second index (see figure 4.6).

The third option provides some extra benefits. The second dimension can be used to provide
additional signal properties for each array entry. Or it can be used to provide a signal in
different formats. For the flexibility this option offers I chose to use this model in COPDA.
A two-dimensional array can be viewed as a one-dimensional array of one-dimensional arrays. The nomenclature used in the rest of this thesis is related to this view: the two-dimensional array is treated as an array of multiple channels, each of which comprises an array of one or more sub-channels. A channel provides access to the current 'value' of a vector-valued signal. (In the case of signals that carry a single value, these vector values are one-dimensional.) A sub-channel provides access to a (scalar) element in these vector values. A sub-channel can therefore be viewed as the current value of a scalar-valued signal.

To clarify a potential source of confusion: this two-dimensionality has nothing to do with time - including time in this model would add a third dimension. When time is included, each channel is best considered to be a vector-valued signal, while each sub-channel is best considered to be a scalar-valued signal. However, the channels only provide access to the current sample of each of the signals involved.

As the arrays involved are abstract arrays, their sizes (and hence the range of valid indices) do not matter. An array entry at a position (channel and sub-channel combination) that is not defined by an I/O device is always read as the value invalid. Trying to write into such a void array entry is ignored.

This model of the 'abstract two-dimensional signal array' has a drawback: for data sources whose signals cannot easily be associated with numerical array indices it may be hard for the high-level user to be sure what signal is available in what channel. For instance, the situation may occur that for one data file the heart rate signal is made available on channel 1 by the data source component and the SpO2 signal on channel 2, while for a different input file these may be reversed. For the data recorded with the HPDATA system this situation does not occur, as the channel number is defined for each signal type (namely that signal's MFC). For the high-level user of COPDA it would be easier when the signals would be identified by a signal/channel name instead of being identified by a channel number.

To solve this problem I implemented a system where channels can be identified by a name. It is the task of the data source components and data destination components to 'map' these channel names to the channel number they use internally. For the high level users this means that they do not need to know at what channel number a certain signal is present. When they want to start using the SpO2 signal, they start reading data from the channel named “SP02”. If the data source does not provide an SpO2 signal (for instance because that signal was not recorded), the user can still read from the “SP02” signal, but the value returned will always be the invalid constant (see section 4.7.2.3).

### 4.7.4 Data sequencing model

This subsection describes the model COPDA uses to sequence the data it processes. This model not only describes how COPDA works together with data sources and data destinations to access subsequent data samples, but also how offline programs can process multiple batches of data. This low-level topic will have a profound influence on the high-level users, as it has an influence on the language used to specify the data processing operation (as will be shown in a later section in this chapter).

For simplicity's sake, most of this section will approach the problem from the viewpoint of a data-pull style program (in which the data source must be instructed to retrieve the next sample). Most conclusions will hold for data-push programs as well (in data-push programs, active retrieval of data by the data source is replaced by passive waiting for the next sample).
The basic idea of the COPDA sequencing model is implicit in the *sequential data processing paradigm* (section 4.7.2.1). It can be summarised in the pseudo-code algorithm of algorithm 4.1.

```plaintext
while not finished do
    begin
        tell the data source to load the next sample;
        tell the COPDA engine to process the sample;
        tell each data destination a sample was processed;
    end;
```

**algorithm 4.1** The simplest form of the data sequencing model

This algorithm is, however, too simple. It has no room for instructing the data source component where it should get its data from or instructing the data destinations what exactly to do with the processed data (e.g. specifying what file to store the results in). It also does not tell data destinations when processing has finished (e.g. to allow closing an output file).

The algorithm also does not explain what causes the *finished* function to conclude that processing has finished. This might be the result of the data source discovering that no more samples are available; it also might be the result of a user interface action or the data processing operation performed by the COPDA engine may abort further processing. To keep the model relatively simple, it is a good idea to only consider the 'normal' way of terminating the loop (the data source reaching the end of the input) and leaving the special cases as details for the implementation. (A suitable implementation of handling such non-standard loop termination causes is the use of *exceptions*, if they are supported by the language used for implementing the algorithm.)

Before suggesting an improved version of the algorithm, I will introduce a slightly different notation for the algorithms. As the algorithms work with several components, I will use a more object-oriented notation. To keep the notation short I will use the following abbreviations for the components involved: *engine* is the COPDA engine, *src* is the data source, *dest* is the set of all data destinations. So instead of writing *tell the data source to load the next sample* I will write *src.load_next_sample*. Any text following the marker `//` up to the end of the line is comment.

```plaintext
src.Initialise; // open input file
dest.Initialise; // open output file
engine.Initialise; // initialise processing algorithm
while not src.Finished do // search/wait for next sample
    begin
        src.Load Next Sample; // load sample from file
        dest.Prepare_For_Next_Sample;
        engine.Process_Sample; // execute processing algorithm
        dest.Finish_Sample; // save result to file
        src.Finish_Sample;
    end;
engine.Finalise; // provide a summary
dest.Finalise; // close output file
src.Finalise; // close input file
```

**algorithm 4.2** An improved form of the data sequencing model. The comments indicate actions that might be implemented by the method (these are examples that only apply to file-based processing).

An improved version of algorithm 4.1 is given in algorithm 4.2. The comments indicate example actions that could be performed by the method, but may be not applicable to every conceivable data source or data destination.
This algorithm allows each of the three components processing time before and after the processing of all samples (for initialisation and finalisation) and it allows processing time before and after processing each sample to the I/O devices.

I tried using algorithm 4.2 as the sequencing model in COPDA, but I discovered a missing feature when using it in the offline program I used as testing ground. To see the problem it is necessary to think about how offline programs will process data. One of the ways offline programs use data is to run one data processing algorithm on the data for one patient. For this case algorithm 4.2 works perfectly. However, often the user wants to apply the data processing algorithm consecutively on the data of several patients. Or more generically: the user wants to apply the data processing algorithm consecutively to several independent sequences of data. Each sequence comprises a series of equidistantly spaced samples. To include the concept of sequences in the data sequencing model I extended algorithm 4.2 to algorithm 4.3 by including an extra loop.

```
engine.StartUp; // initialise processing
src.Configure;  // load input specification, defining
                // what sequences to process
dest.Configure;
while not src.Finished do
begin
  src.Init_Sequence; // open next input file
  dest.Init_Sequence; // open output file
  engine.Initialise;  // initialise processing algorithm
                      // for next sequence
  while not src.EndOfSequence do
    begin
      src.Load_Next_Sample; // load sample from file
      dest.Prepare_For_Next_Sample;
      engine.Operate;       // execute processing algorithm
      dest.Finish_Sample;   // save result to file
      src.Finish_Sample;
    end;
  engine.Finalise;
  dest.Finish_Sequence; // close output file
  src.Finish_Sequence;  // close input file
end;
engine.ShutDown; // provide total processing summary
```

Algorithm 4.3 The data sequencing model (the execution loop). This is the loop that a COPDA-based application should implement. Once again, the comments are only examples of actions that might be implemented by the method (these examples apply to file-based processing).

This algorithm is nearly a straightforward extension of algorithm 4.2. There are five places in this algorithm where the COPDA engine and its data processing algorithm can perform an action: the methods StartUp, Initialise, Operate, Finalise and ShutDown. These methods relate to the CPMDL language used to specify the operation to be performed by the COPDA engine. Section 4.8.4.1 will provide the details of this topic as seen from the high-level view. These five methods are the entry points where the low-level program developer invokes execution of the high-level CPMDL language, and as such are at the very heart of the functionality provided by COPDA.

---

16 Note that 'sample' is used here not in the sense of a single scalar value, but to indicate the set of all sample values measured at one point in time - the collection of the values in all available channels.
Algorithm 4.3 describes what each of the components (data source, data destinations and COPDA engine) must be able to do, viewed in the light of sequencing the data. As such, it defines (some of) the interfaces that are to be implemented by these components. When reviewing the details of the definitions of the related interfaces (in Appendix B), the methods named in algorithm 4.3 can be recognised in the interfaces IScriptEngine, IInputDevice and IOutputDevice.

4.7.5 Asynchronous output

In algorithm 4.3, the data destinations are used according to the sequential data processing paradigm: at each sample time a value is written to each channel. For some types of output this access style models exactly how the data destination uses the data. For example, this approach exactly matches the style used by data destination components that write a block containing all signal values at each sample time. In general, any data destination that expects data at each sample time can use this output model.

For some other types of output, however, this output model is not suitable: for example, occasionally providing a textual message as a result of data processing (e.g. when a certain pattern has been recognised by a pattern recognition algorithm). In general, output that is not produced synchronously with the input samples conflicts with the sequential data processing paradigm.

Sometimes it is still possible to use the sequential data processing model for this type of output. When during each sample time at most one output is produced and that output never has the ‘invalid’ value, the output can be modelled as a synchronised sample stream. At sample times when an output was generated, that generated value is used as the synchronised output value; for sample times at which no output value was generated, invalid is used as the synchronised value. The data destination receiving this synchronised stream from the COPDA engine can then ‘decode’ this stream by only using the non-invalid values.

This synchronising of asynchronous output is trying to find a way around the restriction imposed by the sequential data processing paradigm. There is a more elegant way to implement this kind of asynchronous output: by treating them not as a (synchronised) data destination but as an entirely separate class of output.

This leads to a modification of COPDA’s input/output model as presented in section 4.7.1 and figure 4.2. This modification is depicted in figure 4.7, introducing asynchronous outputs into the model. Asynchronous output components do not get their data from the synchronous data streams (the fat grey arrows in the figure), but get their data in the form of interface calls from the COPDA engine. When COPDA wants to produce output on an asynchronous output component, it calls a method of the interface of that component. Asynchronous output components are used for a wide range of purposes. For this reason, I decided not to provide a standard interface to be used for all asynchronous outputs in COPDA (unlike the data source and data destination standards for synchronous I/O). Instead, asynchronous output components are made available to high-level (data-processing) programmers via COPDA’s extension library mechanism, which will be detailed in section 4.9.
4.7.5.1 The COPDA console

A typical example of an asynchronous output component is a text window that displays messages generated by COPDA (either by COPDA itself or by the CPMDL code it executes). Such a window is commonly called a console. Having a standard 'place' to send messages that are generated during data processing is highly desirable for a system like COPDA. One way to provide such a console is to implement its user interface and include it in COPDA. However, this conflicts with one of the design goals of COPDA: independence of user interface. A better solution is to require the (low-level) program designer to implement the console functionality in his user interface and let COPDA access this functionality via a COM interface (thus treating the console as a component).

This is exactly what COPDA does: the COPDA engine component expects to be provided by the program developer with a component implementing the IConsole interface (see Appendix B). What this console component does with the text messages sent to it by COPDA is entirely at the discretion of the program developer. For an offline program, the console is most likely a console in the classical sense: a window that shows textual messages. For a real-time program the 'console' can for instance be a component that pops up a dialog box when a message is written or a component that logs the message in a report.

4.8 Data processing

This subsection will introduce the data processing facilities provided by COPDA as seen from the high-level COPDA user (the data processing algorithm developer). First I will take a closer look at these users and identify two different groups. Then I will introduce the CPMDL language that is used to specify data processing operations to COPDA.
4.8.1 High-level users

In section 4.5 the users of COPDA were classified into two groups: 'low-level' programmers that use the COPDA components in their programs (program developers) and 'high-level' users that specify the data processing algorithms that COPDA should execute (application developers). The latter group includes users of offline programs and developers of real-time COPDA applications. The tools used to develop data processing algorithms for real-time applications are offline programs (simulating a real-time environment to test algorithms). In other words: all high-level COPDA users are users of offline COPDA programs.

However, amongst users of offline programs, different sub-groups can be identified. Each of these groups of users has different requirements for ease-of-use and flexibility.

1. Medical researchers that only need standard signal processing operations. Many data processing algorithms can be described by building a network of standard signal processing operations. In such a network, the output of one operation is used as an input to another operation. For an example, see figure 4.8. In the context of COPDA such signal processing operations are called modules. Examples of modules include filters, threshold detectors and modules that combine signals (e.g. by calculating the difference).

I will refer to this group of users as module users.

2. Advanced researchers that need more control and may need non-standard or customised processing operations. A network of modules can often still be used to describe the tasks that this group of users wants to perform, but the actions performed by some of these modules are no longer standard signal processing tasks but customised algorithms. The needs of this group of users are catered for when the functionality provided by modules can be customised. Often the algorithm to be executed by such customised modules can be described in a language that is more powerful than being limited to a few standard signal processing algorithms but does not need to be as powerful as a generic programming language. This is exactly the role played by the CPMDL language; CPMDL is the acronym for COPDA Module Description Language.

I will refer to this group of users as CPMDL users.

3. Users for whom the flexibility provided by CPMDL is not sufficient, either because of performance criteria or because functionality is required that CPMDL does not provide. These users can extend CPMDL with new functions via the extension mechanism provided by COPDA. This task involves using a conventional language to develop a library component that COPDA can load dynamically.

I will refer to this group of users as extension programmers.

In the sense of section 4.5, users in this group are both high-level users and low-level programmers at the same time, forming a bridge between the two groups. Because their task is building primitives for data processing tasks, extension programmers can be considered high-level programmers. For their use of a conventional programming language as their tool, extension programmers should be considered low-level programmers as well.

The rest of this section will focus on the needs of the first two user groups, modules and CPMDL. COPDA extensions and CPMDL extensions are discussed in section 4.9.
4.8.2 Introduction to modules

Data processing algorithms can usually be described as a set of building blocks (modules) connected in a network. For an example, see figure 4.8. Sometimes only modules implementing standard signal processing algorithms are required to describe a full data processing algorithm. On other occasions a data processing algorithm can be described by a network comprising several ‘standard’ modules and one or more ‘custom’ modules.

![Figure 4.8](image)

Figure 4.8 An example data processing algorithm. In this case an input signal is first median filtered (noise removal), next a ‘trend’ is calculated and produced as the algorithm’s output. The trend is calculated by taking the difference of the intermediate signal and a delayed version of that same intermediate signal.

In COPDA, modules are processing blocks that apply a data processing algorithm to one or more input signals and produce one output signal. (The choice for limiting the number of module outputs to one is explained in section 4.8.3) The algorithm performed by a module is executed once at each sample time, using the current value of each input signal to calculate one new value for the output signal. This description assumes that all signals are synchronised (sampled at the same sample times). If signals weren’t synchronised the unambiguous interpretation of the phrase ‘at each sample time’ would not be possible.

Some data processing algorithms cannot (easily) be described by such a network of modules. Typically these are algorithms that cannot easily be expressed in the sequential data processing paradigm. However, the same reasons that make them not easily expressed in this paradigm also make uniform treatment of offline and real-time processing difficult. As this uniform treatment is a design goal of COPDA, leaving out support for this category of problematic algorithms in COPDA (by restricting COPDA to algorithms described as a network of synchronous modules) is justified.

The data processing algorithm to be performed by a COPDA program is specified as a network of synchronous modules. A data processing algorithm specification for COPDA comprises a description of a set of modules and the description of the connections between these modules. For this reason the algorithm specification is called a module network.

This module network paradigm has its influence on how the first two user groups identified in the previous section may look at the algorithm descriptions. For the first user group, the module users, the algorithm description consists of a set of predefined modules and their interconnections. For the second group, the CPMDL users, the algorithm description still consists of a set of modules, but the (sub-)algorithm to be executed by each module is customisable.

The task module users have to perform when implementing a data processing algorithm is to create a set of modules and connect the inputs of each module to an output of another module. The task for CPMDL users includes the same task, but additionally requires
specification of the algorithm to be performed by each non-standard module. This latter action requires an amount of flexibility that only can be provided by a (textual) programming language – CPMDL.

The first task however is ideally suited for a graphical interface: modules and their interconnections are easily represented graphically. Such a graphical interface must provide a way to create new modules (both new instances of standard modules as well as blank modules for customisation) and a way to connect module inputs to module outputs. Since COPDA does not provide any user-interface related functionality, it does not provide a graphical editor to construct module networks – creating such an editor is the task of the programmer of a COPDA program that provides support for editing module networks. COPDA does provide support for such programs. First, the CPMDL description of modules provides room for storage of some graphical parameters (such as a module’s position and size). Second, COPDA provides a mechanism that allows easy synchronisation of a graphical representation of a module (in the user interface, outside COPDA) with the component used internally by COPDA to represent modules (see the description of IBaseModule, IAttach and IModule in Appendix B).

4.8.3 CPMDL design

Before devising a new language to specify module networks, it is advisable to look around if any known languages are available to represent such module networks. I was not able to find a serious candidate language. So I devised a new language, CPMDL, to describe the algorithms (module networks) to be executed by COPDA.

This section will analyse the requirements for the CPMDL language.

The module network to be executed by COPDA is described in the COPDA Module Description Language (CPMDL). CPMDL must be able to express two different types of information:

1. What modules are contained in the module network and how they are ‘connected’ (that is: where does each of the module inputs get its signal from).

2. For each non-standard module, describe the algorithm to be executed by that module.

In the above paragraphs I used the concept of standard modules a few times: modules that implement ‘standard’ signal processing algorithms. However, it is hard to define which algorithms are ‘standard’ and which algorithms are ‘non-standard’.

One way to approach this is to let COPDA provide a set of ‘standard’ modules and define ‘standard’ as whatever COPDA would provide as standard modules. A kind of module library mechanism could allow tailoring the set of ‘standard’ modules.

A radically different approach is to let COPDA not provide any ‘standard’ modules at all. This removes the distinction between ‘standard’ and ‘non-standard’ modules. In this case CPMDL must provide a description of the algorithm to be executed for each module in a module network. Instead of letting COPDA provide a set of ‘standard’ modules, a COPDA program used to design CPMDL module networks can maintain its own set of ‘standard’ modules in the form of module templates. Such a module template is a full description of a single unconnected module that can be copied into the CPMDL module network that is being edited.

COPDA uses this second approach: the concept of ‘standard’ modules is not used; the algorithm executed by each module used must be completely described in CPMDL instead.
When a COPDA program wants to provide 'standard' modules to its users, that program has
to define its own mechanism for doing so. As will be shown in the chapter 5, the offline
COPDA program called DataWand uses module templates to provide a set of 'standard'
modules.

A module network description comprises a description of several modules and their
interconnections. The interconnections between the modules define for each of a module’s
inputs to which module’s output it is connected. When the number of outputs a module can
have is restricted to one (that is: each module produces exactly one signal, not more), any
output in the module network is uniquely identified by its module; in this case it is valid to
talk about the output of a module. In this case interconnections between modules are
completely determined when for each of a module’s inputs the module it is connected to is
specified.

I fixed this maximum number of outputs of a module to one in an early stage of the
COPDA/CPMDL design process. For many modules this is not a problem, as data processing
primitives (the algorithms that are implemented in modules) typically produce only one
output signal. Only when large parts of COPDA were already implemented, I realised that for
several algorithms having multiple module outputs can be useful. For example, when
calculating the median over a number of samples, it is trivial to produce the maximum and
minimum of those same samples as a side-effect of the calculation. Allowing modules to
have more than one output would have required major redesigns at that time. Instead I
developed a system that allows modules to have 'secondary' outputs that are accessed via a
module’s normal output.

Summarising: A module network is specified by describing each of the modules in it. The
module descriptions include for each of the module’s inputs the name of the module it is
connected to. So a separate specification of the interconnections between modules is no
longer required.

4.8.3.1 Requirements
The specification of a single module requires the elements listed below. Some of these
elements were already introduced above, other elements will be introduced later.

- Module ‘layout’ information
  - A unique module name (allowing to reference the module by its name)
  - The number of inputs of the module (zero or more)
  - For each input:
    - The name of the input itself (used to distinguish the various inputs)
    - The name of the module the input is connected to (or none for inputs that are
      not yet connected)
  - The number of outputs of the module (zero or one)
  - Additional information about the module for use by graphical user interfaces:
    position, size, a short description of the module
- Definition of the algorithm to be performed.
- Declaration of the local variables used by the module’s algorithm.
- Declaration of any COPDA extensions required for executing the algorithm

There are many possible ways to express this information. A designer could try several of
these approaches and carefully evaluate each of them, probably to discover that there is no
'best' way to express the information. I took the more practical approach: trying a very few
options, improving when certain points 'feel' not good. In essence CPMDL was 'designed' by evolution (while designing the DataWand program described in section 5.1).

### 4.8.4 An introduction to CPMDL

This and the following sub-sections provide an introductory description of the CPMDL language. For a complete description see Appendix C. An example of a CPMDL module description is given in section 4.8.5.

#### 4.8.4.1 Sections

The CPMDL language describes module networks. As described above, this entails only describing the individual modules in the network. So a CPMDL description consists of nothing but module descriptions. There are no features like 'global variables' that have to be declared outside any module definition.

Each module description comprises one to eight sections. The different sections in essence describe the different points listed in the previous subsection. There is one section (the header section) that conveys all of the 'module layout' information, one section (the uses section) that defines requirements for COPDA extensions and one section (the declare section) that contains local variable declarations. The remaining five sections are the executable sections that describe the actual algorithm to be performed by the module. Only the header section of a module is required, other sections can be omitted. For each of the non-header sections there can be reasons to be empty. Leaving all of them empty (a module with only a header) does not make sense though, but allowing existence of such useless modules is simpler to implement than disallowing them.

For the reason to have five different executable sections I refer back to section 4.7.4 and algorithm 4.3 (the data sequencing model). In algorithm 4.3 the COPDA engine component is invoked for five different purposes: once before any processing starts (startup), once when each new sequence starts (initialise), once for each sample in that sequence (operate), once at the end of each sequence (finalise) and once when all processing is finished (shutdown).

