Unterschrift Betreuer

Diplomarbeit

Software modules for an electrochemical measurement system

Ausgeführt am
Institut für Chemische Technologien und Analytik
der
Technischen Universität Wien
unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günter Fafilek

durch

Markus Demetz
Reziastraße 269
39046 - St.Ulrich (Italien)

Wien, am 12 Februar 2007 ______________________

Abstract

This thesis deals with the creation of software components for the control and execution of measurement tasks in a complex system of elechtrochemical instruments. Many of these measurements require a coordinated cooperation between instruments and measurement devices of different vendors. This is a big challenge for all those who execute and observe the measurement tasks. Many of these instruments are shipped out with an additional software in order to facilitate complicated settings on the front panel of such an instrument. In most cases, such programs only support the functions of the instrument self. The work of this thesis presents software components which focus on the cooperation between the instruments. This is the point where many of the existing software begin to fail. The main aspect of these components is the dynamic embedding of new instruments without the need to change the existing system. The creation of the integration modules as well as the interpretation and execution of measurement tasks and the supporting frameworks are discussed in this work.

Kurzfassung

Diese Diplomarbeit befasst sich mit der Erstellung von Software Komponenten für die Steuerung und Durchführung von Messaufgaben in einem komplexen System von elektrochemischen Messgeräten. Viele solche Messaufgaben erfordern eine koordinierte Zusammenarbeit von Geräten verschiedenster Hersteller. Dies ist eine große Herausforderung für diejenigen, die die Messabläufe durchführen und überwachen. Die meisten Geräte werden mit einer Steuerungssoftware geliefert welche sich jedoch im großteil der Fälle auf die Funktionen des Gerätes beschränkt. Im Rahmen dieser Diplomarbeit werden Software Komponenten vorgestellt, die sich im wesentlichen auf die Zusammenarbeit von mehreren Messinstrumenten konzentriert welche mit herkömmlichen Programmen nahezu unrealisierbar ist. Das Hauptaugenmerk der vorgestellten Software Komponenten ist das dynamische Einbinden von neuen Geräten ohne das bestehende System zu verändern. Die Erstellung solcher Module, sowie die Interpretation und Ausführung von Messaufgaben und der Frameworks die dafür zur Verfügung stehen werden dieser Arbeit besprochen.

Contents

Introduction

This thesis is about the creation of software components which allow the controlling, performing and execution of electrochemical measurements in a system of electrochemical instruments. Due to the complexity of these measurements, there are many software programs around the world which are designed to interoperate and to help humans when using electrochemical devices, since the usage of the front panel of such an instrument often requires a detailed knowledge about the instrument itself. These programs often help users to by-pass complicated settings on the front panel of such instruments by doing many settings automatically. Not only the complexity itself, but also the iterations of measurements, changing some measurement parameters, require a constant presence of humans. A well known program is LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench), a platform and development environment for a visual programming language from National Instruments. It is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms [wik06c].

Motivation

The creation of this software is part of a project, formely called *MessTool*, which has been called into life from *Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günter Fafilek* and *Projektass. Dipl.-Ing. Bernhard Egger* in autumn 2004. I joined in May 2006 within the context of a Diploma Thesis of Informatics. The intention of this project was to create a system that allows to do measurements which are not possible at all or very unflexible with these popular software environments on the one side, and the possibility to expand the system when new requirements arise on the other side.

Goals

1. **Data collection:** One of the main goals is to collect measurement data for further analysis. Measurement data should be collected in a

way, that does not require constant presence of a human, doing parameter changes and starting new iterations or filling parameters into measurements on other instruments.

2. **Data exchange between instruments:** Measurement data collected while doing some measurement should not only be available as an output, but in the same moment, also as an input for other commands on other devices, perhaps on other computer stations. This is the point where many of the existing software begin to fail.

3. **Extension:** It should be possible to extend the system and accordingly the software when attaching a new measurement device without having expertise programming knowledge. One should be able to fill out a template by writing the neccessary base functions and/or base commands to generate a plugin for a new DLL module.

In order to satisfy the former specified requirements, I did some research work on the existing software programmed by *Schmidt David* and *Kalchgruber Martin*. The graphical user inteface has been programmed by Schmidt, and the *Hardwarecontroller* (which controls communiction between measurement instruments) by Kalchgruber. My experience in modelling and programming led me quickly to the conclusion that the existing software could not satisfy flexibility, maintainability and extensibility. So I decided to do some changes in the existing database design and start a new approach of the controller components behind the GUI. I spent a few weeks, thinking and modelling before doing implementation.

The first meetings with *Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günter Fafilek*, *Projektass. Dipl.-Ing. Bernhard Egger*, *Schmidt David* and *Kalchgruber Martin* took place at the end of April 2006. At the beginning, of course, I didn't know about the structure and the idea behind the project. The task given to me in the first days was to create a tool with which it would be possible to build or generate DLL modules wich would act as plugins when adding a new electrochemical instrument to the software system.

Problems

I already mentioned the challenge to perform inter-instrumental measurements, meaning the usage of devices from different vendors using different connections or busses. Most of the the electrochemical devices are shipped with software tailored to this product or product family, making it almost impossible to perform measurements by exchanging results or by using result of measurements as input parameters of others respectively.