During each of these calls the COPDA engine lets all modules execute their algorithms. So the algorithm to be performed by a module is split in five parts, according to the five purposes mentioned above. Each of these parts is described in its own executable section. Not surprisingly, the names of the five executable sections are: startup section, initialisation section, operation section, finalisation section and shutdown section.

The syntax used inside each of the sections is not the same for all sections (though all executable sections use the same syntax). In this respect CPMDL can be considered a layering of different languages. At the top level, CPMDL defines a syntax that defines how to split the CPMDL source code in the descriptions of separate modules and how to split each of those descriptions in separate section descriptions. Each of the sections can then be parsed according to its own syntax rules. This two-layer approach allows for robust error recovery and allows parsing of sections when other sections in the same module contain errors. This feature of CPMDL has proven to be useful in developing algorithms described in CPMDL.

The syntax of CPMDL is specified in one single syntax specification, see Appendix C. The top-level syntax is simple: each section starts with one of the eight keywords (module, uses, declare, startup, initialisation, operation, finalisation or shutdown) which must be placed at the start of a line. There is no explicit end-of-section indication. A section ends when the next section starts or when the end of the file has been reached.
There are two properties of this top-level syntax that are atypical for the rest of CPMDL: the use of keywords and the dependency on location (the keyword must start at the beginning of a line). The rest of CPMDL does not use any keywords at all (though statement names like ‘if’ and ‘while’ may be considered keywords as they influence the surrounding syntax). There are only ten true keywords in the language: the eight provided above and initialization and finalization, which are synonyms for initialisation and finalisation for use by programmers that prefer US-English to UK-English. Furthermore, CPMDL in general is a free-format language, meaning that the exact placement of spaces and line-breaks is not part of the syntax of the language. There are only two exceptions to this rule: the situation noticed above (the keywords that start a new section must be the first word on a line) and comments: comments in CPMDL start with an exclamation mark (‘!’) and end at the end of the line.

For a complete CPMDL syntax description, see Appendix C. I will highlight some general issues related to identifier naming and expand a bit on the content of the executable sections.

Identifiers are used in several places in CPMDL (like any other language): variable names, input names, function names, etc. Like other languages, identifiers in CPMDL normally consist of a letter followed by zero or more alphanumeric characters. However, the names used for module inputs and for variables have an extra requirement: variable names are identifiers prefixed with a ‘$’ character, input names are identifiers prefixed with a ‘#’ character. In early versions of COPDA, these prefixes made the CPMDL parser simpler. I decided not to remove the requirement for these prefixes in later, more advanced versions of the parser for the following reasons:

- The prefixes make it easier for humans to recognise a variable or input name in CPMDL code (differentiating it from an argument-less function call).
- The prefixes prevent naming conflicts with function names.
- The prefixes often allow displaying more sensible error messages in case of syntax errors.
- Compatibility with existing CPMDL scripts.

4.8.4.2 Executable sections

The five executable sections of a CPMDL module description contain the ‘executable code’ for the algorithm implemented by the module. The syntax used inside these sections was designed to be simple yet powerful. The design of this syntax was influenced by my knowledge of several other languages, especially C, (Object) Pascal and LISP.

The description of an algorithm comprises a block of statements. Like C, CPMDL does not differentiate between functions and procedures; any ‘subroutine’ is a function (the return value of which may be ignored). A statement in CPMDL is either a function call (including method calls) or one of a set of conditional statements (like a ‘while’ loop or an ‘if’ statement). As will be explained in the following section there are no separate assignment statements in CPMDL (assignments are performed via method calls). Most parts of COPDA treat conditional statements as function calls, although the syntax for conditional statements is slightly different. Conditional statements have one or more blocks of statements as one of their ‘arguments’ (enclosed in curly brackets). Function calls that are used as statements are terminated by a semicolon (like the C language).

CPMDL can only use functions that are defined in COPDA or in any COPDA extensions (‘libraries’). CPMDL does not provide features to define new functions in CPMDL itself. I decided that this capability is not necessary, as the concept of modules itself provides a way to modularise a data processing algorithm.
4.8 Data processing

To keep the parser (and the structure of extensions to the language) simple, expressions in CPMDL are kept simple: the only two types of expressions allowed are function calls and constants. The concept of function calls in CPMDL is rather broad though: it includes accessing the value of a variable and any method calls on variables (as will be shown in section 4.8.4.3, variables are treated in a pseudo-object-oriented fashion). However, this means that all expressions use a prefix notation. There are no ‘operators’ (infix notation) in CPMDL.

Function calls use a variation of the more or less standard syntax used in languages like Pascal and C. A function call is denoted in CPMDL as the name of the function followed by a list of arguments enclosed in parenthesis. Each argument is again a function call or a constant. Functions that do not require any arguments use the Pascal notation for functions without arguments: not only the list of arguments is empty, but the parentheses that normally surround the argument list are omitted as well. As there are no infix operators in CPMDL, there is no need for an argument separator. Arguments in CPMDL are written after each other, separated only by white-space (there are no commas in between the arguments). Each function call returns a single value (of the uniform data type introduced in section 4.7.2.2).

4.8.4.3 Variables

Variables are treated in an unconventional way in CPMDL. Variables are treated like ‘pseudo-objects’: each variable has a ‘class’ that determines a set of ‘methods’ that can be used on the variable. The class of a variable is specified when the variable is declared in the declare section of the module. I use the term pseudo-objects instead of simply objects to indicate that new classes cannot be defined in CPMDL. New variable classes can be defined however externally and be made available from COPDA libraries.

The ‘var’ class of CPMDL variables provides the features that a programmer would expect from a normal variable: values can be assigned to it and its value can be retrieved. However, the concept of variables used in COPDA is more powerful than just storing and retrieving values. Each variable class provides a distinct behaviour; storing and retrieving values is only one such behaviour. For example, variables of the dump class provide the capability to use text files. This class provides methods to open and close a file and to write messages to that file.

Variables are declared in the declare section of a module. The syntax is similar to variable declarations in the C language. First the type of the variable to be declared is listed (that is: the name of a variable class). The class name is followed by a list of one or more variable names, terminated by a semicolon. For some variable classes it makes sense to initialise a variable with a value as soon as it is declared. This can be accomplished by writing an ‘=’ sign and a string containing the initial value after a variable name. This feature is extensively used with a special group of classes called properties, which are introduced in section 4.8.5.

Normally, all variables are declared in the declare section of a module. There is one exception to this rule. In CPMDL, a module’s input names (the labels given to the inputs of a module) are also treated as variables.17 As input names by definition are already declared in the header section, there is no need to re-declare them elsewhere. Treating inputs as objects allows the definition of functions (methods of the input class, to be more precise) that provide functionality specific to module inputs. As each module input is connected to the output of exactly one other module, this includes functionality specific to another module’s output. For

---

17 In the CPMDL documentation these variables are referred to as signals and their class is called signal. To prevent confusion with other uses of the word signal I do not use these terms in this text.
example, because the outputs of COPDA modules are the entities that are responsible for storing previously calculated samples, input names provide a method for accessing previous output values of the connected module.

Before a module starts accessing previous values stored in another module’s output buffer, it first uses an input name method to specify the number of samples that should be ‘remembered’. This method can only increase the output buffer size, otherwise incorrect behaviour may occur when multiple modules are connected to one output. To explain this, consider the situation where modules B and C both have an input that is connected to the output of module A. Module B first requests module A to store the five most recent output values (of module A). Next, module C requests module A to store the two most recent output values. A subsequent call of module B to module A to access the output value of four samples ago would fail if module A had responded to module C’s request by decreasing the buffer size from five to two. Correct behaviour is ensured when module A ignores any request to decrease its buffer size.

There is no need for a variable class that accesses the output of the current module. Instead, CPMDL uses the following model to set the output value of a module (made possible by the fact that a module can have at most one output): the output value of a module is the return value of the last statement in the operation section of a module. For this mechanism to work correctly, every statement (both functions/methods and control statements) must return a value. Control statements (such as while) return the value returned by the last statement that was executed in their statement block, or they return invalid when no statement in any of their statement blocks was executed.

4.8.4.4 Example 1

To clarify several of the concepts described above, an example may be helpful. The module definition below describes a module called ‘Delta’. It has one input, named ‘#in’, which is connected to another module called ‘Other’ (remember that input names always start with a ‘#’ character). It performs part of the algorithm depicted in figure 4.8: it calculates the difference between the current value of the module input and the previous value. Two different implementations of this task are provided. The first implementation (algorithm 4.4) uses the fact that module outputs store previous samples the module produced. The second implementation (algorithm 4.5) uses a variable to store the previous value of the input.

```
module Delta-l(#in=Other) @(0 0 65 97) "delta module"
initialization
   #in.require(2);
   ! Make sure the other module stores at least two samples
   ! (the current and the previous) In this case it doesn’t
   ! matter whether this statement appears in the
   ! initialization or startup sections
operation
   sub(#in #in.delay(l));
   ! Subtract the previous sample (#in.delay(l)) from the current
   ! one (#in), which could equivalently have been written as
   ! #in.delay(0)). This is the last statement in the operation
   ! section, so the result of the calculation is the output value
   ! for the module. Note the usage of the 'sub' function to
   ! subtract two values; there are no operators (such as '-') in
   ! CPMDL, all operations are written in prefix (functional)
   ! notation.
```

**algorithm 4.4** CPMDL example 1: a module that outputs the difference between the current input sample and its previous value.
module Delta-1(#in=Other) @(0 0 65 97) "delta module"
declare
  var $previous $res;
  ! Declare variables of the 'var' class: an ordinary
  ! variable in the classic sense, used for storing values.
initialization
  $previous.set(invalid);
  ! Make sure results from a previous sequence do not
  ! influence the current one. (invalid is a function
  ! that always returns the value 'invalid').
operation
  $res.set(sub(#in $previous));
  ! Subtract the previous sample from the current one.
  ! The result is stored in the variable '$res' and
  ! variable assignment is no exception to that rule.
  ! The first time this statement is called, $previous
  ! has the value 'invalid'. The 'sub' function returns
  ! 'invalid' when any of its two arguments is 'invalid',
  ! so the first output this module produces will be
  ! 'invalid'.
  $previous.set(#in);
  ! Remember the current sample value.
  $res;
  ! Set the output value for the module.

algorithm 4.5 CPMDL example 2. Performs the same operation as example 1, but in a
different way (using variables instead of using the methods of module
inputs).

4.8.4.5 Input and output

A module can have at most one output. Most modules will have exactly one output, though
some modules will have none. Modules without an output are referred to as sink modules.

On the other side, modules may have zero or more inputs. Modules without any inputs are
referred to as source modules. An example of a source module is a module that acts as a sine
wave generator: it produces a fixed frequency, fixed amplitude sine wave and doesn’t need an
input signal to do so.

In COPDA, source modules and sink modules play an important role. It may sound confusing
at first, but modules with no inputs are used for input and modules with no output are used
for output. The confusion is caused by the fact that the word input is used in two different
meanings (and similarly the word output): inputs of a module (essentially: outputs of another
module) and inputs to the entire module network (the external signals that are to be processed
by the network).

To clarify this statement, it is relevant to realise again what the role of a module network is:
to describe a complete data processing algorithm that is applied to one or more input signals
and results in one or more output signals. Up till now I skipped the subject of how the
algorithm specifies which signals are to be processed and how it specifies what to do with the
results. Section 4.7.3.1 described the model that is used to represent signals in COPDA as
seen from the viewpoint of a low-level programmer: they are presented to the COPDA engine
(and hence to CPMDL) as elements in a virtual two-dimensional array of signals. Each signal
provided by a data source component is uniquely identified by a channel number\(^{18}\) and a sub-
channel index. For a high-level user, CPMDL provides access to signals using the same
model. The current value of a signal is accessible via two CPMDL functions: the value
function is used to read the current value of a specified input channel and the write function is

\(^{18}\) In the most recent builds of COPDA, channels can be identified by name.
used to write the new value of the specified output channel (input and output channels are specified by a channel name/number and a sub-channel index).

Normally, a CPMDL development environment (an offline COPDA program) will provide its user with a standard module that encapsulates the value function and a standard module that encapsulates the write function. That first standard module has no module inputs and produces the value of a single signal on its output. It is the module that encapsulates the concept of an input signal to the algorithm implemented in the entire module network. Conversely, the second standard module has no need for a module output to be used by other modules as an input.

Summarising: Module inputs and outputs are not a model for data transfer with the outside world, only a model for data transfer in between modules. Modules can have other, 'hidden', inputs and outputs for communicating with the outside world. In CPMDL the 'hidden' inputs and outputs are available as functions (or methods). The value and write functions mentioned above are examples of such a 'hidden' input and output. As the data transfer between modules only conveys synchronous data, asynchronous output is only possible using 'hidden' outputs. An example of this is the CPMDL message function, which is used to print messages on the COPDA console (see section 4.7.5.1).

4.8.5 Module properties

The first category of users identified in section 4.8.1 specify a data processing algorithm by building a network of pre-programmed modules. If the modules would not provide any form of flexibility, this would not be a usable solution. For example, when the designer wants a module that implements a median filter over seven samples, but only median filters over three, five and nine samples are available, the designer is out of luck. It would be much more useful to have a generic median filter module for which the number of samples it operates on was configurable. Or to put it more general: it would be useful for modules to be able to define a set of properties for each module. Such properties are values that can be changed by the user of the module and are used to change some details of the algorithm implemented by the module.

COPDA supports the use of properties in modules. The topic of module properties influences each of the four groups of COPDA users identified so far (low-level programmers, extension programmers, CPMDL programmers and module users). The influence for module users is obvious: they gain the ability to modify properties. To do this some kind of user interface must be available to change these values. It is the task of the low-level programmer (who builds the COPDA program) to provide this user interface.

The basic role of the CPMDL programmer is clear as well: he must define what properties are available for a module. Properties are defined in CPMDL as variables of a special class. More specific: a variable of one of a group of classes, called the parameter classes. Inside the executable sections property variables act as constants: these classes have no methods, only their actual values can be used. There are multiple parameter classes. Each type of parameter class is used to store parameters of a different type. For example, the class named parameter is used to store string-valued properties, while the class named intparam is used to store integer properties. The value that is stored by a parameter is defined by the initialising string that appears in the variable declaration (see section 4.8.4.3).

Properties provide some features that may not be obvious at first sight: type checking and custom editors.
Each parameter class can define which strings it accepts as valid values. A module user trying to specify an invalid value for a parameter will get an error message. More precisely: it is the low-level programmer who has to deal with the error message from the COPDA engine component when he tries to use that component’s method for modifying a module property. The low-level programmer should show the error message to the module user. It is still possible to specify an invalid property value in the CPMDL declaration of the property. In this case however the COPDA engine will refuse to execute the module set, treating the invalid value as a CPMDL syntax error. What a ‘valid’ value is and what not depends on the class of the property. It is the task of the programmer of this class (an extension programmer) to define this.

Properties can be used for any data that can be represented as a string (which, in essence, is anything). For some classes such a representation may be difficult to understand and hence difficult to edit. Even if a string representation of a parameter is not difficult to understand, editing a property value using a different mechanism than typing in a text string can be appropriate. A good example is for instance a property representing a file name. For such properties a parameter class can define a custom editor, a dialog box showing a custom user interface for that parameter. In the case of file name properties, such an editor would show a standard Windows ‘open file’ dialog box. How these custom editors are accessed in the COPDA program’s user interface is a decision that is up to the designer of that program (the low-level programmer).

4.8.6 Module networks in the COPDA engine

This subsection gives a short impression of how COPDA uses module networks.

The goal of COPDA is to execute module networks. To do so, a COPDA program loads a module network description from a file into the COPDA engine and applies algorithm 4.3. This view simplifies certain aspects. More importantly, it hides some of the features provided by COPDA.

The fact that is missed in this simplified description is that the loading of the modules is a two-stage process.

In the first stage, the loading stage, the module network is loaded from a file into the COPDA engine. During this stage COPDA only partially parses the CPMDL code: it divides the module network description in separate modules, and each of these modules in its separate sections. It tries to parse the uses sections, (loading any required libraries; see section 4.9.3), and it tries to parse the declare sections of all modules, determining which properties each module has. The text of executable sections of the modules is stored in the COPDA engine without being parsed.

In the second stage, the run stage, the module network is actually executed. This stage is initiated by COPDA parsing the executable sections into an internal data structure (parse trees) that can be directly executed. Then the COPDA program can perform algorithm 4.3 to do the actual data processing. After processing, COPDA releases the memory used for the internal data structures.

In the loaded state (after the first stage) COPDA stores all module information in memory. In this state properties can be changed and executable sections can be edited (using any kind of text editing interface). After modifications have been made the module network can be written back to a CPMDL file. This allows COPDA to be used as the core of a CPMDL editing tool or to include editing support in offline COPDA programs: the editing tool itself
doesn’t need to know much about CPMDL; all saving, loading and interpreting of CPMDL is
done by COPDA itself. Editing the module network is also made simple: a new module is
added to the network by loading a CPMDL description of the new module; the new module is
automatically added to the existing module network. The COPDA engine provides methods
(for low-level programmers) to create and modify the connections between modules. A
COPDA program can retrieve the text of any module section, let the user edit it, and store it
back.

4.9 Extensions

A system like COPDA can never be complete. To be usable for a long time, COPDA must
define a mechanism that allows its functionality to be extended. There are two different
categories of COPDA features that need to be flexible: on one side data input and output, and
on the other side CPMDL functionality. This section introduces the two mechanisms to
provide new features to COPDA and CPMDL.

4.9.1 Extending input and output capabilities

The basic mechanism used to extend COPDA’s input and output capabilities follows from the
content of section 4.7.1. COPDA can process new types of (synchronous) data input when the
program’s data source component is replaced by another one. To produce new formats of
synchronous data output, add a new data destination component. The only requirement on
these components is that they implement the correct COM interfaces, as described in
Appendix B.

Adding new asynchronous outputs is equivalent to adding a new function to CPMDL (as
explained in section 4.8.4.5). This is the topic of the next section.

4.9.2 Extending CPMDL

The task of CPMDL extensions is to extend the functionality provided by CPMDL without
having to modify (and recompile) COPDA. The standard solution to make any program
extensible without having to recompile it, is the use of DLLs. In COM this means
programming a component and making it available in a DLL. This DLL can then be found by
any COPDA program or COPDA application that wants to use it via a lookup in the
computer’s registry database. In the context of COPDA such DLLs are called COPDA
libraries.

There are two aspects of CPMDL that can be extended by using libraries: new functions and
new variable classes. Defining a new variable class includes the definition of zero or more
methods for that class. In the rest of this sub-section I will describe what ‘defining a new
function’ does to the COPDA engine and in the next sub-section I will describe the COPDA
library mechanism that is used to implement this.

All of the items that reside inside the COPDA engine, such as functions, classes, methods,
but also modules and a few other things are stored in the following way. All of these objects
are actually COM components. The COPDA engine component uses an associative array\(^\text{19}\) to
store all these components. From a normal (non-COM) object-oriented programming

\[^{19}\text{An associative array is an array whose index is not a number but a string; it associates a string with a ‘value’ in the same sense that normal arrays associate a number with a ‘value’}.\]
viewpoint, storing different kinds of objects in the same data structure is only possible when all of these objects have a common ancestor. The fact that any component can be stored in such an array is made possible because every COM component is required to implement the IUnknown interface.

The associative array in the COPDA engine component thus stores string-IUnknown pairs; each component (actually: interface) stored in the array can be accessed via its name. To add a new function to CPMDL an extension programmer must program a component that implements the IExecute and IFunction interfaces and add this component to the COPDA engine's associative array. Once that component is made known to COPDA in this fashion, CPMDL programmers can use the new function (via the name that was used to store it in the COPDA engine). When COPDA executes that CPMDL function, that component's IExecute.Execute method is called, which should implement the function. The arguments to that method are a list of the function call's arguments and an interface to the COPDA engine component.

Adding a new CPMDL method is the same as adding a new 'normal' CPMDL function, except that the name given to the method must have a special form: the name of the class, followed by a '.' followed by the method name. For example, the set method of the var class (which is used to assign a value to the variable) is known in the COPDA engine's associative array as a function called 'var.set'. Like methods in other object oriented languages, a method call is a normal function call, except that an extra function argument is prepended to the argument list: the first argument in a method call is the object the method applies to. For example, from the viewpoint of an extension programmer implementing a CPMDL method, the CPMDL statement '$a.set(1)' is treated as if it was written as 'var.set($a 1)'.

Each CPMDL 'class' must also be implemented as a component and be made known to COPDA (as a component that supports at least the IClass interface). The task of this component is to create other components, namely the components that represent the variables. These latter components in turn should support the IVariable interface and in case of parameter classes they should also support the IParameter interface, whose methods define the behaviour of properties (such as validity checking).

More details about the required interfaces and about the implementations of functions, classes, methods and variables are provided in Appendix B.

4.9.3 COPDA libraries

The mechanism used to insert objects like functions, classes or methods into the COPDA engine is called the library mechanism. A COPDA library is a COM component. The task of a library is to 'register' functions, classes, etc. in the COPDA engine's associative array.

Each COPDA library has a unique name that is used to find the DLL implementing the library and to prevent a library from loading more than once. A COPDA library can be loaded either by the COPDA program using the COPDA engine's AddNamedLibrary method or by the COPDA application by loading a module with a non-empty uses section into the engine (the uses section contains a list of library names).

Most COM components mentioned in this thesis are not true COM components in the sense that they are not accessible via the operating system's COM functionality (because they are typically not registered in the system's registry). COPDA libraries are the notable exceptions to this rule. COPDA uses the name of the library to construct the name of the COM object that implements the library and then uses that name to open the library (using a standard
COM function which requires the library component to be registered in the system registry). Once opened, COPDA calls the library component's `RegisterLibrary` method.