Solutions

The already existing version called *MessTool*, programmed by *Schmidt D.* and *Kalchgruber M.* allowed only a subclass of the desired flexibility. The core of the entire system is a database based on MySQL. All data used by the system, like available functions, completed measurements and results are stored here. The database can be accessed from anywhere in the system from any client knowing the credentials.

I decided to review database. It has a tree like table structure which my changes didn't affect. The table fields underlied some changes. I wanted to create a unique application programming interface (short API) which both sides of the software system (client and controller) can use.

Acknowledgement

I would like to express my gratitude to *Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günter Fafilek* for discussions and the review of the manuscript.

I thank all those who gave me the possibility to complete this thesis.

Chapter 1

Hardware

1.1 The General Purpose Interface Bus

Great part of todays electronic instruments, such as meters and analyzers, make use of the advantages of cost-effective and powerful desktop and notebook computers. Together with software, we can speak about the birth of virtual instruments. Application software empowers the user with the tools necessary to build virtual instruments and expand their functionality by providing connectivity to the enormous capabilities of PCs, workstations, and their assortment of applications, boosting performance, flexibility, reusability and reconfigurability while diminishing at the same time development and maintenance costs [Zso].

1.1.1 History

The General Purpose Interface Bus (short GPIB, IEEE488) also called General Purpose Instrumentation Bus has its roots in the 1960's. In these days, Hewlett Packard (HP) was a manufacturer of test and measurement instruments, such as multimeters and analyzers.

To enable easier interconnection between instruments and controllers, such as computers, HP developed the HP - Interface Bus (HP-IB) which was relatively easy using the technology at this time. Now, the need for a standard interface for communication between instruments and controllers from various vendors arose. Other manufactors copied the HP-IB by giving it the name GPIB. It's high transfer rate of 1 Megabyte per second, quickly led the GPIB to gain popularity, so that the bus was standardized by the Institute of Electrical and Electronics Engineers as the IEEE Standard Digital Interface for Programmable Instrumentation. This bus is now used worldwide and is known by three names:

- General Purpose Interface Bus (GPIB)
- Hewlett-Packard Interface Bus (HP-IB)
- IEEE 488 Bus

The original document for the IEEE 488 Bus, contained no guidelines for a preferred syntax and format conventions. So work on the specification continued to enhance system compatibility and configurability. The result was a supplemented standard named IEEE488.2 which didn't replace the old standard IEEE488 which has been renamed IEEE488.1. The IEEE 488.2 specification provides a basic syntax and format conventions, as well as device independent commands, data structures and error codes. Many instruments still do not conform to the IEEE 488.2 standard.

To stop the increase of different vendor solutions, the IEEE 488.2 has been extended to the Standard Commands for Programmable Instrumentation (SCPI). Each class, also from differend vendors, must obey to this standard, in order to guarantee system compatibility and configurability among these instruments.

1.1.2 Design

The IEEE 488 Bus is an 8 bit communication system between 2 or more electronic devices. 30 devices could be addressed, but the bus only allows 15 to be connected at the same time. Address limits can be circumvented directly by the use of bus expanders or indirectly through the use of an isolator or an extender. The device addresses are set with mechanic switches on the devices or digitally on more sophisticated ones. For some devices (i.e. 1286 electrochemical interface), two GPIB ports are provided. One is used for ASCII commands and data, and the other for high speed binary (DUMP) output [Ins88]. The address for the ASCII input/output is called the MAJOR address and is always an even number. The address immediately following a MAJOR address is called the MINOR address and is, of course, an odd number.

When a computer is used as a device on the bus, it acts as a controller and is supervising the communication exchange between the other devices. Figure 1.1 shows a female IEEE488 Bus connector, it uses 24 pin most commonly in a stackable male/female combination that allows a daisy chain of

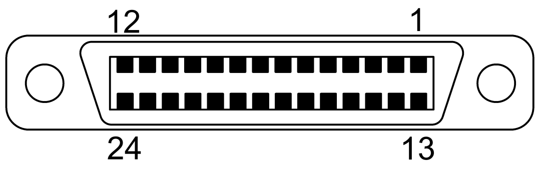


Figure 1.1: A female IEEE488 connector (origin: [wik06b])

devices. Total cable length, the sum of chained connectors, is limited to 20 meters. The stacked connectors should be less or equal to 4.

Only one device can talk at a time, and it is called the *active talker*. On the other side, all devices which are listening are referred to as *active listeners*. In order to optimize bus speed, the controller instructs all other devices to unlisten [Kei03].

1.1.3 Interface signals

The IEEE-488 interface system consists of 16 signal lines and 8 ground lines. The 16 signal lines are divided into 3 groups (8 data lines, 3 handshake lines, and 5 interface management lines) [Sof05].

Pin	Function	Group
1-4	DIO 1-4 (Data input-output)	Data
5	EOI (End or identify)	Controller
6	DAV (DAta Valid)	Transfer
7	NRFD (Not Ready For Data)	Transfer
8	NDAC (No Data ACcepted)	Transfer
9	IFC (InterFace Clear)	Controller
10	SRQ (Service ReQuest)	Controller
11	ATN (ATteNtion)	Controller
12	SHIELD	Common
13-16	DIO 5-8	Data
17	REN (Remote enable)	Controller
18-24	GND (GrouND)	Common

The bus operates at the speed of the slowest device and all devices have to be in ready state before some operation begins.

Data lines

The data lines (DIO1 trough DIO8) are used to transfer addresses, control information and data. The talker is sending data on the data lines, and the listener is receiving them. DIO1 is the least significant bit, and the bus uses negative logic with standard TTL (Transistor-Transistor-Logic).

Handshake lines

The handshake lines guarantee an error free transmission of the data on the data lines. They operate in an interlocked sequence and ensure a reliable data transmission regardless of the transferrate, since the transferrate is the rate of the slowest device on the bus.

DAV (DAta Valid)
The source controls the state of the DAV - line. If this line is set to low, it indicates to all listening devices that the data lines contain valid data.

NRFD (Not Ready For Data)
The listeners on the bus indicate with this signal, that they have not handled yet the data on lines DIO1-DIO8. The line is driven by all devices when receiving commands, and by Listeners when receiving data messages.

NDAC (Not Data ACcepted)
This line indicates wheter the data on the data lines has been accepted or not. The line is driven by all devices when receiving commands, and by listeners when receiving data messages.

Figure 1.2 shows the complete handshake sequence for one data byte.

The *handshaking* takes place as follows:

1. The talker or controller that wants to push data on the bus sets the DAV line to high, indicating that the data is not valid. It must ensure, that the NRFD and NDAC lines are both low, than it can put the data on the data lines.

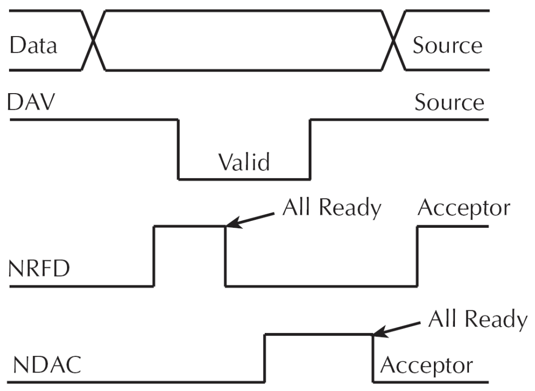


Figure 1.2: The handshake sequence for one data byte (origin: [Kei03] figure G-2)

2. When all listeners are ready for receiving data, each releases the NRFD (not ready for data) signal. When the last listener releases the NRFD, the signal goes high and the talker takes DAV to low. Now the data lines are set to the valid state.

3. In response, each listener takes NRFD low again indicating its busy status while reading from the data lines. When finished, it releases NDAC (not data accepted). The NDAC remains low until the las listener has accepted the data. Now the controller or talker can set DAV high again to transmit the next byte of data.

Bus management lines

There are five lines, which control the bus activities.

ATN (ATteNtion): This signal indicates that a command byte is present on the data lines (e.g. an address)

IFC (InterFace Clear): The systemcontroller can reset the bus and become the active controller.

REN (REmote Enable): If activated, all bus members go into remote modus, and into local modus when released.

EOI (End or Identify): Becomes active with the last data byte, indicating the end of a message.

SRQ (Service ReQuest): Whenever a device is requesting a service, this line is asserted. The controller then polls the devices to find the requesting one and performs whatever action is neccessary (equivalent of a hardware interrupt [Ryn05]).

Ground lines

The eight remaining lines are ground-return or shield-drain lines.

1.1.4 The High-Speed GPIB protocol

In 2003, National Instruments, has developed the high-speed GPIB handshake protocol. Normal transfer rates on the GPIB bus are on the order of 1Mbyte/s. The motivation to increase the transfer rate of a GPIB system led to the developement of the patented high-speed GPIB handshake protocol called HS488 [Ins03].

All devices in the GPIB system must be compliant to use the HS488 protocol. If non-HS488 devices are involved, the HS488 devices automatically use the standard IEEE 488.1 handshake to ensure compatibility.

1.1.5 A GPIB replacement

Many customers, especially those sensitive to price, are saying that the 25-plus year old IEEE std. 488.1 (GPIB) needs to be replaced [Pur99].

There where many discussions about a replacement of the GPIB through one out of USB, IEEE 1394 (also known as Firewire) or network I/O. The reason is, that customers prefer I/O that is built into the computer.

The good of GPIB

- IEEE 488 is a widely used interface which has reached a high level of maturity. It has been around 25+ years and is a widely trusted and reliable communication bus.