This `RegisterLibrary` method can do anything it wants. The intention is that it registers functions, classes and methods in the COPDA engine. However, it can be used for other purposes as well. For instance, it can add a new `data destination` to the COPDA engine (relieving the COPDA program from this task, and allowing a COPDA `application` to add a new data destination to the engine) or even replace the `data source`.

To prevent loading a library twice, the library component must also add itself to the engine's associative array. COPDA does not load a library when it is already loaded; COPDA knows that a library has been loaded already when the library name already occurs in its associative array.

### 4.10 Conclusions

#### 4.10.1 summary

I designed and implemented COPDA, a software foundation that provides data processing features for both `program` developers as well as for `application` developers (the distinction between the two was defined in section 4.2.2 and further refined as the low-level vs. high-level views in section 4.5).

#### 4.10.1.1 Low-level (program) view summary

For `program` developers COPDA manifests itself as a single COM component, the COPDA engine. The COPDA engine provides the following functionalities:

- The engine acts as an `interpreter` that can load data processing algorithms in memory for later execution. The algorithms (called `module sets`) are described in the CPMDL language. See the following subsection (4.10.1.2) for a summary of CPMDL features.
- The engine can `execute` the module set it has loaded in memory.
- The engine provides support for `editing` the in-memory description of the module set (and saving the modified module set to a file for later use).

The COPDA engine is not a monolithic component, but comprises a (variable) number of sub-components. The number of sub-components in the COPDA engine depends on the `module set` that is loaded and on the currently loaded extensions (libraries).

To access (read and write) data, the COPDA engine uses `data source components` and `data destination components` (see section 4.7, especially figure 4.2). These components are program-dependent and must be provided by the `program` developer to the COPDA engine. In `offline` programs, these components are used to read or write various file formats. In real-time programs the data sources and destinations read or write real-time media, such as network streams.

The task of data sources and data destinations is to make this multitude of file and media formats available for use by the COPDA engine. They do so by presenting the data in the form that is required by COPDA's `data transfer model` (section 4.7.3, especially figure 4.6) and COPDA's `signal model` (section 4.7.2). The latter model actually comprises two sub-
models: the *sequential data processing* paradigm (introduced in section 4.7.2.1) and the model used to represent signal values (section 4.7.2.2).

At the moment of writing this section, I have implemented three different data sources (for the HPDATA system, for EDF files (see section 6.3.2) and a 'dummy' data source that can be used for processing calculated data). I also implemented two data destinations (for writing HPDATA-style SBF files and for displaying signals graphically (see section 5.1)). The HPDATA data source and data destination are actually implemented as one component that implements both the data source and data destination *interfaces*.

To use the COPDA engine, a COPDA program has to implement the *data sequencing model* (see algorithm 4.3 in section 4.7.4). This is not implemented in the COPDA engine itself because its implementation will be different for different types of programs (real-time *data-push* programs vs. offline *data-pull* programs).

### 4.10.1.2 High-level (application) view summary

To the application developer, COPDA presents itself as a CPMDL interpreter. The task of an application developer is to design a *module set* that performs the application’s task when executed by a COPDA program that suits the application’s needs.

The idea behind the CPMDL language is to describe a data processing algorithm as a *network of modules*. Each module performs an operation on one or more signals and produces one output signal (e.g. a filtering operation or the addition of two signals).

For some users (application developers) it will be sufficient to connect predefined modules into a network. Other users will require being able to create new or modify existing modules. CPMDL is the language that describes both the module network (how the modules in the network are interconnected) and the algorithm to be performed by each module. Details of the CPMDL language and how it is used by the COPDA engine were described in section 4.8.

One of the goals of COPDA (and CPMDL) is to be *extensible*. COPDA defines a mechanism (the *library* mechanism) that allows COPDA users to add new functions, classes and methods to CPMDL at runtime (section 4.9). The same mechanism can also be used to add data source components or data destination components to the COPDA engine. Adding new functions to CPMDL can be useful for three reasons:

- Implement functions not yet provided CPMDL
- Implement speed-optimised versions of algorithms
- Implement interactions with other applications, other computers, the operating system or the user that are not yet provided by COPDA (this includes adding new *asynchronous data destinations* - see section 4.7.5).

The current version of COPDA implements 8 control statements, 68 functions, 61 methods in 13 variable classes, including one class defined in a 'library'. Two other *libraries* implement another 13 functions. A Windows help file is available that documents all of these functions, methods and classes.

### 4.10.2 Requirement evaluation

The evaluation of the requirements and design goals of COPDA is postponed to section 6.2.
Chapter 5
Using COPDA

This chapter describes example COPDA programs and COPDA applications. Section 5.1 describes *DataWand*, an offline COPDA program. Section 5.2 describes the use of DataWand in analysing medical problems. Section 5.3 describes the use of COPDA in real-time programs.

This chapter uses terminology that was defined in the previous chapter, such as the concepts *data source* and *data destination* introduced in section 4.7.1. This chapter also uses the *nuances* in terminology defined in the previous chapter, most notably the distinction between *applications* and *programs* as defined in section 4.2.2.

5.1 DataWand

DataWand is an *offline* program (see section 2.2.1) that is built using COPDA (see chapter 4). DataWand is special amongst COPDA programs in that it was developed together with COPDA. Co-developing DataWand together with COPDA provided me with a test bench to test concepts before including them in COPDA. To be more precise, at the start of the design process, there was only a data processing program (DataWand). Designing the data processing core of this program as a separable system usable for other tasks than offline data processing was an idea that evolved in the early stages of the design.

This section focuses on aspects specific to DataWand, such as its user interface and its CPMDL development support. The data processing features of DataWand are the data processing features provided by COPDA. Those features were already described in chapter 4.

5.1.1 Design goals

As in most software designs, the design of DataWand was the result of a gradual refinement of a set of initial requirements and wishes into a detailed set of requirements. As DataWand was co-designed with COPDA itself, many of COPDA’s requirements influenced the design of DataWand and many of DataWand’s requirements influenced the design of COPDA. In the early stages of the design the distinction between COPDA and DataWand was blurred; only during the refinement stages of the design the distinction became clear.

The original goal of DataWand/COPDA was to ‘process’ data provided by the system described in chapter 3. ‘Processing’ this data includes filtering data, viewing data, analysing data and performing pattern recognition tasks. The initial requirements and wishes for the program that became DataWand and the underlying COPDA system were:

1. Preferably, DataWand should be a stand-alone system, not depending on any external programs (such as compilers) or third-party packages or toolboxes. This allows distribution of DataWand not to be encumbered by licenses typically associated with such third-party tools (see also section 2.6.2.3).
2. DataWand should be highly extensible. It should be possible to 'plug in' modules that add new functionality or change existing functionality, without having to recompile DataWand. If DataWand had to be recompiled to use plug-ins, the DataWand user would be required to have access to DataWand's sources and the compiler used to build DataWand (Delphi 3). Not having to recompile DataWand also opens the way to hot-pluggability of plug-ins: plug-ins can be loaded dynamically at the discretion of the application that is being run by DataWand.

Aspects of DataWand that are eligible for extensibility include data input (supporting alternative data formats), data output (supporting alternative data formats and supporting communication with other programs) and functionality (adding new operation primitives for processing data).

3. On one hand, DataWand should be usable by medical researchers lacking an extensive knowledge of signal processing techniques. On the other hand, it should provide a powerful data processing system for use by engineers. These two sub-goals (simplicity and power) can be conflicting.

5.1.2 Design process of DataWand

This section describes how DataWand evolved from the simple script execution environment it was in its first stages of development to the Integrated Development Environment (IDE) for CPMDL modules and Runtime Environment it is now, comprising a module network editor, a module functionality editor and a runtime environment (CPMDL interpreter).

5.1.2.1 The early stage: a script processor

My original idea was to let the user describe the operation to be performed by DataWand as a script written in a dedicated signal processing language. After loading a script into memory, the user would specify the data the script would operate on and run the script. This approach is good for engineers having programming knowledge, but describing the data processing operation in a (textual) script is not very suitable for non-programmers. Instead of directly addressing this usability aspect, I will describe the history of DataWand and COPDA and how this usability problem 'evolved away'.

In the early phase of the development of COPDA and DataWand, COPDA was considered a script interpreter and DataWand was seen as the user interface for it. Evolution of the design started when I improved the scripting system.

A script in the original scripting language consisted of three sections, called the construction section, the variable section and the operation section.

The first section (the construction section) defined a network of modules. A 'module' in this early version was a 'black box' with zero or more input signals and exactly one output signal. At each sample time a module performs an operation resulting in a new value for its output signal. The output signal value is stored in a 'shift register', so other modules using the output signal can access a (limited) number of the previous module output samples. The operation was restricted to a single expression. The specification of each (proto-)module comprised three parts: a name for the module, the number of output samples stored in the output buffer and an 'operation expression' that described how to generate the output sample. The (optional) input signals of the module were not mentioned explicitly, but only implicitly as references to other modules in the operation expression. The operation expression could describe operations as simple as 'get the current value of the Heart Rate parameter'. It could
do some actual calculation using the input signals (output values of other modules), such as calculating the difference between two input signal values. The calculation could involve accessing values calculated previously for another module; a simple 'high pass' filter could be implemented by making the 'high pass' module calculate the difference of the current and previous value of its input signal. To keep the syntax simple, the operation expression was limited to one single expression in this predecessor of CPMDL. The operation expression could only use the outputs of modules that were declared earlier, preventing circular references amongst modules.

The optional second section of the script (variable section) allowed to declare (global) variables, which allowed the 'module network' to have a 'state'.

The third section of the script (global operation section) described what to do with the values calculated by the 'modules' (e.g. storing them in a file). The global operation section typically consisted of a few initialisation statements, a loop iterating over all input data samples (including a set of statements that were executed for every sample) and zero or more finalisation statements. Its role was similar to algorithm 4.3 (the COPDA execution loop), except that part of the functionality executed by modules in algorithm 4.3 had to be coded directly in the script. Where input data samples came from (from which file) was specified outside the script. The global operation section was the only section in which variables could be used.

5.1.2.2 A more powerful module concept

There were various aspects in this system that needed improvement to make it more usable. The actual ‘work’ was split in a rather arbitrary way between the construction section and the operation section. Most algorithms could be implemented either by doing most work in the module calculations and only a little bit of work in the global operation section or by doing most work in the global operation section and using the modules only as a storage facility to store previous samples of data inputs. Distributing code over multiple smaller functional units (modules in this case; procedures and functions in conventional languages) has always been considered a better programming practice than having the same code as one big lump. The major problem was that the definition of the operations to calculate module values was restricted to one expression (to keep the syntax simple), disallowing more complex functionality to be used in the module definitions. It would be nice to use a multiple-statement description like the one in the global operation section to define a calculation to be performed to obtain a module's value. I decided to extend the module definitions to allow for more complex operations.

A second problem was the use of global variables. Common programmer's wisdom tells that using global variables should be avoided in any programming language in favour of local variables whenever possible. This however requires a concept of 'local variables' to exist in the first place. In the scripting system, local variables could be added as variables that were only 'visible' in the definition of one module.

Local variables usually require some form of initialisation before they are used. A mechanism should exist in the modules that allowed them to initialise any local variables before any signal output values were calculated. I solved this problem by making explicit the structure already typically present in the 'old' global operation section: splitting it into a separate initialisation section, operation section and finalisation section, and distributing the contents of these sections from the old 'global operation' section into three sections for each module. Essentially, the form of a 'script' changed from multiple sections, one of which described all
modules, into a set of module descriptions, each of which contained multiple 'sections'. Furthermore I introduced a fourth section in each module for declaring *local variables*.

These changes nearly removed the need for the old style 'global operation section' and removed the need for the (global) variable declarations. The module definitions defined a network of interconnected modules. Each module has zero or more inputs connected to outputs of other modules and each module generates exactly one output value. The fact that each module has exactly one output in COPDA results from the original nature of a module in the original scripting language: a module was a storage for a single sequence of output values together with a description of how to calculate the next value for that sequence. As storing previous output values is one of the tasks of a module, other modules connected to this module's output can access previous output values (a principle that is commonly used in filtering operations). The global operation section was only required to define how to write the output values of the modules at the end of the module network to a file (the system only supported file output in this phase of development). These operations could easily be defined in a module as well, albeit a module whose output is never used and hence can be omitted. Using such 'input-only' modules allowed the removal of the global operation section entirely and to define the entire script as nothing but a set of module definitions.

### 5.1.2.3 Graphical interface, variables and properties

These two modifications in the scripting system led to the predecessor of the CPMDL language that is used in COPDA. I realised that the modules in a (proto-)CPMDL script and their interconnections could be visualised graphically. I extended DataWand to provide a graphical interface to visualise the modules in a script. This version allowed editing the script graphically by adding or deleting modules and by changing the interconnections in between modules.

The first extension to be made to this version was to introduce a general framework for variables that are not a storage for a number, string or similar standard value, but that are objects with a set of methods depending on the type of the object. Such objects can abstract any kind of 'state' or 'storage' functionality and via its methods provides a clear way of documenting what functions can operate on these objects. An early example of such an object is the *dump* object that allows writing lines of text to a file. Some changes were introduced to the existing variables to make them fit the 'object and accompanying methods' model as well. This change influenced the way the following problem was solved.

The module system allows defining frequently used modules as module 'templates'. Such module templates can be copied into a 'script' graphically by the user of DataWand. Using such 'canned' modules allows users that are not well versed in the art and skill of programming and signal processing to start using DataWand for simple data processing tasks. In the earlier version of DataWand the only operations a user could perform directly in the GUI were: creation of a new module (by copying a template), deleting a module, changing the size and location of the on-screen representation of a module and interconnecting two modules. A problem was that to change anything in the *functionality* of a module required modifying the textual description of the module. For instance, to change a module calculating the median of the most recent 5 samples of its input into a module calculating the median of the most recent 7 samples required the user to open the textual description of the module and to manually replace the occurrence of the number '5' with '7'. This is more than can be expected from a user of DataWand who just wants to use an existing median filter module without having to know how it is implemented. However, being able to change such values is important. The solution to this problem was inspired by the way a similar problem is solved.
5.1 DataWand

in Borland's Delphi. I modified the module system to allow each module to have zero or more properties. A property is a value that can be changed by the user by other means than editing the text describing the module directly. Although COPDA does not possess a graphical user interface for editing properties, it allows COPDA programs such as DataWand to read and write property values. In their turn, such programs can present these property values in a graphical user interface, and provide the user with editing capabilities (see section 4.8.5). When the user changes a property value, the program 'writes' the new value into the module text. Inside CPMDL, a property is treated as an object with a constant value (while a script is running, a property value is constant).

5.1.2.4 Extensibility

The requirements and wishes of section 5.1.1 are met by this system: the module system enables all required forms of data processing, it is a stand-alone system and it provides both a (high-level) graphical module editing interface as well as a textual module editing interface (which in DataWand can be accessed directly from the graphical module editing interface). The point not mentioned in the previous paragraphs is extensibility.

During COPDA's and DataWand's design I did keep extensibility in mind though. Extensibility was one of the reasons to build the design around Microsoft's COM technology, allowing any extensions to be language-independent components. The CPMDL language is extensible using libraries. A library is a dynamically loadable component that can incorporate new functions and new object types for use in CPMDL scripts, extending the vocabulary of CPMDL. To support libraries, an extra section is introduced in a module description: the uses section. Libraries are loaded by including their name in the uses section of any module.

Furthermore extensibility is achieved by not fixing the input and output components used by DataWand, but making them replaceable. This allows reading input signals from media different than the data files produced by HPDATA (the data acquisition system described in chapter 3) and allows writing signals to other file formats.

5.1.3 Final design specifications

Design specifications change and are refined during the process of designing a system. I started designing DataWand from the requirements and desires described in section 5.1.1. The final design specifications at the end of the design are described below. Some of the original requirements are distributed over several 'new' specifications. Other specifications are added to highlight features that proved useful during the design.

1. (Processing data) DataWand should provide the capability to process patient data files recorded by the system described in chapter 3. 'Processing' this data includes filtering the data, viewing data, analysing data and performing pattern recognition tasks.

2. (Prototyping) DataWand, being an offline program itself (section 2.2.1), should provide the capability to design signal processing and analysis operations (both modules and complete module networks) that are to be used in COPDA applications run by other COPDA programs, including real-time applications.

3. (Visualisation) DataWand should provide the capability to show numerical data (both input data and the results of processing these input data) graphically.

4. (Simplified processing - modules) DataWand should provide the capability to perform standard signal processing operations on its input data. Users that are not well trained in
Using COPDA

signal processing techniques and programming techniques should still be able to specify how standard operations are to be performed on an input signal.

5. (Advanced processing - CPMDL) DataWand should provide the capability to perform complex signal processing and analysis tasks. Using this advanced functionality should only require the use of DataWand and not involve using external programs such as compilers. The advanced processing functionality uses a text-editor interface accessible from the simplified processing user interface.

6. (Non-numerical signals) DataWand should provide the capability to process data that are not purely numeric in nature. Such data includes textual signals (each sample is a text string instead of a number) and numerical signals with missing samples (samples can be a number or an 'invalid' constant).

7. (Synchronous and asynchronous output) DataWand's signal processing and analysis operations should be able to produce one of two kinds of output (or both): an operation applied to a signal can result in another (synchronous) signal (which in turn can be processed further or shown graphically) or an operation applied to a signal can result in zero or more (asynchronous) messages (e.g. a textual message that a certain pattern has been recognised).

8. (Sequential processing) DataWand can process its input in a sequential fashion (process data for one sample time, then process data for the next sample time, etc.). It is not required to be able to perform signal processing tasks that require all samples of a signal to be known in advance (such as typical statistical analysis applications do). Sequential processing allows treating real-time signals and offline data in the same fashion. Requiring advance knowledge of sample values would disallow this equal treatment, as advance knowledge of sample values is not possible for real-time signals. Equal treatment of both data access paradigms types is desirable: it keeps COPDA's structure required for supporting both data types relatively simple and makes using DataWand as a development environment for real-time applications easier.

9. (Module flexibility) DataWand's standard signal processing capabilities should be easily extendable and modifiable by using the advanced user interface to create new 'standard' modules.

10. (CPMDL flexibility) The operations (CPMDL functions) available in DataWand's advanced user interface (topic 5) should be extensible by a user who has access to any compiler able to produce COM components. This extensibility should not require access to the full source code of DataWand, but be entirely defined via COM interfaces.

11. (Input flexibility) DataWand should provide the capability to use other input sources than the files recorded by the system described in chapter 3. This allows using DataWand for other purposes than just processing this specific kind of data. Supporting other input sources is implemented by (dynamically) loading a different component to perform the input operations.

12. (Output flexibility) DataWand should provide the option to use alternate output destinations, such as destinations that are intended for use with real-time programs. An example of such an alternate output destination is a component that sends a message to another program when a certain pattern is detected. Supporting alternate output destinations is implemented by loading a COM component that performs the new output operations.
Some of these points can be summarised in the following sentence: DataWand should be an Integrated Development Environment (IDE) for CPMDL modules, including functionality to construct module networks (= data processing algorithms) from predefined modules, functionality to create new modules or modify existing modules and DataWand should be a Runtime Environment for module networks: the functionality to run COPDA applications.

5.1.4 User interface design

One of the challenges in designing DataWand was the design of its user interface. COPDA is designed to have no user interface at all, not restricting the designer of a COPDA program in his choice of a user interface. Most requirements for DataWand's user interface can be derived from the general requirements for DataWand provided in the previous section. This section describes the aspects of DataWand's user interface design.

Many guidelines exist for designing user interfaces. A good starting point for finding information on user interface guidelines can be found on Microsoft's web site (see [UI Design]). User interface elements should behave more or less in the same way across programs (on the same operating system). A user of a Windows program will be very confused when that program has no title bar. He will expect that clicking the button with a small cross at the far right of the title bar will close the program. He will probably expect that there is a menu bar, for which the last entry in the first menu also closes the program. In English-language programs, he will probably expect that first menu to be labelled File and that last entry to be labelled Exit.

In user interface design it is a good idea to 'steal' as much ideas from similar programs as possible. This results in a user interface with many elements that are already familiar to the user and is therefore easier to learn. As DataWand can be viewed as an Integrated Development Environment it is useful to keep other IDE's (such as Delphi or Visual C++) in mind during the design of the user interface. Several user interface elements of DataWand were inspired by Delphi's user interface.

![DataWand Diagram]

*Figure 5.1 Specifications to be made by the DataWand user, as seen in the context of the COPDA I/O model (as introduced in figure 4.2).*
The user interface of Data Wand comprises one main window that in addition to the standard user interface elements that any Windows program user expects (menu bar, status bar, tool bar) has a main area that can show five different tabs. Each tab provides a user interface to a different aspect of Data Wand. See figures 5.2, 5.3, 5.4 and 5.7.

To use Data Wand to process data, a user has to specify three elements (see figure 5.1):

- **what data** to operate on (input specification)
- **what operation** to perform on the data (operation specification)
- **what to do with the result** (output specifications).

Each of these three specifications is related to a different part of the COPDA I/O model: the input specification tells the **data source** where to get its data, the operation specification tells the **COPDA engine** what to do with the data, and the output specifications tell the **data destinations** what to do with the result. Each of these three specifications has its own part in Data Wand's user interface (a separate tab). The **Input** tab provides access to all input specification functionality, the **Output** tab provides access to all output specification functionality, and the **Module Set** tab provides access to the operation specification functionality. Section 5.1.5 describes design issues related to the input and output specification part of the user interface and section 5.1.6 describes the user interface issues related to the operation specification.

The two other tabs display output produced by Data Wand. The first tab (labelled Messages, see figure 5.2) is the console output (see section 4.7.5.1). It displays messages generated...
5.1 DataWand

during processing, both explicitly generated by modules and implicitly generated by DataWand itself.