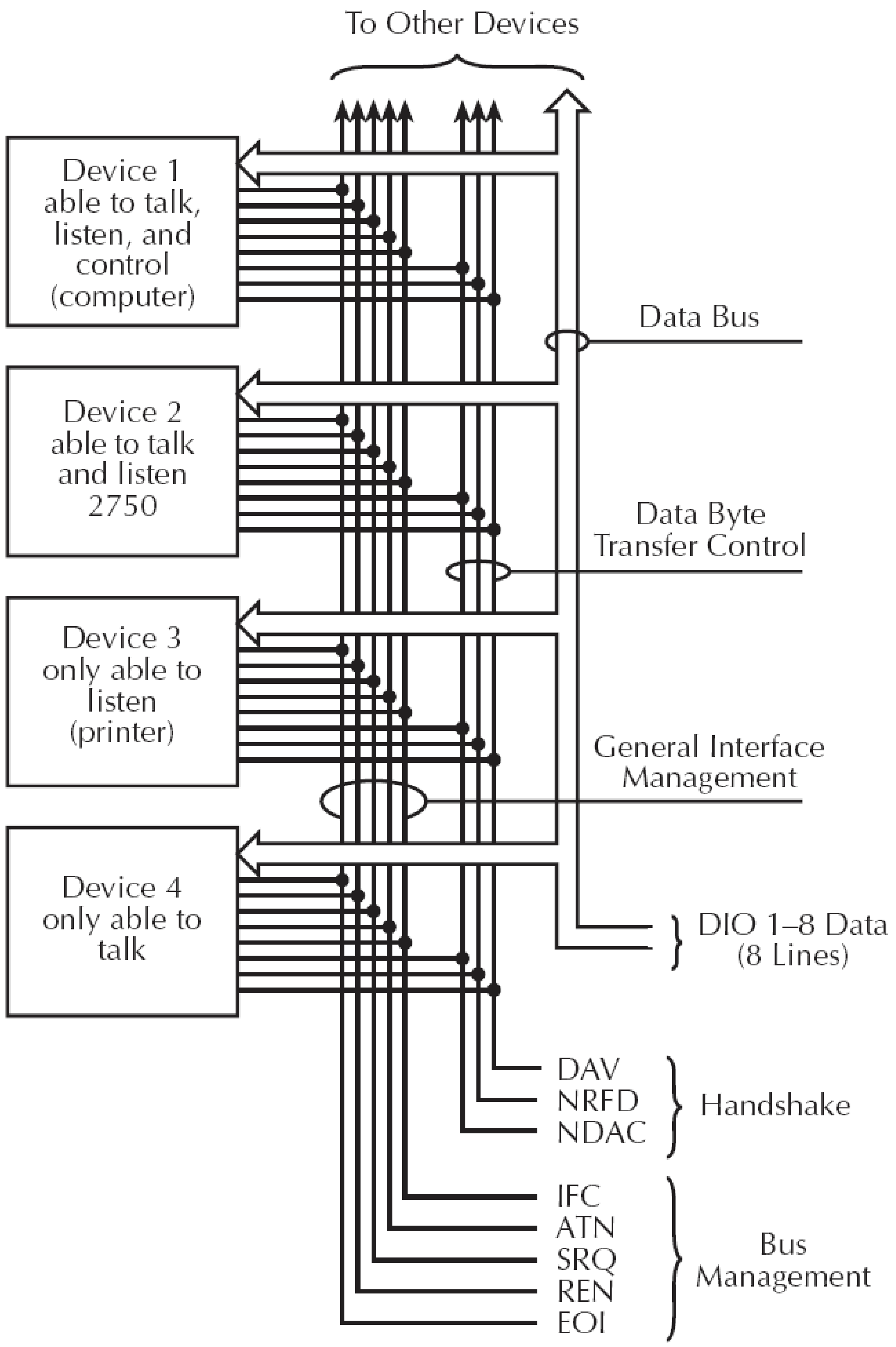

To Other Devices
Device 1 able to talk, listen, and control (computer)
Device 2 able to talk and listen 2750
Device 3 only able to listen (printer)
Device 4 only able to talk
Data Bus
Data Byte Transfer Control
General Interface Management
DIO 1–8 Data (8 Lines)
DAV
NRFD
NDAC
Handshake
IFC
ATN
SRQ
REN
EOI
Bus Management

Figure 1.3: IEEE488 bus configuration (origin: [Kei03] figure G-1)

- It has lines dedicated for specific functions: SRQ (equivalent of a hardware interrupt), local lockout, etc.
- A lot of software has already been written for the countless number of GPIB devices around.

The bad of GPIB

- Maximum cable length is 2 meters.
- Maximum of 15 instruments on the bus at one time (including the controller).
- Bus speed (theoretical max 1.5MB/s, this doesn't take into consideration HS488).
- Cables are expensive.
- Need a GPIB controller card.
- The number of slots in PCs is being reduced each year.
- Costs around $500.
- Installation hassle with drivers.

1.2 Electrochemical Measurements / Devices

The idea behind the project was to perform local corrosion measurements with an electrode array. Corrosion is bounded locally, so it would be a great help to know the exact parameters leading to corrosing process [Kal06].

Corrosion tests can take several days even weeks, so an automatic way would be great. To perform an efficient measurement the need of at least two devices arises. A Multimeter/Switch device is needed to select the actual electrodes while the second one, that could be a frequence analyzer or a voltmeter which does the actual measurement. This requires interoperability between at least two interfaces.

For testing the operability of the software, following devices have been used:

- Solartron 1286
- Keithley 2750 Multimeter / Switch
- HP4192A Impedance Analyzer

Chapter 2

Software

2.1 Definition

Software is a program that enables a computer to perform a specific task, as opposed to the physical components of the system (hardware). This includes application software such as a word processor, which enables a user to perform a task, and system software such as an operating system, which enables other software to run properly, by interfacing with hardware and with other software [wik06e].

2.2 Software libraries

A program usually requires additional software from a *software library* in order to be complete for execution. Libraries are not independent units, they are helping modules which can be used by other programs. All of the larger programs are constituted in a modular way by subcomposing the entire application into libraries. They contain software routines, which gives the possibility to share code and data over different modules or over the entire system.