The last tab (labelled Graphs, see figure 5.3) is the graphical user interface for the graph recorder data destination component (which is not a stand-alone data destination but is built into Data Wand, as per point 3 in section 5.1.3). It shows the graphs that result from the data processing that was performed. It can also show graphs of the input data.

![DataWand's graphs tab, displaying input and output graphs.](image)

5.1.5 Input and output specifications

According to the final design specifications of section 5.1.3, DataWand must at least be able to process the data files stored by the system described in chapter 3. Specifying what data is to be processed by DataWand entails specifying which data files are to be processed. This may sound trivial, but users of the data acquisition system are probably barely aware of the fact that data is stored in files at all. They will prefer to treat the recorded data not as separate files (with names carrying no indication of their content) but as a set of pseudo-files, each recording data for one patient. Each of these pseudo-files actually consists of one or more episode data files (see section 3.6.2.2). To find what episode data files belong to a certain patient, the episode database has to be queried.

5.1.5.1 File drag and drop

The first versions of DataWand only allowed selecting raw data files as input. No mechanism was provided to select input files by a patient's name or identifier. Implementing that
capability would require accessing the episode database, which was something I wanted to avoid\(^{20}\).

When a user wanted to select data pertaining to a particular patient, he had to select the data files for that patient by dragging those files from an explorer window\(^{21}\) and dropping them in the appropriate box in DataWand's user interface (this method of selecting input data is still supported in the current version of DataWand as one of the three supported input specification methods). Using file drag-and-drop has advantages and drawbacks.

An advantage is that database support in DataWand is not required. Another advantage is that this user interface can be used for file-based data sources other than the data source for the data recorded with the HPDATA data acquisition system described in chapter 3 (data sources were discussed in section 4.7.1). I consider building into DataWand a user interface for only one data source a violation of the goal of input flexibility.

The main disadvantage of the drag-and-drop approach is that the user must know the name of the files containing a certain patient's data. For the HPDATA files this requires the user to look up the patient information in the episode database manually. Another disadvantage is that the drag-and-drop user interface to import these files from the explorer into DataWand doesn't work very well in practice, and is definitely not intuitive for first-time users. This is not due to inherent comprehensibility problems of drag-and-drop in general, but of using drag-and-drop across different programs (explorer and DataWand in this case). An inter-program use of drag and drop causes focus changes (a different window becomes active and 'jumps' in front of all other windows) that are confusing to the user, especially when one of the two program windows involved is in its maximised state.

5.1.5.2 Configuration strings as lists of filenames

To allow a more flexibility in specifying what data is to be processed than attainable via file drag-and drop, I devised a system using configuration strings as the preferred way to perform the input specification. The 'old' drag-and-drop user interface is still available in a form that is compatible with this configuration string system.

When using the drag-and-drop interface, the list of the dragged-and-dropped filenames is converted to a string by DataWand (by concatenating the filenames, separated by commas). This simple form of input configuration string is then sent to the data source component (see section 4.7.1) to configure it (for the HPDATA data source that is: inform it what files it should load).

Using a string for this purpose is a generic way of configuring any data source, independent of what that data-source's specific requirements are. Such a configuration string may contain other information in addition to filenames to store data-source specific configuration options. This allows DataWand to store the input specification without requiring it to 'understand' the string. How such a configuration string should be formatted depends on the data source, DataWand/COPDA only needs to know that it is a string.

Creating a valid configuration string for a data source does require some knowledge of the kind of data the data source provides though. In the early versions of DataWand mentioned

\(^{20}\) Adding episode database access would require adding the Borland database engine to DataWand, which would make DataWand's size much bigger, just to be able to convert patient names to file names for the benefit of the data source importing the data from the data acquisition system described in chapter 3.

\(^{21}\) DataWand is a Windows '95 application, so the user can be assumed to have access to the Windows 95 explorer. Of course, any application supporting file drag and drop can be used for this purpose (e.g. the Windows Commander).
5.1 DataWand

above, this knowledge was built into DataWand for the data source importing the HPDATA-acquired data: DataWand knew that a comma-separated list of filenames was a valid input configuration string and knew how to build such a string from the names of the dragged-and-dropped files.

My reason for not defining a HPDATA-specific user interface for selecting input data by patient name or patient identifier was that making such a specific interface part of DataWand conflicts with the spirit of keeping DataWand independent of what data source it currently uses. However, the solution using the file drag-and-drop user interface and interpreting the resulting configuration string as a list of filenames is not independent from the data source either. The fact that the (early) version of DataWand mentioned above needs to have knowledge of how the input configuration strings are structured proves that. Furthermore, this user interface style (drag-and-drop of files) would not make sense for a data source that does not use files (e.g. a data source reading data from a network stream or a data source generating data procedurally).

5.1.5.3 Delegating input specification to the data source

A solution to this cosmetic dilemma is found by considering any user interface for configuring one type of data source not to be a part of DataWand, but to be a part of the data source. This view allows (or even requires) data sources to include their own 'user interfaces' (dialog boxes) to define and edit configuration strings. The use of COM as the paradigm underlying the design of COPDA and DataWand gives a direction of how such functionality of a data source can be implemented: data sources (which are COM components) should implement a (COM) interface that has a method whose implementation should show a dialog box that allows the user to 'configure' the data source. What 'configure' means depends on the particular data source at hand. The result of this 'configuration' (a string) can be stored by DataWand (or other COPDA programs), and DataWand can later use this string to re-apply the same configuration.

In the current version of DataWand, the HPDATA data source implements a configuration dialog box that does allow selecting patients by their name (accessing an episode database to map patient names to patient data file names). See figure 5.5.
Using COPDA

The COM interface that defines this functionality is the `InputDevice` interface, whose `ConfigurationDialog` method should pop up the (data source dependent) configuration dialog box (all COPDA interfaces are described in Appendix B).

Like data sources, data destinations also should implement a configuration dialog to allow users to set the specific properties of the data destination. This configuration dialog box is accessed by DataWand via the `ConfigurationDialog` method of the `IOutputDevice` interface (that every data destination component must implement). (See Appendix B).

Input configuration strings can become quite complex for data sources that have many configuration options. To prevent the user from having to re-enter all input configuration settings every time he starts DataWand, it is useful to be able to save a configuration string to a file and be able to restore the same settings later. DataWand can save the input configuration strings to files called patient data selection files (PDS files). Details about how PDS files are structured are provided in Appendix E. DataWand’s input tab (figure 5.4) provides the user interface to load and save PDS files and provides access to the (data-source defined) user interface to edit PDS files (for example, see figure 5.5).

5.1.6 Operation specification

Besides specifying what data to operate on, a DataWand user has to specify what operation to perform on the input data. Section 5.1.2 already described how the operation specification method evolved from a purely text-based method to a graphical method.

Conceptually, the operation to be performed is specified as a network of modules. Each module performs a task using zero or more input signals and produces one (or no) output signal. A module can produce output that is not a signal (e.g. displaying a message if some criterion is met).
All combinations of numbers of inputs (zero or more) and numbers of outputs (zero or one) can occur in modules. Examples of tasks that can be performed by a module include: selecting a vital sign parameter from the input data (no input signal\textsuperscript{22}, one output signal), delaying a signal (one input signal, one output signal), recording a signal for later display in a graph (one input signal, no output signal), calculating the difference between two signals (two input signals, one output signal) and specifying the input configuration string in a module (no input signal, no output signal). An example module network using these modules is shown in figure 5.6: boxes represent modules, lines between modules represent signals.

\includegraphics[width=0.5\textwidth]{figures/module_network.png}

\textit{figure 5.6} An example module network (also known as a 'module set').

The graphical representation of the modules is deliberately kept simple. Each box shows the name of the module, an input connector and a name for each input, an output connector and a resize box. The module's name indicates the purpose of a module. The input connector names indicate the purpose of each input. The connectors (the small black rectangles at the left and right of a module) indicate the presence of a module's input or output and provide graphical feedback when the user connects one module's output to an input of another module. The resize box (the black square in the lower right corner of each module) allows the user to make the size of the box smaller or larger.

I have considered displaying other information inside the modules, but I decided against it in order to keep the user interface simple. One of these non-displayed items is an output name. Displaying an output name is not necessary as each module can have only one output and hence giving outputs a name is redundant - the name of the output 'is' the name of the module. Another item omitted from the user interface is an icon shown inside each module to indicate its function. Such an icon could give a visual clue what the function of each module is without having to read the name of a module. It also could be used in the palette (see next paragraph) to give a visual clue about the modules in it. However, I decided against including such icons in modules for two reasons. First, using icons in this fashion costs some 'screen real estate': without making the default size of a module larger, such an icon and the names of inputs will often overlap, leaving a messy impression of the screen. I do not want to leave out the display of input names, as these are the only indicators of the function of each input. Second, using a different icon for each module requires a module designer to design an icon for each module installed in the palette. I consider this in conflict with my intention for designing new modules to be a relatively simple operation.

\textsuperscript{22} There is an input in this case, but it is not a signal in the sense of inter-module connection.
The user interface to add modules to the module network was inspired by Delphi (the rapid application development tool that was used to design and implement DataWand itself). At the top of the module set tab is a palette (see figure 5.7): an array of named icons, each representing one module type. The modules on the palette are categorised in a few pages containing modules that are similar in functionality. This grouping of modules was used to keep the number of module types shown at one time low. To add a module to the module network the DataWand user first selects the type of module to be placed by clicking on one of the buttons in the palette. Next the user places the new module by clicking in the drawing area. The module will be placed at the location the user clicked. This user interface has proven to be a good way of implementing the task of adding new modules to a module set. A drawback of my implementation of the palette is the fact that all icons representing the different module types in the palette are identical: the only way the buttons are differentiated visually is by the name written on each of them. Adding specific icons for each module type would have been possible but was not implemented. Doing so would have made adding new modules on the palette by DataWand users more complicated, as they would have to design a button bitmap for each module.

To connect one module's output to another module's input, the user double-clicks the module whose output is to be connected. This puts DataWand in connect mode: a line starting at the output to be connected is drawn toward the mouse pointer. When the mouse hovers over another module that line snaps to the input connector closest to the mouse pointer. The user then clicks the mouse button to confirm the connection. Various colouring effects of the
modules provide extra feedback during this operation. (As is usually the case with descriptions of GUI operations, this textual description looks more complicated than the actual operation is).

Most modules perform operations that can be parameterised. E.g., a module that delays an input signal sample can have a ‘parameter’ specifying how many samples the signal is to be delayed. Such ‘parameters’ are called properties in DataWand. These properties can be specified by the user of the module via the property editor part of DataWand’s module set tab (see figure 5.7). Like the idea of the palette, the property editor was inspired by Delphi. The property editor shows a compact list of the properties (see section 4.8.5) of the currently selected module. The property values can be edited by the user by typing in the property editor. The property editor is always visible - when no module is selected it is empty. An alternative implementation to provide similar functionality would have been the use of property sheets, like the ones used in visual basic or the Windows Explorer. Such a property sheet is a dialog box that displays information and implements dedicated editors for each of the properties of an object (see figure 5.8). Though less flexible, I consider a property editor a more user-friendly user interface for its purpose: it is permanently visible (unlike a dialog box) and it is usually more compact (as is clearly visible in the figure).

DataWand additionally supports custom ‘editors’ for its properties. These take the form of dialog boxes and provide the same kind of flexibility and customisation that can be achieved with a property sheet: custom ‘editors’ can provide type checking and/or they can provide a more complex user interface than just a single text field. These custom property editors are invoked by clicking on a small button that is shown between a parameter’s name and its value.
in the property editor. The custom 'editor' dialog box is invoked via an interface method of the parameter class of the property currently being edited (parameter classes were introduced in section 4.8.5).

5.1.7 CPMDL development

Data Wand can be used as a development environment for the CPMDL description of modules (see point 5 in section 5.1.3). I built a CPMDL language-aware text editor into Data Wand that provides support for auto-indentation and language help. The editor is a dialog box that pops up when the Edit entry from the right-click menu of a module is chosen (see figure 5.9).

There are several reasons why integrating a text editor in Data Wand is a good idea. First, having an integrated editor circumvents the need to switch between Data Wand and an external text editor during the development of CPMDL code. In this sense having an integrated editor is what is to be expected by the user from an Integrated Development Environment. Secondly, the built-in text editor has some knowledge of the CPMDL language, aiding the user in nicely indenting the code and allowing context sensitive help (see below).

The third reason is less visible at first. The built-in editor does not edit CPMDL files but edits modules that are currently loaded into the COPDA engine (loaded in the sense of section 4.8.6). The save and load functionality that the user perceives is the save and load functionality of COPDA, not a feature of the editor. This has two benefits. The first benefit is that a user can change various features of the modules in the module network and try the effects of the changes (by running the module network) without having to save the module network. If he is not happy with the results the changes can be discarded. The second benefit is that it allows coupling the run-time environment of COPDA with the design-time environment (Data Wand's CPMDL editor), which is useful for debugging purposes.

When during the processing of data an error occurs, the editor dialog for the module where the error was generated pops up and the text cursor is set to the statement that was executing when the error occurred. This situation occurs not only for run-time errors, but also includes the detection of syntax errors in CPMDL when the modules are loaded (see section 4.8.6). Optionally, this feature can be partially disabled during the actual processing of data, to prevent run-time errors (such as incorrectly formatted data files) from aborting processing a batch of sequences: only the faulty sequence is aborted in this case; processing continues with the next sequence.
The language help feature of this editor allows accessing help for CPMDL functions and objects: when the text cursor is inside a function or statement name (e.g. in the word ifelse in figure 5.9), the CPMDL function reference help file pops up (in the standard Windows help fashion: in a separate window) with the help topic on that function selected. If the cursor was on a variable name, the help topic for the class of that variable is shown (e.g. if the cursor is on $vv in the figure, the help topic on the var class is shown). If the cursor is on a method invocation, the help for that method is shown (e.g. if the cursor is on $vx.set, the help for the var.set method is shown). These help features are available when the text cursor is on a word and that word is a function name, variable name, class name or method invocation that refers to a known topic in the CPMDL reference help file.

5.1.8 Evaluation of the usability of DataWand

To evaluate the usability of DataWand I created a questionnaire that I gave to DataWand users.

The goal of the questionnaire is to get insight in how DataWand is used by various users and to discover problems and suggestions for improvement. The questionnaire will be provided to future DataWand users. In this section I present some of the preliminary questionnaire results.

The questionnaire has questions concerning three different aspects. These aspects can be summarised as follows:

1. User information.
   - In what user category does the user fit (see sections 4.5 and 4.8.1).
   - Since when has the user been using DataWand.
   - Is the user still using DataWand / when was the last time the user used DataWand.
   - How much of his time the user spends using DataWand (that is: how important DataWand is for his work).
   - For what task does the user use DataWand.

2. User satisfaction.
   - Does DataWand provide the features needed to perform the user’s task (both data processing capabilities as well as user interface features).
   - If not, what features are missing according to the user (it may be that the capability is available, but the user was not aware of it, indicating lacking documentation or user interface problems)
   - Are there any tasks that can be done in DataWand, but only in an inconvenient way (find suggestions for user interface improvements).
   - Other suggestions.

3. How DataWand is used.
   - What parts of DataWand are used by the user: the graphical module interface, designing own modules (CPMDL programming), programming extension DLLs (libraries, see section 4.9.3), programming new data sources (check if the answer matches the user category (first question)).
   - Which standard modules are used by the user and which were never used.
   - For what functionality did the user add new modules to the palette (these can be considered suggestions for new standard modules).
The users I gave the questionnaire to did not all use the same version of Data Wand. Several of the users were working with a version of Data Wand that provided features that were added after most parts of this thesis were written and most of which are not detailed in this thesis (some will be mentioned in chapter 6).

At the moment of writing this part of the thesis I have received four filled-in questionnaires. Two of the respondents are using Data Wand in the development of data processing algorithms for the IBIS project (see [IBIS]). One respondent used Data Wand for the development of data processing algorithms for use in the Catharina Hospital's ICU. One respondent recently evaluated Data Wand for use as an educational tool.

The first three respondents used Data Wand very intensively (more than fifty percent of working hours, during at least one month). In the classification of section 4.8.1, these did most of their work with Data Wand as module users and CPMDL users. Two of the three also implemented COPDA libraries (to be able to use existing source code written in C++).

All respondents were satisfied/impressed with the features provided by Data Wand. Feedback from the respondents and from other Data Wand users indicates that the CPMDL language is very easy to learn and use.

Several suggestions were given by the respondents for further improvements to Data Wand. The indicated suggestions were:
1. Copy and paste for modules.
2. Grouping: Ability to select a group of modules and perform standard user interface interactions (moving, copying, pasting) on that group.
3. Linked module properties: being able to link a property of one module to another property in another module (in practice properties of different modules are often related).
4. Module hierarchy: being able to create new complex modules as a collection of simpler modules.
5. Feedback: being able to use an output signal as an input signal (creating a loop in the dependencies between the modules in the network)
6. Multiple module outputs. Allowing a module to have more than one output.
7. There is a small inconsistency between the user interface of the module network design tool and the graph viewer. The graph viewer uses drag-and-drop to select parameters to be viewed, while the user interface of the module network design tool requires two clicks to place a new module (one click on a button to select a module type and one to place that module). Both user interfaces could have used the same mechanism instead of different mechanisms.

The first suggestion already has been implemented in newer versions of Data Wand. It is possible to ‘copy’ the currently selected module and paste it as a new module in the designer. The ‘copy’ operation copies the textual CPMDL module description, allowing the copy and paste mechanism to interact with standard text editor programs. A ‘copied’ module can for instance be ‘pasted’ in a text editor such as Notepad and be edited there. The modified module description can then be copied in Notepad and be pasted back in Data Wand.

None of the other suggestions have yet been implemented in newer versions of Data Wand or COPDA. All of these suggestions have implications and side effects that may not be obvious at first sight that make their implementation far from trivial.
5.1 DataWand

5.1.9 Summary

DataWand is an offline COPDA application. It provides users with a ‘laboratory’ for processing (analysing, viewing, filtering, converting, etc.) medical data. What data file formats are supported depends on what COPDA data sources are installed in DataWand. In the current version of DataWand, the three data sources mentioned in section 4.10.1.1 (HPDATA file input, EDF file input and ‘dummy’ input) are built into DataWand.

The two (synchronous) data destinations built into DataWand are HPDATA file output and the ‘graph recorder’ (figure 5.3). The COPDA console (the default asynchronous data destination) is implemented in a straightforward way in DataWand (figure 5.2).

The functionality provided by DataWand has two distinct aspects:

- DataWand can be used in the design and implementation of data processing algorithms.
- DataWand can be used to run data processing algorithms on offline data.

For design of data processing algorithms (module networks), DataWand provides a two-layer user interface: a graphical layer and a textual layer. Users can use a graphical user interface to build a module network from standard modules in a way similar to a drawing program (figure 5.7). Modules define a series of properties that can be changed by the DataWand user to change details of the module’s behaviour (such as specifying a cut-off frequency of a filter module). To change the basic behaviour of a module (including adding new properties to a module), the user can edit the CPMDL in the textual design interface, which is seamlessly integrated in the graphical interface (figure 5.9). By editing a ‘blank’ module this mechanism also allows the user to design new modules.

The current version of DataWand includes a palette with 49 standard modules to choose from. Users can add modules they design themselves to this palette.

The CPMDL editor (the textual user interface) has some features that aid users with CPMDL programming. Most importantly, the editor supports context-sensitive help: the user can invoke help on the function or method the cursor currently is in by pressing the F1 key.

For running module networks, DataWand provides the user interface to select and configure the data sources (via DataWand’s input tab, figure 5.4) and data destinations (via the output tab). The user interface used for actually configuring a data source or data destination is not implemented in the DataWand program user interface code itself, but is a functionality of that device. (This is not strictly true, as currently the data sources and data destinations themselves are integrated in DataWand, and as such this user interface code is ‘implemented in the DataWand program’).

One of the two implemented data destinations is the graph recorder (figure 5.3). It allows the DataWand user to view the results of processing input signals as well as viewing the input signals themselves. The design of the graph recorder was heavily influenced by the graph viewing health assessment application I developed for use with the HPDATA data acquisition system (section 3.7.1).

DataWand is currently being used in the EU-funded IBIS project (see [IBIS]) and DataWand has been used by several students working with data collected by the HPDATA system (see [Cuijpers, 99], [Pablo, 99]).
5.1.10 Requirement evaluation

The evaluation of the requirements and design goals of DataWand is postponed to section 6.3.

5.2 Example of using DataWand

One of the effects noticeable in the graphs recorded by the system described in chapter 3 are the changes in several parameters when a patient has an ischemia or a myocardial infarction ('heart attack'). An ischemia is a reduction of blood flow through the heart, often occurring before a myocardial infarction, and hence can be seen as a warning. Early recognition of ischemia is an example of a useful task for a medical warning system.

This section describes how DataWand can be used for developing an ischemia detection application. Such an application (in the form of a COPDA module set) can be executed later in a real-time COPDA program.

The reader should be aware that the aim of the example is to make plausible that DataWand can be used in such a warning system's design. Actually designing or implementing such a system is outside the scope of this thesis and the case presented in this section is therefore incomplete. See [Pablo, 99] for an MSc. student report further analysing this subject.

In figure 5.10 the graphs of various parameters influenced by ischemia are shown: clearly recognisable are a depression in both ST segment values and increases in arterial blood pressure and heart rate. A mild version of the phenomenon (ischemia) is seen around the 22:00 time marker, while a more severe version (a myocardial infarction) occurs around 02:00. One aspect of DataWand, useful for medical researchers, is already demonstrated here: the ability to view (multiple) graphs of a patient's vital signs can aid in identifying phenomena that may be suitable for further analysis. Alternatively, seeing graphs in which a phenomenon should be visible but cannot easily be recognised by humans can be a good indication that trying to detect that phenomenon by a computer is a very difficult (or even impossible) task.

The goal of a researcher using DataWand in this case can be to construct a reliable algorithm that can detect certain patterns and that can provide a warning as soon as the pattern is detected. The problem is that the signals that are relevant in this case are often noisy or contain a large percentage of dropped measurements (when the sensors involved mark a sample as 'invalid'). The graphs shown in figure 5.10 are of an exceptionally good quality and do not contain any invalid samples.

The ST segment values that seem to be the best indication of the pattern are especially vulnerable. These values are derived by the monitoring equipment from the ECG measurements, which are easily disturbed when a patient moves or is moved. If so, periods where only one out of several samples is valid occur frequently.

The blood pressure signal and the heart rate signal can be used to support or falsify conclusions drawn from analysing the ST segment values. However, there are many causes for variations in the blood pressure and heart rate signals that are not related to ischemia, so those signals cannot be used to detect ischemia without looking at ST segment values. Furthermore, the blood pressure signal frequently shows 'spikes': false measurements that show up as a short-term sudden extreme increase or decrease in blood pressure, for instance caused by a calibration procedure. Examples of such spikes can be seen in the graphs of figure 5.10, where sharp downward pointing spikes occur around 23:00 and around 03:00.
5.2 Example of using DataWand

A filtering operation can first be applied to the blood pressure signals to remove these spikes. The spikes can be recognised easily because during the spike the systolic and diastolic blood pressure are reported as being exactly equal. In other words: the sensor can no longer recognise the blood pressure wave, but only sees a constant pressure. This situation normally does not occur. As this is a common problem, a module that performs a 'filtering' operation that removes these spikes is available in DataWand. This module marks the offending samples as 'invalid'.

A feature that seems to be a good indicator of ischemia is the steep drop in the ST segment: this is especially clear when looking at the myocardial infarction around 01:30. The ST segment drops more than one mm in four minutes. DataWand's graph viewer helps the researcher to quantify such features.
A researcher can use DataWand to test hypotheses, e.g. the hypothesis that a drop in an ST segment value of more than one mm in a time interval of four minutes is a good indication of MI. A scenario of doing so could be as follows. First, the researcher can build a module set that calculates the change in an ST signal over four minutes (by calculating the difference of the current ST segment value and the value it had four minutes ago) and test if the moments the resulting signal drops below -1 mm coincide with known moments of MI or ischemia. (The problem of getting a list of 'known' moments of occurrence of these phenomena is not considered here).

When he does so, he will quickly discover that this approach is not sufficient: the calculation of the difference has a side effect that the noise on the two signals whose difference is calculated (the ST signal and the delayed ST signal) is effectively added together (because variances of independent signals add together when signals are subtracted). Though this is a well known fact in signal processing theory, looking at the graph of the resulting difference signal should be sufficient to discover that smoothing the ST signal before calculating the difference would be a good idea. The researcher now applies some kind of smoothing operation (e.g. a median filter or a low-pass filter) to the ST signal before calculating the difference. Though the resulting difference will be less noisy, there will probably be several occurrences of the signal dropping below -1 mm that are not explained by known occurrences of ischemia or MI.

![Figure 5.11](image)

*Figure 5.11* Results of applying filters to the ST signal. Both graphs show the difference of the ST signal with the value it had 10 minutes before. In the bottom graph, the ST segment signal was first smoothed by a median filter and a low-pass filter. The falling slopes for the ischemia and MI (see figure 5.10) can be determined as the locations where the bottom graph becomes lower than -0.4 mm. Testing against the limit -1 mm only finds the MI.

The researcher can try to refine the criteria for the detection and after several iterations and after tests on the data of many patients can either come up with a satisfactory algorithm to detect ischemia or MI, or come to the conclusion that detecting these phenomena reliably is not yet within his reach. In both cases, DataWand has provided the researcher with the tools to try several ideas and evaluate their results.
5.3 Real time COPDA programs

Up till now COPDA and DataWand were treated as being heavily interdependent. However, DataWand is just one example of a COPDA program. DataWand's role with respect to COPDA is special, as I developed both together, using DataWand to test COPDA functionality. The goal of my project was to develop a data processing core that can be used in various types of data-processing applications. This section describes how COPDA can be used to develop a real-time medical warning program. At the time of writing this section, such a program has not yet been implemented.

For the rest of this section, assume that the task mentioned in section 5.2 was completed successfully and a module network describing an algorithm that can detect ischemia was designed. Assume further that the signals that will be available in the real-time case are equivalent to the signals that were used to design the application (same sample frequency)\(^{23}\). A COPDA-based real-time program can be designed that executes this COPDA application.

When the module set was used in DataWand, its execution comprised a number of processing steps. Each data processing step processed one 'sample' that was read from a data file. Such a 'sample' consists of a set of parameter values that are measured at one sample moment. The data processing involves 'execution' of the modules in the module network by COPDA's engine component (as described in section 4.7.1.1). It is important to realise that the only components involved in this processing are the COPDA engine, a data source and zero or more data destinations. Parts of DataWand that are DataWand specific extensions to COPDA are not involved.

The only output that is produced by a typical pattern recognition task is an occasional message when the pattern is recognised (that is: typical pattern recognition tasks do not need any data destination). Such messages are typically sent to the console component currently configured for COPDA. When running the module set in DataWand, the console provided by DataWand is used. The other part of DataWand that is used is the main execution loop that, in case of DataWand, retrieves the next sample from the data source (see section 4.7.1.2) and instructs COPDA to process that sample.

For a real-time application, several things are different from the situation when running the module set in DataWand. The way input is retrieved and output is produced are different. First, for a real-time program the data source should be replaced by a data source that retrieves real-time data (from a network) instead of retrieving stored data, as DataWand's data source does. As the COPDA design is component based and a data source is defined to be a separate component, changing the data source used by the engine is a matter of re-configuration. This assumes that such a real-time data source component is available, which is not yet the case at the time of writing.

The COPDA engine's interfaces have no method that starts processing samples until interrupted or finished. Instead, every COPDA program has to define a processing loop itself, that retrieves a new sample into the data source, instructs the COPDA engine to process that sample and then instructs all data destinations to output data. The reason why such a loop was not included as a method in one of the engine's interfaces is that its implementation is different for different application types. One reason for such differences is that inside that

\(^{23}\) In case of the data produced by the HPDATA system this is not the case, as the real-time data is sampled at a rate of once per five seconds and the offline data used to develop the algorithm was sampled at a rate of once per thirty seconds. Typically this can fixed by adjusting some of the parameters of the algorithm. If the situation were reversed (trying to use an algorithm designed for a higher sample frequency at a lower sample frequency), this would in general not be possible.
loop some interactions with the user interface may take place (e.g. DataWand checks if the abort button was pressed by the user). A more important difference is the way typical data-pull style programs (such as DataWand) and typical data-push style programs (such as real-time COPDA programs) will implement the main loop. In a data-pull application, the program instructs the data source to load the next sample. This is repeated each round in the processing loop and can be implemented as a normal loop in the programming language used for the COPDA program. In a data-push application, it is the data source that tells the program that a new sample has been loaded and is ready for processing. It does so by posting a message in the program's message queue\(^\text{24}\). The 'data processing loop' in this case is typically implemented as just one branch of the program's normal event loop (and is not recognisable as a 'normal' loop in the program's code).

So, in a (data-push style) real-time program, the code that is 'driving' the program is implemented differently then it is in a data-pull style program. After initialisation, the real time program has to inform the data source that it is ready to receive events. When new data arrives at the data source, the data source sends an event to the program. The program then instructs the COPDA engine to process the sample and informs the data source that it is ready to receive the next sample. The waiting for the next sample is treated just like waiting for any other event in a GUI program (and is therefore automatically synchronised with GUI events).

The other major difference between real-time programs and programs like DataWand is the way output is produced. Different real-time programs can use different solutions to provide output. For real-time programs that perform pattern recognition as part of a warning system, the problem is that no one is going to sit in front of a computer screen, waiting for a warning message to show up. When a warning is issued (when a pattern is recognised), that warning message should be brought to the attention of the ICU staff (and it should be logged to a file). The responsibility of how best to relay messages to the user of the warning system is delegated by COPDA to the designer of such a system.

There are two obvious ways to relay messages that are generated by COPDA to program parts created by the warning system designer (which in turn should relay the message to the user in an appropriate way). Both use asynchronous data destinations (see section 4.7.5). The first solution is to provide a library (section 4.9.3) introducing a new asynchronous data destination to COPDA and let the modules in the module set use the CPMDL functionality provided by that library to send messages. The second solution is nearly identical but doesn't require the designer to create a new library. To use COPDA, the designer has to provide a component that implements console functionality. This console component already has the capability to display messages. The designer can implement it in a way that displays any messages written to it in a form suitable for a warning message. In the module set, the standard CPMDL function message can then be used to display the warning message. Unless errors occur, these warnings are the only messages that will appear on the console.

Whether a dedicated asynchronous data destination is used or a dedicated console, one issue should be kept in mind when designing the output device: displaying a message should in no case pause the program that is displaying the message. This means that popping up a modal dialog box with the message is not a good idea, as the program will only continue processing the next sample after the dialog box has been dismissed. Some kind of multithreading based solution can be devised to allow the COPDA part of the program to continue while the dialog box is shown, but this raises the question of what to do when a second (or third, etc.) warning is issued while the dialog box is still not dismissed. It is the task of the designer of a real-time COPDA program to resolve these issues.

\(^{24}\) COPDA is designed for use in Windows programs, so each COPDA program has a message queue.
Chapter 6
Conclusions and Discussion

In this thesis the development of several related software projects was presented. Chapter 3 describes the development of a system to acquire data measured on patients in an intensive care unit. Chapter 4 describes the development of a software core (COPDA) that can be used as the basis for development of applications that 'process' data acquired with this system (COPDA is not limited to only this data). Chapter 5 describes the development of an offline data processing program (DataWand) and describes how COPDA can be used as the basis for development of other data processing programs. It also describes how DataWand can be used in practical data processing tasks (applications).

This chapter summarises and discusses the results obtained in the previous chapters and provides suggestions for further developments of the system.

6.1 Data acquisition system

Chapter 3 describes the design of a data acquisition system (the HPDATA system) to collect data that is measured on patients in the intensive care unit of the Catharina Hospital in Eindhoven. A summary of the system I designed was given in section 3.7.1 and depicted in figure 3.4.

The goals for that data acquisition system were listed in section 1.2.1. In this section I will review how these goals were achieved.

- The system should collect data measured by the patient data monitors in the ICU of the Catharina Hospital.

The implemented system uses the already present infrastructure in this ICU (the SDN network) to access the data. Due to limitations in this network and the PC plug-in board used to access the network, only a subset of all measured data is collected (see section 3.5). This subset comprises all slowly-varying signals (parameters) such as heart rate, but excludes fast-varying signals (waveforms) such as the ECG. The parameters provided by the patient monitors in the Catharina Hospital’s ICU include several signals that are derived from the waveforms. This ensures that part of the information conveyed by the waveforms is obtained by HPDATA (in the form of these derived parameters).

- The system has to make those data available for other applications running on the computers in the ICU’s computer network.

The acquired data is made available for applications in two different ways. First, the data is broadcast over the local area network for the benefit of real-time applications. Secondly, the data is stored in data files for use by offline and health-assessment applications. (The classification of data processing applications in real-time, health-assessment and offline was discussed in section 2.2.)
Conclusions and Discussion

These two tasks are performed by two different PCs: a PC running the DOS operating system reads data from the SDN network and broadcasts it as data packets on the local area network; a second PC running the Microsoft Windows 3.1 operating system picks up these packets and archives them.

The actual coupling to ICIS (the PDMS used in the Catharina Hospital’s ICU), which was the original incentive for the project, has not been implemented. The main reason for not implementing this coupling was not technical but organisational: the rules for computers connected to the hospital’s backbone network imposed by the Hospital’s Central Information Department prevented testing the software.

- **The system should make those data graphically available for viewing by intensive care staff.**

  I developed a (health assessment) program for viewing graphs of the HPDATA-acquired data. This program is used by the ICU staff during their daily patient health assessment sessions. It has been in use since August 1996.

In addition to listing what the functionality of the implemented system is, it is also relevant to note what it does not do. Being usable outside the specific setting of the Catharina Hospital’s ICU was not a goal of the HPDATA’s design (as explained in section 3.2.1). Porting the system to another hospital’s ICU will only be useful when that ICU has the same infrastructure (SDN) available.

One important lesson I have learned during the two and a half years the system has been in use is that software that should be running uninterrupted for extended periods of time should not be implemented on a computer running Microsoft’s Windows 3.1/95/98 operating system. Microsoft has recently officially confirmed that a PC running Windows 95 or Windows 98 will crash when it has been running without rebooting for 49 days (2^32 milliseconds to be precisely; after this time a millisecond counter ‘wraps around’ which causes some operating system ‘software devices’ to crash). The older DOS operating system, though being a rather unpleasant environment for program development and being susceptible to Year-2000 problems, is more stable than its successor, Windows. Windows NT and the upcoming Windows 2000 do not have this problem.

6.1.1 Recommendations

After completing any software system’s design, inevitably the moment comes when the designer realises that he would have designed certain parts of the system differently if he were to redesign the system again. Identifying these parts (and proposing viable modifications) can be a valuable lesson for the designer of the system as well as for other designers that design similar systems.

The reasons why a designer would like to take a different approach in case of a re-design are twofold. The first reason is that ‘new’ principles and methods have become available that may have affected the design process. The second reason is that some of the parts of the designed system may prove to have unanticipated undesirable side-effects.

An example of ‘new’ technology is the use of intranet technology (TCP/IP or UDP/IP) instead of IPX for distributing the acquired data. This technology is not really new (as was already mentioned in section 3.5.2.2), but it seems that the IP based protocols are becoming the new standards for all forms of networking and are replacing the ‘older’ IPX and NetBEUI.

---

25 Microsoft has released a patch that fixes this problem on Windows 98 recently.
6.2 Data Processing Core

The functionality provided by COPDA can be concisely described as: COPDA is an interpreter for a custom data processing language (CPMDL). To appreciate the functionality provided by COPDA, it is necessary to view COPDA from the viewpoint of different users.

The first group of users are programmers of data processing programs, for which COPDA is a 'component library' that can be compiled into a program to provide data processing capabilities. These users view COPDA (more precisely: the COPDA engine) as a COM.
Conclusions and Discussion

component that can be used to process data according to COPDA's Input/Output model (see section 4.7.1).

The second group of COPDA users are the users that use COPDA-based data processing programs to develop data processing algorithms. These users are essentially users of the CPMDL language. Data processing algorithms are described in CPMDL as module networks: an algorithm is described as a network of modules, where each module is an entity that performs a relatively simple operation on one or more input signals, producing an output signal. The network is formed by connecting module outputs to the input of other modules. This structure allows CPMDL to be used at two different 'levels': at the 'macroscopic' level users can build complete data processing networks by just interconnecting several 'standard' modules that are predefined by the COPDA-based program. (The COPDA program may provide a user interface to do this graphically - DataWand does this). At the 'microscopic' level, users can modify the CPMDL code of an existing module to create new functionality. A new module can be created by modifying the CPMDL code of a 'blank' module. The CPMDL language provides most features that are necessary to code any data processing algorithm. When a certain function is missing, users can add new functions to CPMDL by coding the desired function in a 'COPDA library' using a conventional programming language. A COPDA library is a DLL that adheres to certain conventions specified by COPDA.

6.2.2 The goals of COPDA

Section 1.2.2 (on page 3) provided a list of goals for the 'data processing core', i.e. for COPDA. This subsection discusses how COPDA achieves these goals.

- The system should provide designers of data processing applications with a set of reusable software building blocks (the core) for performing typical data processing tasks.

  COPDA is such a software core. It provides its data processing capabilities in different forms to different users.

  The main separation in targeted user groups (see section 4.4.4) is between program designers and algorithm designers.

  For program designers (called low-level users in chapter 4), COPDA provides a single core COM component (the COPDA engine) and specifies the COM interfaces that define how to use this component. The interface specifications define the interfaces related to the four models that describe how the COPDA engine communicates with the rest of a COPDA-based program: the input/output model (section 4.7.1), the signal model (section 4.7.2), the data transfer model (section 4.7.3), and the data sequencing model (section 4.7.4). In several places in this thesis I am talking about the components (plural) that are provided by COPDA. There are two causes of confusion about the number of COM components provided by COPDA. The first cause is that it is a matter of taste whether the data source components and data destination components that are included in the COPDA package should be considered proper parts of COPDA or whether they should be just considered as examples of data sources and destinations. The second cause is that the COPDA engine component is not monolithic, but comprises a large number of sub-components (such as functions and methods, module components, CPMDL classes, execnodes, libraries and many more).

  For algorithm designers (called high-level users in chapter 4), COPDA provides an extensible set of data processing primitives. These processing primitives come in two
6.2 Data Processing Core

different forms, each catering for a different sub-group of the **high-level users** (section 4.8.1). Both sub-groups make use of *modules*, blocks that operate on one or more input signals and produce an output signal. For one sub-group (*module users*, working at the macroscopic level mentioned in section 6.2.1), these modules themselves are the processing primitives they use to define data processing algorithms. Users in the second sub-group (*CPMDL users*, working at the microscopic level mentioned in section 6.2.1) create new modules (or modify existing modules) using the data processing language CPMDL. For this second group of high-level users, the ‘primitives’ are the functions provided by CPMDL.

The concept of ‘building blocks’ mentioned in the goal description is different for each of the user groups. For *low-level* users the ‘building blocks’ are the **COM** components involved in COPDA: the COPDA engine and the related components such as **data sources** and **data destinations**. For **CPMDL users**, the building blocks are the functions and methods provided by CPMDL. For **module users**, the building blocks are **modules**.

In the case of the *module users* the ‘building blocks’ themselves (predefined standard modules) are *not* provided by COPDA. COPDA does not predefine any standard modules. Each COPDA program that allows the editing of module networks\(^{28}\) can define its own set of standard modules.

- **A mechanism should be defined to extend the core with new building blocks. As the individual building blocks in the core (including future extensions) may depend on each other, such extensibility has to be taken into account during the design of the core.**

As ‘building block’ means different things for different users, this goal has different implications for the different user groups.

At the lowest level, the ‘building blocks’ are the **COM** components that together will form a program. This includes the COPDA engine, a **data source** and zero or more **data destinations**. Data sources and data destinations are the mechanisms that allow COPDA to treat different kinds of data input and data output in a unified way. New **data sources** or **data destinations** can be added to extend the COPDA functionality at this level.

At the highest level, ‘building blocks’ are modules. Creating new modules is the heart of COPDA and the raison-d’être for the **CPMDL users**.

At an in-between level ‘building blocks’ are the individual CPMDL functions (and classes and methods). An extension mechanism was defined to extend CPMDL with new functions, see section 4.9.3. In section 4.8.1 a fourth user group (the third sub-group of the **high-level users**) was identified (extension programmers) whose goal is to program these CPMDL extensions.

- **The core should be usable for both data processing programs using batches of data (files) as input (‘offline programs’ and ‘health assessment programs’), as well as programs that use continuous streams of data as input (‘real-time programs’).**

Care has been taken to allow a unified approach for both program types. The essence of the difference between the two program types is the way subsequent samples are retrieved for processing: actively picking the next sample in one case and passively waiting for the next sample in the other case. This essence was kept outside the

\(^{28}\) Typical real-time programs do not have need to edit module networks but only need to *execute* them. For such programs the concept of ‘standard’ modules has no meaning: all modules have the same status.
functionality of the COPDA engine proper; its implementation is delegated to the COPDA program instead, as specified in section 4.7.4.

- **The core should be able to use the data acquired using HPDATA, the data acquisition system I developed during the two-year designer’s course. The implies that the system has to support non-numerical data (‘invalid’ markers and textual signals).**

The HPDATA system influenced two of the choices made for the data representation and data stream representation used in COPDA.

The basic data type that is processed by COPDA is a union of all relevant data types found in HPDATA: integers, floating point numbers, strings, time stamps, and invalid markers. It also includes two extra types that are useful in general programming tasks (boolean) or provide powerful extension opportunities (interface). See section 4.7.2.2.

HPDATA produces a multitude of signals that are all sampled at the same rate. This feature was the basis for the sequential data processing paradigm introduced in section 4.7.2.1. The requirement for all signals to be sampled at the same rate has proven to be a disadvantage when COPDA is used with data from other data acquisition systems. See section 6.2.3 below for my solution to that problem.

- **The core should not be restricted to processing data in a particular format, but users of the core (data processing program designers) should be free to use whatever data format they like to use.**

The input/output model described in section 4.7.1 introduces the concepts of data sources and data destinations, whose goal is to abstract the COPDA engine from the details of the data input and output.

- **The core should not put restrictions that are not relevant to the core’s functionality on programs that are built using the core. This implies that the core should not put unnecessary restrictions on the program’s user interface.**

COPDA does not restrict the user interface of a COPDA based program (other than the fact that COPDA requires the Windows operating system). However, COPDA does define two ‘gateways’ that allow COPDA to invoke user interface features.

First, I/O devices can define their own configuration dialog box that allows specifying settings required by that device (the configuration string of that device, see section 5.1.5). To that end COPDA defines COM interfaces that have a method whose implementation (by I/O device components) should invoke these dialog boxes (the ConfigurationDialog method that appears in both the IInputDevice interface and the IOutputDevice interface).

Secondly, each COPDA module has a position and a size. COPDA itself does not use these properties, but COPDA programs that present a user interface for editing module networks graphically can make use of these properties. These properties are integrated in the module definition (in the module’s header section) and are saved to and loaded from the file describing the module network. Because of this, the description of an entire module network (including its graphical layout) is formed by concatenating the CPMLD descriptions of the modules in that network.