Software libraries can be categorized by three ways:

Source libraries contain collections of value definitions, declarations, functions, classes, etc.

Static libraries become linked after the compiling process to the program. For execution, one file is enough, which could become very large.

Dynamic libraries can avoid the programs to become very large by linking it to dynamic libraries instead of static libraries. Dynamic libraries are accessed during runtime when a funtion or some data from this library is needed.

More about *linking* in a next chapter.

2.3 Programming language

For the project this thesis is about, my colleagues have chosen C++ as implementation language. The reason for this choice is simple. C++ is a general-purpose, high-level programming language with low-level facilities [wik06a]. Many devices are shipped out with the relating libraries to communicate with. Since C++ is one of the most common user programming languages in the industrial section, all of the libraries shipped with are at least written in C. Though there are some incompatibilities between C and C++, Bjarne Stroustrup, the creator of C++, has suggested that the incompatibilities should be reduced as much as possible in order to maximize inter-operability between the two languages [Str02].

So, with a few exceptions, C++ is downward compatible to a well written (ANSI) C code and should pass every C++ compiler.

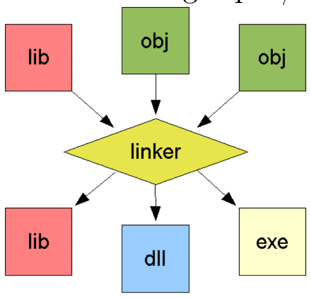
2.4 Linking

Linking is known as the process done by a program, usually called *linker* (also binder or loader), that takes the object generated by a compiler and assembles them together into an executable single program that can be loaded into memory. The objects generated by the compiler usually contain machine code which can be executed. But that's not all. The objects also include other useful information like relocation information, program symbols and debugging information.

I would like to mention the differences between static and dynamic linking and also examine why dynamic linking is the better choice.

Before explaining the difference between static linking and dynamic linking, we have to explain the terms *interface* and *implementation*.

Figure 2.1: Linking input/output



2.4.1 Interfaces versus Implemenation

Interfaces are the expression of "what" something does. Interfaces are designed to remain stable over many releases of the product.

Consider the controls used to drive a car. The driver's interface remains relatively constant. The basic controls for stop, go, and steer stay in the same place. You don't need to know how many cylinders your engine has before you can drive your car. Likewise, a computer's interface must maintain some consistency at many levels to prevent leaving users and administrators in the dark [Coc96a].

Implemenation hides behind the interface and does the actual work. So it differs from *interfaces* since implementation is the expression of "how" something is done and does the actual work that the "what" expression promises.

Implementation changes from one release to the next, ususally without any affection to the interface. It can contain bug fixes, performance enhancements, security issues, etc. Also hardware differencies are handled by changes in the implementation.

If a car engine starts to misfire and you need to lift the hood and tinker with the ignition timing, you suddenly face a lot of implementation details. Outwardly identical cars may harbor major differences under the hood (such as completely different engines). Furthermore, many components change year by year as well [Coc96a].

Linking can be done at compile time, at load time or at run time.
When executing a program, the executable object file is loaded into the main

memory and puts the program in a *ready-to-run* state. It might involve storage space allocation and virtual address mapping to disk spaces [Gro02]. For each input module, the compiler generates an object which has a specified format. Known formats are the ELF (executable and linking format) for executables on the x86 architecture (Linux) and the PE-COFF (Portable Executable and Common Object File Format) on a Microsoft environment. The name "Portable Executable" refers to the fact that the format is not architecture-specific [Mic].

Now each object module has a starting address of zero and a next step called *relocation* must be done. All symbolic references or names of libraries are beeing replaced with the actual usable address in memory. The *zero* addresses must be adjusted so they point to the correct runtime addresses. A program is made up of different subprograms which are referenced through symbols. The linker resolves the references and re-targets the absolute jumps and patches the callers object code.

2.4.2 Static Linking

When linking statically, the compiler combines an application by joining it with all the parts of various library routines it uses. Static linking is the original method and occures usually while development. In this case, the output is a unique executable file.

Libraries in Microsoft Windows have the ending *.lib* (library) while in Linux and Unix systems they have the ending *.a* (archive).

2.4.3 Dynamic Linking

When linking dynamically, the symbol resolution gets deferred until the program is executed. This makes use of a dynamic or shared library with the ending *.so* (shared objects) in Linux and Unix systems and *.dll* in the Windows environment (dynamic link library). A shared or dynamic link library can be loaded into an arbitrary memory address during runtime. The executable simply contains some relocation and symbol table information that allow references to code and data in the dynamic link library to be resolved at run time. Such a library can be loaded from an application at an arbitrary time of execution.

When loading a dll in a program, the exported dll-functions are called with 2 different linkage methods. In this case we speak of *Load-Time Dy-*

namic Linking (dynamic linking at program execution) and *Run-Time Dynamic Linking* (dynamic linking at runtime).

Load-Time Dynamic Linking

When using *Load-Time Linking*, the application makes calls to exported dll-functions which in this case act like local functions. When using this approach, we need a header file (.h) and an import library (.lib) when compiling the application. Thus, the linker gives information to the system to load the DLL and to resolve the DLL-function at execution time.