- **The core should provide features that allow data processing applications developed for one COPDA program to be usable for other COPDA programs with minimal or no adaptations.**
6.2 Data Processing Core

This was the main guideline when designing the CPMDL language. Data processing algorithms for use with COPDA are described in CPMDL and can be independent of the COPDA program they are used with.

6.2.3 Modifications

COPDA has already proven to be a powerful and flexible basis for developing data processing programs and applications. During its use though, some less desirable aspects of COPDA became apparent as well.

One of those aspects is not really a problem but a limitation that must be understood by designers of COPDA programs. The main driving goal behind the design of COPDA is flexibility. This flexibility comes at the cost of efficiency. When high data throughput is important in a design for a data processing program, COPDA is probably not the best choice as a basis for that program. The use of an interpreter for executing CPMDL code (instead of compiling CPMDL into a program) comes at the cost of some overhead execution time. Measurements indicate that about fifty percent of execution time in DataWand is spent in the execution of the CPMDL code in modules. The other fifty percent is spent in preparing the data sources and data destinations for each sample and in other overhead tasks.

The actual throughput that can be achieved by DataWand is hard to quantify, as it heavily depends on many factors, including: number of signals to be processed, frequency of those signals, complexity of the algorithm, efficiency of the data source and data destination implementation, etc.\(^{29}\)

Another less desirable aspect is that the data representation and data stream representation used in COPDA was initially tailored to the data acquired using the HPDATA data acquisition system. For the data representation this is not really a problem, as it means that the data representation in COPDA is more powerful than needed for most other data acquisition systems (most other data acquisition systems only use numbers as data values). The problem lies with the sequential data paradigm: the fact that all signals are assumed to be sampled at the same rate. For many data acquisition systems this is not a valid assumption. Section 4.7.2.1 already indicated how signals sampled at different rates or sampled at irregular sample moments can still be used (using oversampling and re-synchronisation). These methods come at a severe performance cost though, especially when the signals to be processed are sampled at vastly different rates.

A different solution has already been implemented. During the writing of this thesis, development of COPDA continued. COPDA as described in chapter 4 is a snapshot of the COPDA that existed when I started writing that chapter. The main improvement made to COPDA since then is to introduce a concept of module rates: the modules in the module network are no longer necessarily being processed at the same rate. The COPDA engine behaves as if it processes data at a high virtual sample rate (e.g.: once per millisecond). A module is not executed for every virtual sample though, but only once per MRATE samples, where MRATE indicates the module’s ‘rate’\(^{30}\). The module’s ‘rate’ can be set in the initialisation section of the module. If it is left unspecified, a module uses the same rate as the

---

\(^{29}\) To give an impression of the speed order of DataWand, loading 500 seconds worth of four 250 Hz ECG signals from an EDF file and writing those signals to four writer modules that are not connected to any data destination, took about 43 seconds on a Pentium II running at 350 MHz.

\(^{30}\) Strictly speaking, the word rate is used incorrectly here. As used here, a higher rate indicates less frequent execution, so the module’s rate actually indicates the inverse of its processing frequency.
module connected to its first input (or the virtual sample rate if the module has no inputs). Module outputs are treated in a ‘sample-and-hold’ fashion: When a faster module uses the output of a slower module, the value output by the slower module is used multiple times. Conversely, when a slower module uses data produced by a faster module, some of the generated output values are skipped. That slower module can still access the skipped values though, as the output values generated most recently by the faster module are stored in that faster module’s output buffer. This module rate system makes processing multi-rate data more elegant and far more efficient.

6.3 DataWand

Section 5.1 described the design of DataWand, the data processing toolbox program as mentioned in section 1.2.3.2. That section provided the initial goals for this program. This section reviews these goals.

- **Provide medical researchers with an environment to test hypotheses related to the data acquired with HPDATA.**

  As DataWand is based on COPDA, and COPDA was designed to work with the HPDATA-acquired data, DataWand can process these data.

  DataWand provides a ‘module network design tool’ that allows building module networks using a graphical user interface. As explained in section 4.8.2, module networks are equivalent to algorithm descriptions. Using only the graphical user interface provides access to a set of modules implementing several standard data processing primitives.

  This graphical user interface integrates seamlessly with a textual user interface that allows the definition of new modules or modification of existing modules by editing their CPMDL description. Using this feature (editing CPMDL) requires the user to have some programming skills.

  Section 5.2 provided an example of how DataWand can be used in medical research. Some functions required for scientifically sound hypothesis testing require advanced statistical processing capabilities that are not provided with DataWand. However, DataWand can export results in text files that can in turn be imported in other programs, such as spreadsheets or statistical analysis packages.

- **Provide a prototyping environment for developing data processing algorithms for use in real-time programs built using the data processing software core.**

  DataWand can be used to develop data processing algorithms. The result of such a development is a CPMDL description of that algorithm. This same description can be used in other COPDA-based programs (e.g. real-time programs), if the input signals provided by that other program are ‘compatible’ with the input signals used to test the algorithm in DataWand.

  The fact that data processing algorithms prototyped with DataWand can be used with other COPDA-based programs was a design goal of CPMDL, the language used to describe the algorithms.
• Provide a test environment for the data processing software core itself and for its extensions.

The development processes of COPDA and Data Wand were heavily intertwined. During the implementation of COPDA I used Data Wand as the user interface to test newly implemented COPDA features. Some of the features of COPDA were inspired by the user interface of Data Wand. For example, the fact that CPMDL allows storing graphical parameters of modules in a module header only makes sense when those modules are intended to be viewed graphically.

‘Extensions’ to COPDA come in two kinds. First, there are CPMDL extensions: new functions and classes. These extensions work purely at the COPDA level, and Data Wand will always be able to use them.

The second kind of ‘extension’ comprises new I/O devices (new data sources or new data destinations). The user interface of Data Wand allows the user to select any of the installed data sources and allows selectively enabling or disabling any of the installed data destinations. Furthermore, Data Wand provides access to the configuration dialogs of each of these devices. However, the current version of Data Wand cannot use every conceivable data source: data sources for real-time programs are not supported, as Data Wand is an offline program (see section 2.2.1) and hence uses a data-pull approach to access input data.

6.3.1 Modifications

As initially conceived, Data Wand contained several features tailored to processing the data acquired with the system described in chapter 3. Some of these features were only identified as such by me when I started experimenting with other data sources.

For example, one of these issues applied to the graph recorder, the data destination built into Data Wand that ‘records’ and shows graphs of input signals and processed signals.

Originally, the graphs only faked showing floating point numbers - in reality the graphs only showed integers for which the y-axis optionally could be scaled by either 0.1 or 0.01. This nicely matches data recorded from the SDN (chapter 3), in which all data values are actually ten-bit integers that can optionally be scaled by one of these two factors. Rounding errors produced when converting the floating point signal values to the integers stored in the graph recorder often caused data interpretation problems though. For this reason I changed the graph recorder to store true floating point numbers.

6.3.2 User satisfaction

Early reactions (section 5.1.8) indicate that users are very happy with the features provided by Data Wand. Several users have been using Data Wand intensively for extended periods of time (several months) for data processing work.

Data Wand has been used by students in the analysis of data recorded by the HPDATA system (see [Pablo, 99] and [Cuijpers, 99]).

Currently Data Wand is being used in the IBIS project ([IBIS]) for development of data processing algorithms. It is the intention to use the algorithms designed in that project in a (yet to be built) real-time COPDA program.
6.4 The future of COPDA and DataWand

Unlike the data acquisition system (HPDATA), development of COPDA and DataWand did not stop when I started writing this thesis. I had to take one version of COPDA and DataWand as the reference point for writing chapters 4 and 5, but several modifications have been made afterwards. Some of the additions (the multi-rate support system and the changes in the graph recorder) were already described earlier in this chapter.

At the time I'm writing this part of the thesis, COPDA (in the form of DataWand) is being used to develop data processing algorithms for the IBIS project (see [IBIS] and [Saranummi et al, 96]). In IBIS data files in the EDF data format are used (see [Kemp et al, 92]). To access these files I had to program a new data source and to plug in that data source in DataWand. I discovered a few missing features when I started using this new data source. The most relevant ones were already mentioned in the previous sections of this chapter. The only real change needed to make the EDF data source usable was in DataWand's graph recorder component (as described in section 6.3.1). To improve efficiency when processing multi-rate data I implemented the multi-rate system in COPDA (as described in section 6.2.3).

Another addition I made to COPDA did not change COPDA itself but is essentially a COPDA extension: support for processing vectors of floating point numbers. I added two CPMDL variable classes (section 4.8.4.3), vector and complex, that aid in processing arrays of floating point numbers. From the viewpoint of COPDA these additions are extensions (except that they are not provided as separate DLLs but are compiled in COPDA's standard library). From the viewpoint of high-level COPDA users, these extensions however provide important new features. These classes make converting a certain group of standard data processing primitives to COPDA much easier: primitives that process data not sample by sample but block-of-samples by block-of-samples. The prototypical example of such a primitive is the FFT algorithm (which is implemented as a method in the complex variable class).

The response from users (5.1.8) indicates that DataWand (and therefore also COPDA) is a very useful tool for various data processing tasks, such as data analysis and data processing algorithm design/validation. Some minor improvements to the user interface of DataWand, such as being able to group modules, were suggested.
Appendix A: SBF files

A.1 Introduction

SBF files are the files used to store the data recorded by the HPDATA data acquisition system discussed in chapter 3 of this thesis. This appendix presents the details of this file format.

The data files consist of a sequence of variably-sized records (see figure A.1).

![figure A.1: SBF file structure](image)

Each record consists of two parts. It starts with an eight-byte header part, followed by the header-dependent data part. The header part contains three fields: the size (in bytes) of the record (at most 65535 bytes), the time the record was created (as a standard UNIX-style timestamp: the number of seconds elapsed since January 1st 1970) and a record type field.

The type is a two-byte integer defining the function and interpretation of the data part of the record. The types that are used in the patient data files are specified below (in object Pascal syntax). This file format, that I will further refer to as Sequential Block File or SBF, is flexible in the sense that new record types can be defined later, while existing software is still able to use the new files by just ignoring the records with types it does not know.

Defining a new record type consists of two parts: a two-byte integer must be chosen to uniquely identify the new record type and the layout of the data part must be defined.
A.2 Type definitions

This section presents the (Object Pascal) type definitions of the record types used in patient data files.

// The SBF record header
TRecordHdr = packed record
    timestamp: Longint; // number of seconds since 1 Jan 1970
    wSize: Word; // size of record in bytes
    iType: SmallInt; // a constant indicating the type of the record.
end;

// The first record in a patient data file must be either a TFileHdrRecord or a TFileHdrRecord2.
TFileHdrRecord = packed record
    rh: TRecordHdr; // iType is RHT_FILEHEAD = 8
    dwSig1, dwSig2: Longint; // A signature: Fixed values $44544150 and $00415441 (spelling 'PATDATA' when treated as text).
    szPatient: array [0..19] of Char; // Last patient name encountered
    szid: array [0..14] of Char; // Last patient id encountered
    wBed: Word; // Bed number
end;

TFileHdrRecord2 = packed record
    fh: TFileHdrRecord; // iType is RHT_FILEHEAD2 = 9
    tLast: Longint; // timestamp of last sample
end;

// 'Mixed Records' contain all data from one bed at one sample time.
// Its size is variable. The following record definition only describes the start of the 'Mixed Record'.
// The wcXXX and woXXX fields (wcNum, woNum, wcTxt, etc.) point to blocks of data
// ('subrecords') in the complete 'Mixed Record'. The wcXXX fields indicate the number of
// subrecords; The woXXX fields contain the number of bytes from the start of the 'mixed record'
// where the first of the wcXXX subrecords starts. The definition of the four types of subrecord is
// given below.
TMixedRecord = packed record
    rh: TRecordHdr; // iType is RHT_MIXED = 7
    wBed: Word; // Bed number
    bActive: SmallInt; // Boolean: nonzero if any data arrived at all for this bed.
    wcNum, woNum: Word; // points to zero or more TNumEntry records
    wcTxt, woTxt: Word; // points to zero or more TTxtEntry records
    wcSto, woSto: Word; // points to zero or more TStoEntry records
    wcCtl, woCtl: Word; // points to one TCtlEntry record
end;

// Ctl (Control) subrecords store pseudo-parameters: patient identification information.
// Each 'Mixed Record' should have exactly one control record.
TCtlEntry = packed record
    szPatId: array [0..31] of Char;
        // First 18 characters are Patient Name,
        // Next 12 are patient ID. (rest is terminator)
end;
Appendix A:
SBF files

// Numeric data subrecords. A numeric value can comprise one or three numeric values. The
// interpretation of these numbers depends on the Medical Function Code (MFC). See HP’s
// documentation for a list of MFCs. The flags provided include the scaling factor to apply (0 for
// integers, 1 for 'scale by 0.1' or 2 for 'scale by 0.01') to the 16-bit integers stored as values.

TNumEntry = packed record
  pid: TParamId;  // bits 0-4: bed number;
  flg: TNumFlags;  // bits 5-12: MFC
    // bits 2-0: validity (5 means valid)
    // bit 3: three-valued
    // (when not set, val2 and val3 are
    // copies of val1)
    // bits 5-4: scaled by 10<scale>
    // bit 6: ALM
    // bit 7: LIM
    // bit 8: PA
  val1, val2, val3: SmallInt;  // values (scaling applies)
  iHigh, iLow: SmallInt;  // Alarm limits (scaling applies)
end;

// 'stored' data subrecords. This is mostly equivalent to numeric data, except that it is never
// three-valued, but the space for the second and third value is used to store the hour and minute of
// the most recent measurement.

TStoEntry = packed record
  pid: TParamId;
  flg: TNumFlags;
  val: SmallInt;  // value (scaling applies)
  hour, minute: SmallInt;  // time of measurement (not scaled)
  iHigh, iLow: SmallInt;  // Alarm limits (scaling applies)
end;

// Textual data subrecords. These store parameters that contain strings instead of numbers

TTxtEntry = packed record
  pid: TParamId;  // see TNumEntry
  flg: TTxtFlags;  // bits 2-0: validity
    // bit 3: ALM
    // bit 4: LIM
    // bit 5: PA
  szTxt: array [0..21] of Char;  // 0-terminated string
end;
Appendix A:
SBF files
Appendix B: Interfaces

This appendix describes several of the interfaces used in COPDA. It is outside the scope of this document to provide a complete and detailed reference of all interfaces used. Only a terse description of the most relevant interfaces is given instead.

As most of the development of COPDA and DataWand has been performed in Delphi, I will use Delphi syntax (Object Pascal) to describe interfaces.

B.1 Creating the COPDA engine

As discussed in chapter 4, all features of COPDA are accessed directly or indirectly via the COPDA engine component. This component (and most of its sub-components) is implemented in the COPDA.DLL file. This DLL exports the following function, which is used to instantiate a new COPDA engine:

```pascal
function CreateCOPDA(cons: IConsole; wnd: HWND; libpath: PChar; var engine: IScriptEngine): PChar;
```

This function takes three arguments: an application-supplied console component, the program’s main window handle and a string containing the (semicolon-separated) search path for COPDA libraries. The function returns a new COPDA engine component in the engine argument. If the function succeeds, it returns a nil pointer. Otherwise, CreateCOPDA returns a string containing a description of the error.

Like all functions and methods that are likely to be called from a non-Delphi environment, this function uses the windows standard stdcall calling convention to ensure language-independence. Note that the Delphi type PChar indicates a standard C-style NUL-terminated string (in C or C++ denoted as ‘char *’).

The COPDA engine component supports the following interfaces:

- **IIdContainer** (section B.2). This interface represents an associative array: a mapping from strings to interfaces. Most objects in the COPDA system have a name and can be retrieved from the COPDA engine’s IIdContainer via their name.
- **IScriptEngine** (section B.3). This interface contains the functionality of the COPDA engine not covered by other interfaces. IScriptEngine is actually an extension of the IIdContainer interface. Many methods in various COPDA-related interfaces get a pointer to this interface as one of their arguments. Usually that argument is called context or occasionally engine. Usually that argument is typed as an IIdContainer, though it is safe to access it as an IScriptEngine.
- **IModuleContainer**. This interface ‘views’ the COPDA engine as a collection of modules. It is used to iterate over all modules currently loaded in the engine.
- **IConsole** (section B.4). This interface is used as the default place to send textual messages. The COPDA engine ‘implements’ this interface by relaying all calls to this interface’s methods to the external IConsole component that was provided when the COPDA engine was created.
Appendix B: Interfaces

B.2 IIdContainer

The IIdContainer represents an associative array that stores interfaces and provides access to these stored interfaces by their name (a string). This interface is implemented by the COPDA engine component to store nearly all objects in the COPDA system. It is implemented by every module to store local variables and properties. The IIdContainer interface is also implemented by a nameless component that stores all known input devices (data sources).

The methods of this interface that may fail use standard COM result codes (HRESULT) to indicate failure. These functions return S_OK when successful. The function of most methods is clear from their name.

```c
IIdContainer = interface(IUnknown)
  ['{74EB3EE2-C522-11D2-8EEE-00201881BED0}']
  function Insert(name: PChar; value: IUnknown): HRESULT; stdcall;
  function Remove(name: PChar): HRESULT; stdcall;
  function Known(name: PChar): LongBool; stdcall;
end;
```

Note: Existing implementations in COPDA relay Insert(name, nil) to Remove(name), so the ‘Remove’ method is actually superfluous.

```c
function Find(name: PChar; var id: IUnknown): HRESULT; stdcall;
```

Use ‘FindIID’ to apply a COM typecast together with the search.

```c
function FindIID(name: PChar; const iid: TGUID; var id: IUnknown): HRESULT; stdcall;
```

To iterate over all elements in the id container, first call ‘ResetIteration’, then call ‘Next’ until it returns S_FALSE.

```c
function ResetIteration: HRESULT; stdcall;
function Next(var name: PChar; var value: IUnknown): HRESULT; stdcall;
```

Iterations cannot be nested.

```c
function FindIID(name: PChar; const iid: TGUID; var id: IUnknown): HRESULT; stdcall;
```

B.3 IScriptEngine

The IScriptEngine interface contains methods related to many of the features provided by the COPDA engine. A complete description of all methods is outside the scope of this text. Refer to the COPDA software development kit instead.

Several of the methods in this interface do not use return values to indicate an error, but use exceptions instead.

Some of the methods use the TModuleSection enumeration to indicate a module section. This type is used in several other interfaces as well.

```c
TModuleSection = (msModule, msUses, msDeclare, msStartup, msInit, msOper,
  msFinal, msShutdown);

IScriptEngine = interface(IIdContainer)
  ['{A215A920-6913-11D3-8EEE-00201881BED0}']
  function Reset: HRESULT; stdcall;
  procedure AddNamedLibrary(name: PChar); stdcall;
  procedure AddLibrary(lib: IScriptLibrary); stdcall;
end;
```
Appendix B: Interfaces

// I/O configuration: access the console, access the currently active
// input device, access the set of known input devices, fix properties of
// the input device, access the set of output devices. New input devices
// or output devices are registered (made accessible by the COPDA engine) by
// adding them to the output set or input set. After the COPDA engine is
// created, adding at least one input device is the first thing to do.

function GetConsole(var cons: IConsole): HRESULT; stdcall;
procedure GetInput(var res: IInputDevice); stdcall;
procedure GetInputs(var res: IIdContainer); stdcall;
procedure SetInput(index: IInputDevice); stdcall;
procedure InputsConfigured; stdcall;
procedure GetOutputs(var res: IOutputSet); stdcall;

// Time information.
// Most of these functions only return valid information when executing
// the modules' operation sections. TDateTime is a floating point number
// counting the number of days since Jan 1st 1900.
function StartTime: TDateTime; stdcall;
function CurrentTime: TDateTime; stdcall;
procedure BaseRate(var secs, nanosecs: Integer); stdcall;
function BaseRate2: TDateTime; stdcall;
procedure SampleIndex: Integer; stdcall;

// Debugging support / error handling
procedure LogError(m: IModule; sec: TModuleSection; line, pos, len: Integer;
  p: PChar); stdcall;
procedure ClearError; stdcall;
function ErrorInfo(var m: IModule; var sec: TModuleSection;
  var line, pos, len: Integer; var p: PChar): LongBool; stdcall;
  // Returns True when valid

// Parsing of module files. See section B.5 for the IReadStream2 interface
procedure LoadModules(src: IReadStream2); stdcall;

// Accessing the 'run mode'. Setting the run mode to true activates the
// second stage of the CPMDL parser. Some other methods can only be called
// when run mode is true, while others can only be called when run mode is
// false.
function GetRunMode: LongBool; stdcall;
procedure SetRunMode(bStart: LongBool); stdcall;

// Saving the modules to a DWM stream (see section B.5)
procedure SaveModules(var dst: IReadStream2); stdcall;

// Execution of module sets (requires 'run mode' to be set to 'true' first):
procedure ExecuteSection(sect: TModuleSection); stdcall;
// Execute one of the executable sections
// (Startup, Initialization, Operation, Finalization, Shutdown)
function GetCurrentSection: TModuleSection; stdcall;
// Currently executing section (or msModule when not executing)
procedure PrepareSample; stdcall;
// Allows engine to 'prepare' a new sample. Used to cache sample
// time from input device. This is called after the input device
// sample has been loaded, but before the output has been
// prepared.

// Accessing the currently executing module or currently executing node
// (see section B.8). Note that GetCurrentModule and GetCurrentExec
// use a non-standard calling convention and can only be used in Delphi
// applications. Use the 'GetCurrentModuleV' or 'GetCurrentExecV'
// for other applications.
function GetCurrentModule: IModule; // *** ONLY USABLE IN DELPHI !
procedure GetCurrentModuleV(var res: TModule); stdcall;
procedure SetCurrentModule(im: IModule); stdcall;
function GetCurrentExec: IExecNode; // *** ONLY USABLE IN DELPHI !
procedure GetCurrentExecV(var res: IExecNode); stdcall;
procedure SetCurrentExec(ix: IExecNode); stdcall;
Appendix B: Interfaces

function GetCurrentExecX: Pointer; stdcall;
procedure SetCurrentExecX(ix: Pointer); stdcall;

// Support for Aborting execution
function GetAbortFlag: LongBool; stdcall;
procedure SetAbortFlag; stdcall;

// Checking whether there were changes that need saving (enabling Save entry in file menu)
function GetChangeFlag: LongBool; stdcall;
procedure ClearChangeFlag; stdcall;

// COPDA tools
procedure MakeDataValue(var res: IDataValue); stdcall;
// Creates a blank datavalue component (section B.6)
procedure NewModule(var m: IModule); stdcall;
// Creates a virgin module. Use m.LoadStream to set the basic properties.

// Information about virtual channels
procedure GetChannelMaster(var res: IChannelMaster); stdcall;

// Profiling support
function ProfileTime: Double; stdcall;
// Returns time spent executing between entering and leaving run mode.
// The time is reported in processor clock ticks.

// Loop callback
// The host application can specify a callback to be performed in each loop iteration (or during other 'busy' periods).
// Suggested use: check whether abort flag needs to be set.
procedure SetLoopCallback(cb: ICallback); stdcall;
procedure ExecuteLoopCallback; stdcall;

// Access to application window handle (useful as parent for dialog boxes)
function GetWindowHandle: HWND; stdcall;
end;

B.4 IConsole

The IConsole interface is used to represent a 'console': a device that can display messages to the user. A component that implements IConsole should be provided by the application when the COPDA engine is created. The methods of this interface should never raise an exception. When a method is not implemented, calling it should fail silently, returning S_FALSE instead of the normal S_OK.

IConsole = interface(IUnknown)
    ['25062824-8840-11d1-9f1e-0020af397a74']
    function PrintNoLine(txt: PChar): HRESULT; stdcall;
    function Print(txt: PChar): HRESULT; stdcall;
    function Clear: HRESULT; stdcall;
    function ShowStatusMsg(txt: PChar): HRESULT; stdcall;
    // This function may display a (possibly fast changing) status message. It may also be ignored. It is intended to be used in windowed applications that have a status bar.
    function PopupMsg(txt: PChar): HRESULT; stdcall;
    // This method, when implemented, should pop up a dialog box with the given message. While the dialog shows, the execution of the application is HALTED, but continued afterward.
    // This is intended for warnings and errors that are serious enough for the temporary halt in execution, but not serious enough for the abortion of all processing (use exceptions for that purpose)
end;
Appendix B: Interfaces

B.5 IReadStream2

The IReadStream2 interface is used to represent a seekable read-only character stream. COPDA’s CPMDL parser uses this type as a generic representation of character streams (instead of files). This allows the parser to access data from other sources than disk-based files, e.g. a resource in the COPDA program’s executable.

IReadStream2 = interface (IUnknown) [
'{A523CCC0-A789-11D1-9F1E-0020AF397A74}]
function Read(p: Pointer; cntReq: Longint;
var cntRead: Longint): HRESULT; stdcall;
// returns S_FALSE when less data was read than requested,
// S_OK on errors, S_OK if everything went ok.

function EndOfFile: HRESULT; stdcall;
// Returns S_OK on EOF, S_FALSE otherwise

function Size(var sz: Integer): HRESULT; stdcall;
function Rewind: HRESULT; stdcall;
function Seek(offset, Origin: Integer): HRESULT; stdcall;
function Position(var offset: Integer): HRESULT;

and;

B.6 IDataValue

The IDataValue interface is used to represent objects that have a value. IDataValue is used as the uniform data type in COPDA: all values processed by COPDA are represented by IDataValue interfaces. This interface can contain one of seven data types. Whenever possible, type conversions are automatic, so accessing a data value that contains an integer (loaded using SetInteger) as a string (using GetStringConst) automatically converts the integer to a string. Note that the string returned by the GetStringConst method is a temporary value that is only valid until the data value is accessed next.

TCOPDAType = (dtpEmpty, dtpInteger, dtpBoolean, dtpFloat, dtpDate, dtpString, dtpInterface);

IDataValue = interface (IUnknown) [
'{55B90EC0-8844-11d1-9F1E-0020AF397A74}]
function PrimaryType: TCOPDAType; stdcall;
function Valict: LongBool; stdcall;
procedure Clear; stdcall;
procedure CopyTo(v: IDataValue); stdcall;
// copies values from Self into V
function GetInteger: Integer; stdcall;
procedure SetInteger(v: Integer); stdcall;
function GetFloat: Double; stdcall;
procedure SetFloat(v: Double); stdcall;
function GetBool: LongBool; atdc:all;
procedure SetBool(v: LongBool); stdcall;
function GetDate: Double; stdcall; // TDateTime for Delphi
procedure SetDate(v: Double); stdcall;
function GetIntDate: Integer; stdcall; // UNIX-style timestamp
procedure SetIntDate(v: Integer); stdcall;
procedure GetInterface(var v: IUnknown); stdcall;
procedure SetInterface(v: IUnknown); stdcall;
function GetStringConst: PChar; stdcall;
function GetStringBuf(buf: PChar; len: Integer): Integer; stdcall;
procedure SetString(s: PChar); stdcall;

end;
Appendix B:  
Interfaces

B.7 IExecute

The IExecute interface represents any object that can be ‘executed’ in COPDA: CPMDL functions, methods, variables, and a few internal objects. This interface is of particular interest to library (COPDA extension) programmers, as it defines the format that COPDA uses to execute any functions (including functions imported from external libraries). Each executable object in CPMDL has two features: a name and a function that performs the action that ‘executing’ the object entails.

IExecute = interface(IUnknown)
['{6F64B422-8CF2-11d1-9F1E-0020AF397A74}']
  procedure Execute(node: IExecNode; context: IIdContainer); stdcall;
  // This function can use exceptions to indicate errors.
  // See section B.8 for a description of IExecNode.
  // The 'context' argument points to the COPDA engine component.
  // The CPMDL return value of executing this object should be
  // stored in the 'node' argument.

  function GetName: PChar; stdcall;
end;

B.8 IExecNode

The IExecNode interface represents the nodes in the CPMDL parse tree. It is a subinterface of IDataValue (section B.6), so each node acts as storage for a COPDA value. Children of an execution node represent the arguments to CPMDL functions. Each execution node has a functionality (an IExecute interface, see section B.7) that defines the action to be performed when the node is executed. The functionality is a nil pointer for nodes that represent constants.

IExecNode = interface(IDataValue)
['{6F64B420-8CF2-11d1-9F1E-0020AF397A74}']
  function ChildCount: Integer; stdcall;
  procedure Child(index: Integer; var res: IExecNode); stdcall;
  // raises exception for invalid indices

  procedure Functionality(var res: IExecute); stdcall;
  procedure StdExec(context: IIdContainer); stdcall;
  // Shorthand for Functionality.Execute(Self, context);

  procedure Fix; stdcall;
  // Removes functionality; making this node a constant

  procedure SetFunctionality(x: IExecute); stdcall;
  // Modifies functionality

  procedure GetLocation(var sec: TModuleSection;
                var line, pos, len: Integer); stdcall;
  // Retrieves location in CPMDL source (for error reporting)
end;

B.9 Input and output

Input and output devices (‘data sources’ and ‘data destinations’) each use two interfaces for communication with COPDA. Both devices use the IDataRecord interface to represent a set of simultaneously available sample values. Input devices additionally use the InputDevice interface to provide access to input control and configuration. Output devices support the IOutputDevice interface (in addition to the IDataRecord interface) for configuration and control. In this section only a terse listing of these interfaces is provided. For a full description, see the COPDA software development kit.
IDataRecord = interface (IUnknown)
  ['{678446C0-8848-11D1-9F1E-0020AF397A74}']
  // 'channel' can be either a device-dependent 'physical' channel
  // code or a 'virtual channel code' that is valid for all devices:
function ReadValue(channel, subchannel: Integer);
  value: IDatavalue): HRESULT; stdcall;
function WriteValue(channel, subchannel: Integer;
  value: IDatavalue): HRESULT; stdcall;
function TimeStamp: TDateTime; stdcall;
function BaseRate(var secs, nanosecs: Integer): HRESULT; stdcall;
  // Convert channel and subchannel codes to names and vice versa
function ParamCode(nm: PChar): Integer; stdcall;
function ParamName(param: Integer): PChar; stdcall;
function IndexCode(nm: PChar): Integer; stdcall;
function IndexName(idx: Integer): PChar; stdcall;
  // Iterate over all defined physical channels
function FirstParam: Integer; stdcall;
function NextParam(param: Integer): Integer; stdcall;
  // returns 0 at end (0 is never a valid parameter code)
function SubChannelCount(ch: Integer): Integer; stdcall;
  // Convert physical channel codes to virtual codes
function VirtualCode(channel: Integer): Integer; stdcall;
end;

IInputDevice = interface (IUnknown)
  ['{C4990140-C03F-11D2-9F1E-0020AF397A74}']
function Info(key: PChar): PChar; stdcall;
  // Returns information about the device.
  // When the key is not recognized, 'Info' returns NULL.
  // Currently defined keys:
  // 'device' : returns the name of the device (REQUIRED)
  // 'file' : returns the name of the currently opened file (OPT.)
function Configure(configstring: PChar): HRESULT; stdcall;
  // Loads new configuration. Only resets device when configstring=nil.
  // The new configuration completely replaces the existing one.
function ConfigurationDialog(cfgin: PChar;
  var cfgout: PChar): HRESULT; stdcall;
function DataStatus(var status: TDataStatus): HRESULT; stdcall;
function DataDone: HRESULT; stdcall;
function StartSequence: HRESULT; stdcall;
function LockRecord: HRESULT; stdcall;
function FlushSequence(reset: LongBool): HRESULT; stdcall;
procedure BindChannels; stdcall;
  // Creates virtual channels for all input channels.
function SaveConfig(cfg: PChar); stdcall;
function RetrieveConfig: PChar; stdcall;
end;
Appendix B:
Interfaces

IOutputDevice = interface(IUnknown)
['{C4990141-C03F-11D2-9F1E-0020AF397A74}']

function Info(key: PChar): PChar; stdcall;
// Returns information about the device.
// When the key is not recognized, 'Info' returns NULL.
// Currently defined keys:
// 'device': returns the name of the device (REQUIRED)
// 'file': returns the name of the currently opened file (OPT.)

function Status: TOutputStatus; stdcall;

procedure SetVisibility(bVisible: LongBool); stdcall;

function Configure(configstring: PChar;
input: IInputDevice;
context: IIdContainer): HRESULT; stdcall;
// Loads or modifies the configuration. Resets device when
// configuration=nil. Only in that case the new configuration string
// replaces the existing configuration, otherwise the existing
// configuration is modified.

function ConfigurationDialog(cfgin: PChar;
var cfgout: PChar;
context: IIdContainer): HRESULT; stdcall;
// Return values:
// S_OK: value changed
// S_FALSE: dialog canceled or cfg was left unchanged
// This function should not change anything in the device.

procedure SaveConfig(cfg: PChar); stdcall;

function RetrieveConfig: PChar; stdcall;

function NewSequence(input: IInputDevice;
context: IIdContainer): HRESULT; stdcall;

function EndSequence(input: IInputDevice;
context: IIdContainer): HRESULT; stdcall;

function NewSample(input: IInputDevice;
context: IIdContainer): HRESULT; stdcall;

function EndSample(input: IInputDevice;
context: IIdContainer): HRESULT; stdcall;

end;
Appendix C: CPMDL syntax

C.1 Introduction

This appendix provides the syntax definition of the CPMDL language. This first section introduces the concepts used in CPMDL, the following two sections contain the formal syntax definition of CPMDL.

A CPMDL file describes one or more modules. Each module description starts with a module header section. This is followed by the following sections, each of which is optional: a library section (also referred to as uses section), a declarations section, a startup section, an initialisation section, an operation section, a finalisation section and a shutdown section. The start of each section is indicated in CPMDL by one of the following keywords: module, uses, declare, startup, initialise, operation, finalise and shutdown (the keywords initialise and finalise can alternatively be spelled initialize and finalize).

The header section is used to indicate a module’s name and describes its connections with the outside world. The library section is used to indicate any COPDA libraries the module relies on. The declaration section is used to declare COPDA variables. The contents of the last five sections describe actions that can be performed by a module and share a common syntax. More information on how the sections are used can be found in section 4.8 of this thesis.

COPDA’s CPMDL parser first converts the stream of characters that represents the CPMDL source into a stream of tokens. This tokenisation is described in section C.2. Next, the parser interprets this token stream according to the syntax that will be presented in section C.3.

Whitespace characters have no syntactical meaning in the CPMDL syntax. The tokeniser removes all whitespace (including comments) from the input stream. Of course, the tokeniser itself uses whitespace to separate subsequent tokens. Newline characters are also whitespace and therefore have no syntactical meaning in the CPMDL syntax. The tokeniser does use newline characters though (a newline character is used to terminate comments). Though not required by the CPMDL syntax, it is good habit to put the keywords that start a new section at the start of a line. The CPMDL editor built into DataWand actually requires this for correct operation.

C.2 Lexical elements

The lexical analyser part of the CPMDL parser transforms the character stream containing the CPMDL source into a token stream by recognising the following pseudo regular expressions. Comments in these rules are written in italic text, appearing before the rule they apply to.
Appendix C:
CPMDL syntax

The only 'keywords' in CPMDL are the section start identifiers. Note that statement names (such as 'if') are not considered to be keywords.

**KEYWORD**

- 'module'
- 'uses'
- 'declare'
- 'startup'
- 'initialization'
- 'initialisation'
- 'operation'
- 'finalization'
- 'finalisation'
- 'shutdown'

**INT**

```
= [0-9]+ 
```

**ID**

```
= [a-zA-Z_][a-zA-Z_0-9]* 
```

'Signals' are identifiers prefixed with a hash mark. They are used to represent module inputs and are mostly treated equivalent to variable names.

**SIG**

```
= '#'[a-zA-Z_0-9]* 
```

Variable names must be prefixed with a dollar sign in CPMDL.

**VAR**

```
= '$'[a-zA-Z_][a-zA-Z_0-9]* 
```

String constants are enclosed in double quotes. In addition to the following definition the following rules apply: (1) string constants can not span multiple lines (newline characters are forbidden in strings). (2) The standard C-style escape sequences are recognised: the sequence '\"' indicates an embedded double quote and '\n' is used to embed a newline character inside a string constant.

**STRING**

```
= ".*" 
```

**SEMI**

```
= ';' 
```

**POPEN**

```
= '(' 
```

**PCLOSE**

```
= ')' 
```

**BOPEN**

```
= '{' 
```

**BCLOSE**

```
= '}' 
```

**TILDE**

```
= '~' 
```

**EQUAL**

```
= '=' 
```

**DOT**

```
= '.' 
```

**AT**

```
= '@' 
```

The end-of-file marker of the character stream is treated as a separate token.

**EOF**

```
= <eof> 
```

CPMDL comments start with an exclamation mark and extends to the end of the line.

**COMMENT**

```
= '!'\n 
```

Whitespace is any sequence of one or more spaces, tab characters, newline characters and comments.

**WHITE**

```
= ( \n\r<COMMENT>\n)+ 
```

Any characters that do not match any of these rules (e.g. a '+' sign outside a string or comment) is considered to be a syntax error.

In the token stream sent from the tokeniser to the parser, all whitespace is removed, so in the syntax specification presented in the next section, WHITE and COMMENT do not appear.
Appendix C:  
CPMDL syntax

C.3 Syntax

C.3.1 Notation

The CPMDL syntax is described below using a variation of the BNF notation. The following conventions are used:

- Words written in capitals indicate the tokens (terminals) received from the tokeniser.
- Other words indicate non-terminals.
- Nonterminal names starting with an underscore character indicate optional non-terminals.
- Text between ‘triangular brackets’ (‘<...>’) indicates a context condition that must be satisfied to apply the rule. This is used to disambiguate choices. In the CPMDL syntax specification this mechanism is used to disambiguate all choices.

The context conditions are used as in the following example:

\[
\text{example} = \langle \text{context}=\text{STRING}\rangle \text{ stringconst} \\
| \langle \text{context}=\text{INT}\rangle \text{ numconst} \\
| \text{SEMI}
\]

This tells the parser to parse an occurrence of the nonterminal example as the nonterminal stringconst when the following token is a STRING, or as the nonterminal numconst when the following token is an INT. If the following token is not one of STRING or INT, it must be a SEMI and the parser can advance to the next token in the token stream. In the first two cases, the parser does not advance to the next token in the token stream, but the STRING or INT will still be the ‘following token’ when the stringconst or numconst nonterminal is parsed.

Sometimes the context condition has a longer form, where not only the type of the following token is checked, but its value as well:

\[
\text{_init} = \langle \text{context}=\text{KEYWORD}, \text{value}=\text{‘initialization’}\rangle \text{ kw}_\text{init} \text{xblock} \\
| \astempty*
\]

When superfluous, the ‘context=’ attribute can be omitted. The text *empty* is used to indicate that a nonterminal can be empty, not advancing the token stream at all.

The context condition is also used when describing repetition of nonterminals (as usual in BNF variants, repetition is indicated by enclosing the repeated tokens and nonterminals in curly brackets). For instance, the following definition defines the nonterminal cpmdlfile as comprising zero or more moduledefs. As long as the next token in the token stream is the keyword module, the parser can start parsing a moduledef. When the next token is not the keyword module after parsing the moduledef, the following token must be EOF (that is: the input should be finished).

\[
\text{cpmdlfile} = \langle \text{context}=\text{KEYWORD}, \text{value}=\text{‘module’}\rangle \text{ moduledef} \ast\text{EOF}
\]

When the CPMDL parser encounters a known identifier (a token of type ID or VAR), the parser can look up the type of the object represented by that identifier. This information is sometimes used in the context conditions; the context conditions have a ‘class=’ attribute.

Attributes in context conditions can use inverted conditions, using the ‘!=’ symbol to indicate that an attribute should not have the specified value. For instance, the following rule defines a liblist to be a semicolon-terminated list of librarynames:

\[
\text{liblist} = \langle \text{context}!=\text{SEMI}\rangle \text{ libraryname} \ast\text{SEMI}
\]
C.3.2 Syntax of CPMDL

Here is the CPMDL syntax definition. Comments start with a semicolon and are printed in italics.

```plaintext

cpmdlfile = \{<context=KEYWORD,value='module'> moduledef\} EOF
moduledef = moduleheader defsections execsections
moduleheader = kw_module modulename _buffersize _modargs _location _description SEMI
  kw_module = <value='module'> KEYWORD
  modulename = ID ; define new module
  _buffersize = <context=TILDE> TILDE buffersize
    | *empty* ; module has no output
  buffersize = INTEGER ; in current version, the integer must be '1'
  _modargs = <context=OPEN> POPEN {<context!=PCLOSE> modarg} PCLOSE
    | *empty* ; empty _modargs means the module has no inputs
  modarg = argname _conndef
  _conndef = <context=EQUAL> conndef
    | *empty*
  argname = SIG
  conndef = EQUAL modulename ; reference to existing module
  _location = <context=AT> AT POPEN xcoord ycoord width height PCLOSE
    | *empty* ; default location
  xcoord = INTEGER
  ycoord = INTEGER
  width = INTEGER
  height = INTEGER
  _description = <context=STRING> STRING
    | <context=SEMI> *empty* ; default description (empty string)
  defsections = _lib _storage
  _lib = <context=KEYWORD,value='uses'> kw_lib liblist
    | *empty*
  kw_lib = <value='uses'> KEYWORD
  liblist = {<CONTEXT!=SEMI> libraryname} SEMI
  _storage = <CONTEXT=KEYWORD,value='declare'> kw_storage storagedef
    | *empty*
  kw_storage = <value='declare'> KEYWORD
  storagedef = vartype {<context!=SEMI> vardef} SEMI
  vartype = classname
  classname = <context=ID, class=ICLASS> ID
  vardef = varname _initializer
  _initializer = <context=EQUAL> initializer
    | *empty*
  initializer = EQUAL STRING ; variables can only be initialized with strings
  varname = VAR ; define new variable
  execsections = _start _init _operation _final _shut
  _init = <context=KEYWORD,value='initialization'> kw_init xblock
    | *empty*
  kw_init = <value='initialization'> KEYWORD
    | <value='initialisation'> KEYWORD
```

138
Appendix C:
CPMDL syntax

\_start
\_\_operation
\_\_final
kw_start
kw_operate
kw_final
kw_shut

; Contents of the executable sections:

Note the double appearance of func/s_func, method1/s_method1, method2/s_method2: the s_* version is the version with parenthesis; around the arguments (parenthesis are only required in subexpressions; in top-level expressions they are optional)

xblock = \{<context!=KEYWORD, context!=EOF> statement\}
block = BOPEN \{<context!=BCLOSE> statement\} BCLOSE
operate = <value='operate'> KEYWORD
statement = <context=ID,context=ISTATEMENT> blockstatement
| atomicstat
blockstatement = <context=ID,value="if"'> ifstat
| <context=ID,value="ifelse"> ifelsestat
| <context=ID,value="while"> whilestat
| <context=ID,value="dowhile"> dowhilestat
ifstat = if condition block
if = <value="if"> ID
condition = anyexpr
anyexpr = <context=STRING> stringconst
| <context=INT> numconst
| <context=ID,context=IFUNCTION> func
| <context=ID,context=ICLASS> method2
| <context=SIG> argname
| <context=VAR> varname _var_or_method

stringconst = STRING
numconst = INT
func = funcname funcargs
funcargs = POPEN \{<context!=PCLOSE> argument\} PCLOSE
funcname = <class=FUNCTION> ID
argument = anyexpr
_var_or_method = <context=DOT> method1
| *empty*
method1 = DOT methodname funcargs
method2 = classname DOT methodname funcargs
Appendix C:
CPMDL syntax

methodname  =  <context=ID,('classname').(methodname)> is a known function> ID
; classname was either given explicitly (s_method2, method2) or is
; derived from the variable name 'varname'
; (s_method1←s_var_or_method←atomicstat,
 ; method1←_var_or_method←anyexpr)
ifelsestat  =  ifelse condition block block
ifelse     =  <value="ifelse"> ID
whilestat  =  while condition block
while      =  <value="while"> ID
dowhilestat  =  dowhile block condition SEMI
dowhile    =  <value="dowhile"> ID
atomicstat =  <context=STRING> stringconst SEMI
 | <context=INT> numconst SEMI
 | <context=ID,class=IFUNCTION> s_func SEMI
 | <context=ID,class=ICLASS> s_method2 SEMI
 | <context=SIG> argname SEMI
 | <context=VAR> varname s_var_or_method SEMI
s_func     =  funcname s_funcargs
s_funcargs =  <context=POPEN> POPEN {<context!=PCLOSE> argument}
PCLOSE
 | {<context!=SEMI> argument}
s_var_or_method= <context=DOT> s_method1
 | *empty*
s_method1  =  DOT methodname s_funcargs
s_method2  =  classname DOT methodname s_funcargs
Appendix D: Gap-friendly first order lowpass filter

This appendix describes a filter that mostly reacts as an infinite impulse response first order lowpass filter, but can operate on signals whose values can be normal numbers or an invalid marker.