Run-Time Dynamic Linking

When using *Run-Time Linking*, the application calls LoadLibrary or LoadLibraryEx to load the DLL file at runtime. After a successful loading of the DLL, the GetProcAddress is used to resolve the symbolic name of the exported function to get it's address which in turn is used to call it. For this approach, no import library is needed.

The following criteria should ease the decision of choice between the 2 linking methods above.

- Startup power
  When the startup process should be very fast, so one should use *Run-Time Dynamic Linking*

- Simplicity
  When using *Load-Time Dynamic Linking*, the exported functions of the DLL behave like local function. This simplifies their calling.

- Application logic
  Run-Time Dynamic Linking allows the possibility to load different modules as the need arises. I.e. when developing multilingual versions [Mic06].

2.4.4 Reasons why Dynamic Linking is superior

Many people think that static linking has benefits. This has never been the case and will never be the case [Dre].

- Physical memory is used more efficiently by sharing the library over all processes using its code. Consider standard functions like printf(...) or scanf(...) which are used by almost every application. Now, if a system is running 50-100 processes, each process has its own copy of executable code for printf and scanf.

- A security fix or a bug fix is achieved by simply replacing the shared object implementing this issue. When linking statically at compile time, one cannot remind easily which library has been linked a program to. In this case, every program built with the critical library has to be relinked. This alone (with the next) is considered to be the killer arguments [Dre].
- Address space layout randomization cannot be used with position independent executable (PIE). Fixed addresses (or even only fixed offsets) are the dreams of attackers, and when linking statically, all text has a fixed address in all invocations.

In this project, dynamic binding becomes of relevant importance.

2.5 MySQL

2.5.1 MySQL Database Server

The reason of selecting MySQL as the database management system (DBMS) is that the MySQL Server is available at the GNU General Public License (GPL). The MySQL DBMS is owned by the swedish company *MySQL AB* which also offers a commercial licence for cases where the intended use is incompatible with the GPL [wik06d].

An enormous popularity is found in the area of webservers, since normally it can be used as a free licence which is very attractive for providers.

2.5.2 The values of MySQL AB

MySQL AB wants the MySQL server to be:

- The best and the most used database in the world
- Available and affordable for all
- Easy to use
- Continuously improved while remaining fast and safe

- Fun to use and improve
- Free from bugs

MySQL AB and the people of MySQL AB:

- Subscribe to the Open Source philosophy
- Aim to be good citizens
- Prefer partners that share our values and mindset
- Answer emails and give support
- Are a virtual company, networking with others

2.5.3 MySQL++ API

MySQL++ is a C++ wrapper for MySQL's C API. It is built around STL[1] principles, to make dealing with the database as easy as dealing with an STL container. MySQL++ relieves the programmer of dealing with cumbersome C data structures, generation of repetitive SQL statements, and manual creation of C++ data structures to mirror the database schema.

[1]Standard Template Library (STL) is a software library included in the C++ Standard Library. It provides containers, iterators, and algorithms.

Chapter 3

Implementation

The existing software was strictly divided into a *graphical user interface* programmed by Schmidt, and the *HardwareController* programmed by Kalchgruber. The database has been destined to be the only "interface" to the components are exchanging data. The Database has been designed by both programmers (Kalchgruber and Schmidt) and shows a tree-like or even directory-like structure. The specification of the database was also a little rudimentary. After spending a few thoughts about the database, i agreed with the tree-like structure and took it as a basis for a brand new start of the formerly called *HardwareController*.

I saw that the database access was implemented twice, one time in the GUI, the other time in the *HardwareController*. Everyone did it in it's own way, so I saw a potential redundancy in code which led me quickly to the conclusion to implement a database access API which would be called *Task++*. The inspiration of the name was given by the fact that the application should deal with measurements that in turn are made up of tasks. The *++* because of the using of C++ as implementation language and *mysql++* as the mysql database wrapper. My idea was to spend work into programming the base-classes which map the database tables into objects.

The developed software components can roughly be devided into *Task++*, *TaskDLL* and *TaskMachine*. The idea behind this separation was to create reusable modules which are independent from each other to enhance maintainability and reusability.

3.1 Database

3.1.1 Design

The database is the only way the components in the measurement system interact. On one side, the GUI clients access the database to gather information about the available objects in the system which become registered from the TaskMachine described in section 3.4. The other side make(s) up the TaskMachine(s) again which take an event from the GUI client whenever some data in the database is present to start a measurement.

The database design is made up of a distinction between the tables that represent the availabe objects and the ones which store the measurements done. The tables shown in figure 3.1 are placeholders for the *available* objects in the measurement system and have the prefix A. The tables shown in figure 3.2 store the actual persistent data. Once a measurement has been started, these tables contain information about the measurement structure. The prefix used is H which should stand for history. The tables *Measurement* and *Result* are unique in database and do not need a prefix for distinction.

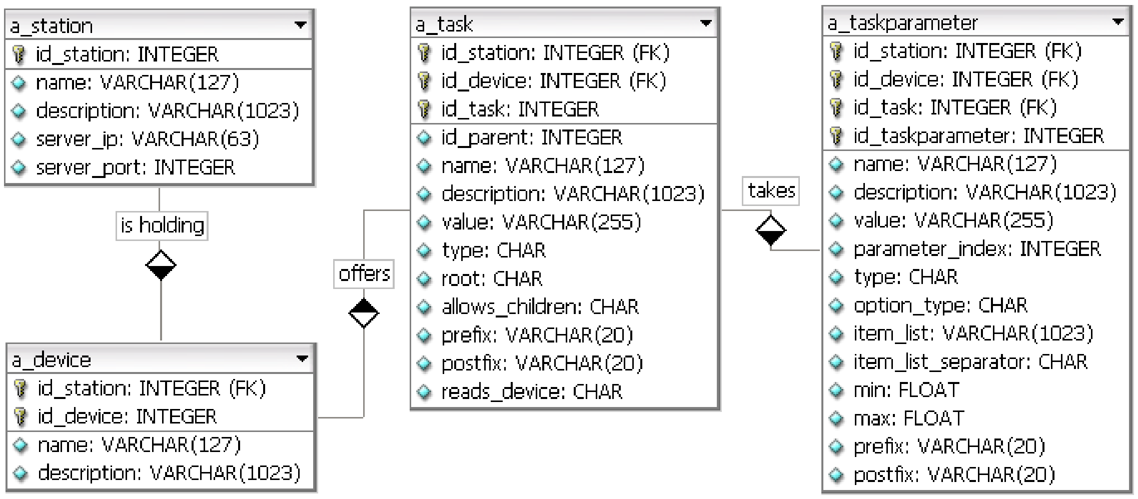


Figure 3.1: The data tables for available objects

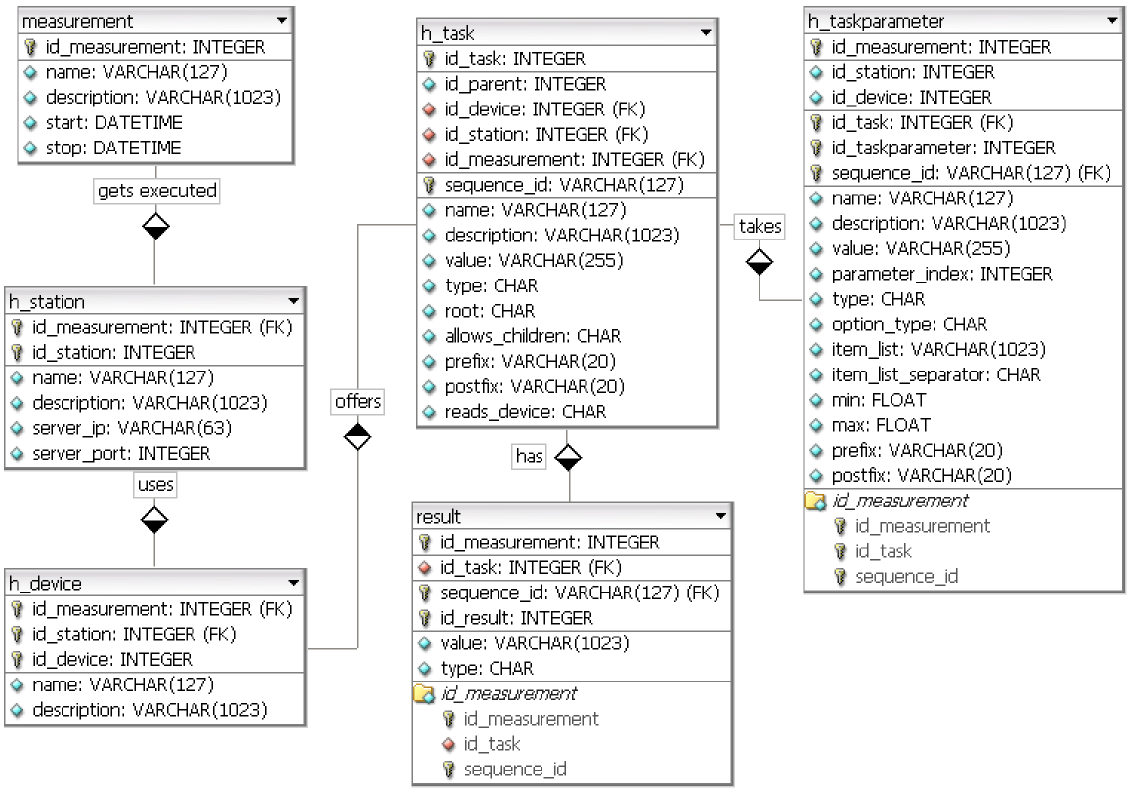


Figure 3.2: The data tables for measurement objects

3.1.2 Description

A_STATION

field	type	option	note
id_station	INTEGER	PK	A computer station with a unique number in the network.
name	VARCHAR(63)	NN	The name of the station.
description	VARCHAR(1023)		A description of the station.
server_ip	VARCHAR(63)	NN	The IP in the network.
server_port	INTEGER	NN	The port the station is listening onto.

A_DEVICE

field	type	option	note
id_station	INTEGER	PK	The id of the station this device is attached onto.
id_device	INTEGER	PK	The number of the device which is unique on this station.
name	VARCHAR(63)		The name of the device appearing in the GUI
description	VARCHAR(1023)		A description of the device.