Not all standard filters or other signal processing algorithms can be usefully converted to accept invalid values. Finite Impulse Response (FIR) filters for instance (formula D.1) can be converted using trivial conversion rules (value * invalid = invalid; value + invalid = invalid). However, if an invalid sample occurs as input to a FIR filter with N taps, the following N output samples will all have the value invalid: the formula for FIR filters will return invalid if any of its input samples x(t-1) has the value invalid. The net effect is that each gap in the input will become far larger. Simply skipping invalid input values in the summation is not a solution in this case, as that is equivalent to replacing the invalid values with valid zero values, effectively applying the filter to a different input signal.

\[
y_n = \sum_{i=0}^{N-1} a_i \cdot x_{n-i} \quad \text{(formula D.1: FIR filter)}
\]

For an Infinite Impulse Response (IIR) filter (formula D.2) the situation is even worse: if an invalid sample occurs at the filter's input, its output will forever be invalid due to the output feedback (when trivial conversion rules are used).

An attempt to convert FIR and IIR filters to a form that is still useful when allowing invalid input values can be based on redefining the summations in formulas D.1 and D.2 to skip any summands that have the value invalid. Usually this will change the filter characteristic enough to make the filter unusable. This is especially true when the operation of the filter heavily depends on mutual cancellation of data samples. There is one subgroup of IIR filters that can be reasonably converted though: first order IIR low-pass filters.

The generic formula for such a filter is:

\[
y_n = \alpha \cdot y_{n-1} + (1-\alpha) \cdot x_n \quad \text{(with } 0<\alpha<1)\]

Here \( \alpha \) is the 'smoothing factor'. The coefficient of \( x_n \) is chosen such that if \( x_n \) is constant, \( y_n \) eventually will become equal to \( x_n \). Assuming \( y_{-\infty} \) to be 0, this formula can be rewritten as:

\[
y_n = \alpha \cdot y_{n-1} + (1-\alpha) \cdot x_n
\]

\[31\text{ it is actually not valid to talk about a filter characteristic, because this modified filter is no longer linear or time-invariant}\]
Appendix D:  
Gap-friendly first order lowpass filter

\[ y_n = (1 - \alpha) \cdot \sum_{i=0}^{\infty} \alpha^i \cdot x_{n-i} \Rightarrow y_n = \frac{\sum_{i=0}^{\infty} \alpha^i \cdot x_{n-i}}{\sum_{i=0}^{\infty} \alpha^i} \quad \text{(formula D.3)} \]

Formula D.3 can be interpreted as an average of an infinite number of samples, using an exponentially decaying scale function: it is the sum of the sample values multiplied by the scale factors \( \alpha^i \), divided by the sum of the scale factors. This formula can be modified to introduce a weight factor for each sample as well, allowing modifying the influence of some samples to the result:

\[ y_n = \frac{\sum_{i=0}^{\infty} \alpha^i \cdot w_{n-i} \cdot x_{n-i}}{\sum_{i=0}^{\infty} \alpha^i \cdot w_{n-i}} \quad \text{(formula D.4)} \]

Formula D.4 can be used in a setting including invalid values by setting \( w_{n-i} \) to 0 for invalid samples and setting it to 1 for valid samples: the \( w \) values can be interpreted as validity values. In addition the definition of multiplication needs to be changed to:

\[
\begin{align*}
    x \cdot \text{invalid} &= \begin{cases} 
        \text{invalid} & \text{if } x \neq 0 \\
        0 & \text{if } x = 0
    \end{cases} \\
    x \cdot \text{valid} &= \begin{cases} 
        \alpha \cdot z_{n-1} & \text{if } x_n \text{ is invalid} \\
        \alpha \cdot z_{n-1} + x_n & \text{if } x_n \text{ is valid}
    \end{cases} \\
    v_n &= \begin{cases} 
        \alpha \cdot v_{n-1} & \text{if } x_n \text{ is invalid} \\
        \alpha \cdot v_{n-1} + 1 & \text{if } x_n \text{ is valid}
    \end{cases}
\end{align*}
\]

initialisation: \( z_0 = 0; v_0 = 0 \)  

(formula D.6)

The resulting formula no longer describes a pure first-order low-pass IIR filter - though it does so when all input samples are valid -, but it describes a filter that calculates the average of all previous valid samples, scaled by exponentially decaying scale factors. For many applications this is a good approximation of a first order low-pass IIR filter. Like a normal IIR filter, the filter of formula D.4 can be calculated incrementally. This is performed by separately calculating the numerator and denominator incrementally (formula D.6).

\[ y_n = \frac{z_n}{v_n}, \text{ with} \]

\[
\begin{align*}
    z_n &= \begin{cases} 
        \alpha \cdot z_{n-1} & \text{if } x_n \text{ is invalid} \\
        \alpha \cdot z_{n-1} + x_n & \text{if } x_n \text{ is valid}
    \end{cases} \\
    v_n &= \begin{cases} 
        \alpha \cdot v_{n-1} & \text{if } x_n \text{ is invalid} \\
        \alpha \cdot v_{n-1} + 1 & \text{if } x_n \text{ is valid}
    \end{cases}
\end{align*}
\]

The drawback of this implementation is that once a valid sample is seen, the filter output will never return to invalid again: gaps are not just made smaller but disappear entirely. However, one can consider \((1-\alpha)z_n\) as a measure of the validity of the output: it will be zero if no valid sample was ever processed, and it would be one if all samples would be valid (a situation that never occurs since \( v_0 \) is initialised to 0). One can extend the filter algorithm by defining the filter's output as being invalid if \((1-\alpha)z_n\) drops below a chosen validity threshold (e.g. 0.5).
Appendix E: Configuration Specifications

E.1 Introduction

This appendix details various aspects related to PDS files and data source specific features. PDS files are used to store configuration options for data source components. In principle, the contents of a PDS is only meant to be understood by one data source component. To improve interoperability and to allow COPDA programs (such as DataWand) to determine for which data source component a PDS file is intended a certain level of standardisation is desired. Below two groups of guidelines for PDS files are specified. The first set of rules are rules that PDS file contents must adhere to. The second set of rules are guidelines that may ease understanding the intention of the various configuration options.

E.2 PDS file requirements

1. A PDS file is a text file.
2. Each line in the PDS file specifies a single 'configuration option'. What a 'configuration option' is and what its syntax is, is up to the data source component.
3. The first line in the PDS file must have the following structure:
   *device=<device name>
   Here, '<device name>' should be replaced with the name of the input device (data source component) the PDS file is intended to be used with. There should be no spaces around the '=' sign.
   This line is used by COPDA applications to ensure the correct data source component is activated.

E.3 PDS file suggestions

For many devices three types of 'configuration options' can be identified: filenames, settings and markers.

1. Each filename should appear on a single line, without any other text.
2. Each setting should be on a single line using the following syntax:
   *<option name>=<option value>
   The idea is that each possible option name appears at most once in a PDS file
3. Each marker should be on a single line and uses the following syntax:
   *<marker name>
   Markers may appear multiple times in a single PDS file. They can be used for instance to delimit groups of related files.
E.4 Input device specifics

E.4.1 The following options are defined for the HPDATA input device:

*device=HPDATA (file input)

The required PDS ‘header’ line

filename

The HPDATA input device reads SBF files (with the extension .pat). Each file to be processed appears on a separate line. The file names should not include path information. Use the *dir option to specify the path data files appear in.

*seq

This marker indicates the start of a new sequence. Subsequent filenames (up to the following appearance of *seq) form a sequence of consecutive files with data for one patient. For each sequence, all module’s initialisation and finalisation sections are executed.

*dir=<absolute path>

Required. This specifies the path subsequent data files are searched in. The path string should not include the final ‘\’.

$dbase=<indexfile>

Optional. This specifies the patient database used to find data files for patients identified by *name and/or *id. This information is used by DataWand’s HPDATA configuration dialog box.

$name=<patientname>

Optional. These specify overrides for the patient name and identifier for the following data files. Can appear multiple times. A *seq resets the overrides.

E.4.2 The following options are defined for the EDF input device:

*device=EDF input device

The required PDS ‘header’ line

*timeunit=<integer>

The number of milliseconds between subsequent ‘clock ticks’. All signals are emitted at an integer multiple of *timeunit. Normally, timeunit will be set to one millisecond.

*maxcount=<integer>

The maximum number of ‘clock ticks’ to process data. This is used together with *offset to specify a subrange within the data file.

*offset=<integer>

The number of timeunits to skip before processing data starts. This is used together with *maxcount to specify a subrange within the data file.

filename

The EDF input device reads EDF files (with the extensions .rec, .edf, and several others). The file to be processed appears on a single line. The file names can include path information. PDS files for the EDF input device can currently specify only one data file.
Bibliography

[de Clercq et al, 99a]

[de Clercq et al, 99b]

[Cluitmans, 96]

[Cuijpers, 99]

[Cunningham et al, 97]

[Gravenstein et al, 89]

[HL7]

[HP78360A]

[HP78360B]

[IBIS]

[Imhoff, 92]
Bibliography

[Inside COM]

[Introduction to Algorithms]

[Jain et al, 97]

[Kemp et al, 92]

[Kennelly et al, 97]

[Lau et al, 93]

[MIB]

[Paredaens et al]

[Pablo, 99]
B. de Pablo Lapiedra. ST segment as a possible predictor for myocardial infarction in an intensive care unit. Eindhoven University of Technology, Department of Electrical Engineering MSc report. April 1999.

[RFC 768]

[RFC 791]

[RFC 793]
Bibliography

[RFC 2068]

[Saranummi et al, 96]

[Schillings, 95]

[Stevens et al]

[Taboada et al I, 97]

[Taboada et al II, 97]

[Tanenbaum]

[UI Design]

[Vilhelm et al, 96]
Samenvatting

In de intensive care afdeling van een ziekenhuis worden vele variabelen bewaakt voor iedere patiënt. Dit proefontwerp beschrijft het ontwerp van software voor het vergaren en analyseren van zulke gegevens. Het werk besproken in dit proefontwerp bestaat uit twee delen.

Het eerste deel is het data acquisitie systeem, HPDATA genaamd, dat ik ontworpen heb voor de intensive care van het Catharina Ziekenhuis in Eindhoven. HPDATA vergaart continu de gegevens die door de patiëntbewakingsapparatuur gemeten wordt. De vergaarde data worden in twee vormen beschikbaar gesteld: in de vorm van netwerk pakketten en in de vorm van bestanden. Computers die op het local area netwerk van de intensive care aangesloten zijn kunnen de stroom van netwerk pakketten gebruiken voor real-time patiëntbewakings doeleinden. De bestanden kunnen nog tijdens het vergaren van de data gebruikt worden door software die artsen assisteert bij het beoordelen van de toestand van de patiënten. De bestanden kunnen later ook dienst doen in het verslag van het verblijf van een patiënt in de intensive care. Ook kunnen de bestanden een bron van informatie voor medisch onderzoek vormen.

Als voorbeeld van het gebruik van de gegevens die HPDATA verzamelt heb ik een programma met de naam HPView ontwikkeld. HPView voorziet artsen van informatie voor de evaluatie van de gezondheidstoestand van patiënten in de vorm van grafieken die inzicht verschaffen in het verloop van de meetwaarden in de tijd. Dit programma wordt sinds September 1996 gebruikt bij de dagelijkse patiëntbesprekingen.

Beide vormen van data (netwerkdata en bestanden) kunnen gebruikt worden door een verscheidenheid aan patiëntbewakings- en gerelateerde software. Het tweede deel van dit proefontwerp beschrijft de software die ik ontwikkeld heb om het ontwerpen en implementeren van dergelijke programma's te vereenvoudigen. Deze software bestaat uit twee onderdelen: een data processing core genaamd COPDA en het programma DataWand. Ofschoon beide oorspronkelijk bedoeld waren om de gegevens die met HPDATA vergaard zijn te bewerken, zijn ze hier niet toe beperkt, maar kunnen ze ook overweg met andere data formaten.

Het programma DataWand stelt gebruikers in staat om, via een grafische user interface, data bewerkingen algoritmen te ontwikkelen en te testen. DataWand wordt gebruikt om dergelijke algoritmen te implementeren en uit te voeren op offline gegevens (gegevens uit een bestand). Het resultaat van de bewerkingen op de data kan bestaan uit graafieken van de bewerkte signalen, boodschappen die tijdens de verwerking gegenereerd worden en bestanden waarin de bewerkte gegevens worden opgeslagen. Voor medisch onderzoek is deze functionaliteit vaak al voldoende. In een ruimere context beschouwd kan DataWand ook gebruikt worden als prototyping en ontwikkel omgeving voor databewerkingalgorithmen.

COPDA is het ‘databewerkingshot’ dat DataWand zijn dataverwerkingsfunctionaliteit biedt. De primaire functie van COPDA is het interpreteren van databewerkingalgorithmen die in de taal CPMDL zijn beschreven (zulke beschrijvingen kunnen ontwikkeld worden m.b.v. DataWand). COPDA is geïmplementeerd als een COM component die in programma's die dataverwerkingsfunctionaliteit wensen te bieden ‘ingebakken’ kan worden. De twee belangrijkste richtlijnen tijdens het ontwerp van COPDA waren het maximaliseren van flexibiliteit en het minimaliseren van de invloed op de user interface.
COPDA *an sich* biedt in het geheel geen *user interface*, maar laat het presenteren van zijn functionaliteit naar de gebruiker over aan het programma dat COPDA gebruikt. Dit zorgt ervoor dat de ontwerper van zulke een programma niet met door COPDA opgelegde *beperkingen rekening hoeft te houden.*

De eis om de *flexibiliteit* te maximaliseren heeft veel aspecten van COPDA beïnvloed (en daardoor indirect ook van DataWand). De vorm waarin data aangeboden dient te worden en weer ‘afgevoerd’ wordt, is niet één of ander concreet bestands- of netwerk-formaat, maar is geabstraheerd in de concepten *data source* en *data destination.* Programma’s die COPDA gebruiken, dienen deze abstracties te implementeren om hun eigen data formaten te kunnen lezen en schrijven (of kunnen standaard implementaties gebruiken voor enkele bekende data formaten). Een ander voorbeeld van COPDA’s flexibiliteit is dat de verzameling functies en klassen die beschikbaar zijn in de taal CPMDL uitgebreid kunnen worden met ‘bibliotheken’ die nieuwe functies bevatten. Zulke ‘bibliotheken’ zijn DLL’s die in een conventionele programmeertaal (zoals C of Pascal) geïmplementeerd zijn.

Enkele van de eigenschappen van COPDA zijn:

- Ondersteuning voor het bewerken van data die (gmarkeerde) meetfouten bevat.
- Uniforme behandeling van verschillende soorten data: data uit bestanden (*offline data*) en data die als een *network stream* binnenkomt (*real-time data*) worden op dezelfde manier behandeld.
- Gesynchroniseerd bewerken van vele signalen, hetgeen *multi-signal processing* en *data fusion* vereenvoudigt.

Enkele van de eigenschappen van DataWand zijn:

- DataWand biedt twee naadloos geïntegreerde user interface stijlen om de dataverwerkingsalgorithmen te bewerken. Een grafische user interface maakt het mogelijk om dergelijke algoritmen te definiëren door blokken die databewerkingsprimitieven te plaatsen en met elkaar te verbinden. Een textuele user interface biedt de mogelijkheid om de in CPMDL beschreven functionaliteit van deze blokken aan te passen (of nieuwe functionaliteit te definiëren).
- DataWand bevat een krachtige ‘tool’ om grafieken van de invoer- en uitvoer-signalen te presenteren.
- Aangezien DataWand gebruik maakt van COPDA voor zijn databewerkingsfunctionaliteit, kunnen in DataWand ontworpen algoritmen in andere COPDA-gebaseerde programma’s gebruikt worden zonder dat daar verdere bewerkingsstappen zoals compilatie voor nodig zijn (zolang geen DataWand-specifieke functionaliteit gebruikt wordt).

DataWand is door stagiaires bij de sectie Medische Elektrotechniek van de capaciteitsgroep Meet- en Besturingssystemen aan de Technische Universiteit Eindhoven gebruikt in een project betreffende het voorspellen van hartinfarcten bij intensive care patiënten. Er wordt momenteel onderzocht of DataWand geschikt is als educatief gereedschap om niet-experts te introduceren in medische signaalverwerking.

DataWand en COPDA worden momenteel gebruikt binnen het Europese IBIS project als gereedschap voor het testen en implementeren van nieuwe databewerkingsalgorithmen. In dit project wordt een heel scala van algoritmen ontwikkeld, variërend van signaalvalidatie tot patroonherkenning. Het einddoel van IBIS is het ontwikkelen van een *IBIS workstation*, een systeem waar de ontwikkelde algoritmen in de praktijk gebruikt worden. Dit systeem zal gebruik maken van COPDA als databewerkingshart.
Curriculum Vitae

Luc Cluitmans was born in Swalmen, the Netherlands, on June 30th 1968. In 1986 he finished secondary school (Atheneum) at the Bisschoppelijk College Broekhin, Roermond. He obtained the Masters degree in electrical engineering from Eindhoven University of Technology in 1992. His graduation work "Using genetic algorithms for scheduling data flow graphs" was re-published as EUT report 92-E-266.

In September 1992 he started working as a research assistant at the Design Automation section of the department of Electrical Engineering of Eindhoven University of Technology. He worked on parameterised integrated circuit module generator software in the context of the European LINK project.

At the end of 1993 he moved to the Medical Electrical Engineering section to start developing data acquisition software for the European ISPOCD project.

In November 1994 he entered the two-year designer course ‘Information and Communication Technology’ at the Stan Ackermans Institute at Eindhoven University of Technology. He received the master of technological design degree in 1996. The result of the final design project of this designer course is presented in chapter 3 of this thesis.

After a few months, during which resumed programming for the ISPOCD project, he worked two more years as a PhD student, culminating in this thesis.

Since February 1999 he is also involved in the European IBIS project, where DataWand is used for data processing algorithm prototyping and COPDA will be used in the data processing platform.
Stellingen

behorende bij het proefontwerp

User Support for Patient Data Analysis

doort

Luc J.M. Cluitmans

1. Het opnieuw uitvinden van het wiel is niet altijd inefficiënt.
   (Deze dissertatie, paragraaf 2.7)

2. Een beschrijving in woorden doet een eenvoudige bewerking in een grafische gebruikers-interface complexer lijken dan hij is.
   (Deze dissertatie, hoofdstuk 5)

3. Trendgrafieken van aan intensive-care patiënten gemeten parameters vormen een waardevolle informatiebron tijdens patiëntbesprekingen.

4. Het idee om nieuwe medische databewerkingsalgorithmen uit te testen op 'artefactloze' data gaat uit van de in het algemeen foutieve veronderstelling dat zulke data een goede benadering zijn voor in de realiteit gemeten medische data.

5. Het programmeren van software met een grafische gebruikersinterface levert meer waardering op dan het programmeren van complexe software zonder zo'n gebruikersinterface, zoals drivers ("besturingsprogramma's") of interfaceloze COM componenten.

6. De RDTSC instructie, die het aantal klokcycli sinds het opstarten van een op de Intel Pentium gebaseerde processor geeft, is geschikt als entropiebron voor gebruik in cryptografisch verantwoorde toevalsgetalgeneratoren (en dan niet alleen voor pseudotoevalsgetalgeneratoren).

7. Juristen hebben moeite met het doorgronden van de praktijk van softwareontwikkeling, zoals blijkt uit wetgeving en internationale verdragen betreffende de export van cryptografische software.
8. Software is nooit af.
(Deze dissertatie, paragraaf 2.7)

9. De redenering dat de recente uitbarstingen van geweld op scholen en andere openbare gelegenheden in de Verenigde Staten gerelateerd zijn aan de populariteit van 'gewelddadige computerspellen' verklaart niet waarom dergelijke geweldsuitbarstingen buiten de Verenigde Staten nog niet voorgekomen zijn.

10. Er zijn slechts twee manieren om te voorkomen dat software illegaal gekopieerd wordt:

- Bepalen dat de betreffende software door iedereen legaal gekopieerd mag worden (freeware).
- Zorgen dat de software dusdanig laag van kwaliteit is dat niemand het wil kopiëren.