A_TASK

field	**type**	**option**	**note**
id_station	INTEGER	PK	The id of the station which the device is attached onto this task belongs to.
id_device	INTEGER	PK	The id of the device which this task belongs to.
id_task	INTEGER	PK	The id of the task which must be unique on this device.
id_parent	INTEGER	NN	The id of the parent task which this task belongs to if it is not a root task. For root tasks, the id_parent is 0.
name	VARCHAR(63)	NN	The name of the task appearing in the GUI.
description	VARCHAR(1023)		A description for this task. What does this task?
value	VARCHAR(255)	NN	The value is the token which is interpreted by the device or the interpreter in the TaskMachine.
type	CHAR	NN	A character flag indicating if this task is (p)hysical, (v)irtual or (l)ogical.
prefix	VARCHAR(20)		The prefix added to the task when rendering the command.
postfix	VARCHAR(20)		The postfix added to the task when rendering the command.
reads_device	CHAR		A boolean flag indicating if the device provides a result. This flag is only set to *physical* tasks.

A_TASKPARAMETER

field	type	option	note
id_station	INTEGER	PK	The id of the station...
id_device	INTEGER	PK	The id of the device...
id_task	INTEGER	PK	The id of the task...
id_taskparameter	PK INTEGER		
name	VARCHAR(127)	NN	The name of the taskparameter shown in the GUI.
description	VARCHAR(1023)		A description of the taskparameter.
value	VARCHAR(255)		The actual value of the taskparameter.
parameter_index	INTEGER		The position of the taskparameter.
type	CHAR		A character flag indicating the type of the taskparameter. (s)tring, (i)nteger, (d)ouble, (b)ool, (v)ariable or (r)eference.
option_type	CHAR		A character flag indicating if this taskparameter is (o)ptional, (u)nique, (r)equired. The combination of unique and required is called (s)elect.
item_list	VARCHAR(1023)		An optional list of selectable items separated by the following item_list_separator.
item_list_separator	CHAR		A single character separator for the itemlist.
min	FLOAT		An optional lower bound for a numeric taskparameter.
max	FLOAT		An optional upper bound for a numeric taskparameter.
prefix	VARCHAR(20)		A prefix added to the parameter when rendering.
postfix	VARCHAR(20)		A postfix added to the parameter when rendering.

MEASUREMENT

field	type	option	note
id_measurement	INTEGER	PK, AUTO	The id of the measurement.
name	VARCHAR(127)	NN	The name of the measurement.
description	VARCHAR(1023)		A description of the measurement.
start	DATETIME		The start time of the measurement.
stop	DATETIME		The stop time of the measurement.

In the following tables, H_STATION, H_DEVICE, H_TASK and H_TASK-PARAMETER only the added fields are listed.

H_STATION

field	type	option	note
id_measurement	INTEGER	PK	This station has been used for the measurement with id_measurement.

H_DEVICE

field	type	option	note
id_measurement	INTEGER	PK	This device has been used for the measurement with id_measurement.

H_TASK

field	type	option	note
id_measurement	INTEGER	PK	This task has been used for the measurement with id_measurement.
sequence_id	VARCHAR(127)	PK	The sequence id of this task. (i.e. 1.3.2)

H_TASKPARAMETER

field	type	option	note
id_measurement	INTEGER	PK	This taskparameter belongs to a task of measurement with id_measurement.
sequence_id	VARCHAR(127)	PK	The sequence id of the task this taskparameter belongs to.

RESULT

field	type	option	note
id_measurement	INTEGER	PK	The id of the measurement.
id_task	INTEGER	FK	The id of the task.
sequence_id	PK	FK	The sequence id of the task.
id_result	INTEGER	PK	The id of the result.
value	VARCHAR(1023)		The value of the result.
type	CHAR		A character flag, (s)tring, (i)nteger, (d)ouble or (b)ool.

3.2 Task++ classes

Task++ is the collection of classes which, amongst others, implement the database access functionality. These classes are packed into a dynamic library which can be used by every software component needing to deal with database access. Mainly, this library is used from the *TaskMachine*, which is described later, and the graphical user interface (GUI).
The classes reflect the database tables, so a database access is provided in an object oriented way.

The creation of this library features the *Load-Time Dynamic Linking* since the DLL is bound at the loading time of the program. The program using the library is compiled using the appropriate header files and the belonging .lib file which contains linkage information. The dynamic link library *taskpp.dll* is copied into the programs execution folder.

For each database table, the library contains a class of a simliar name. Each class provides getter and setter methods[1] for it's attributes as well as methods for writing, reading, deleting and updating the database with the

[1]By convention, setAttribute and getAttribute methods are abbreviated by setter and getter methods

data encapsulated into these objects.

3.2.1 Measurement classes

Station

A station is a uniquely identified computer in the network which can have measurement interfaces attached onto. A std::vector of devices gives information about the devices attached onto. A station object is contructed the following way:

```
Station station = new Station(name, description, ip, port);
```

Device

Once a station object has been created, it is possible to create a device which will be automatically attached to the station. To avoid inconsistency, the default constructor [2] is declared as protected and cannot be accessed from outside [3].
The only available constructor takes 3 arguments shown in the listing below. Therefore it is not possible to create a device hanging around alone in the system.

```
// Create Device
Device device = new Device(station, name, description);
% device->setIdDevice(id_device);
```

This technique of object construction is also used in the other classes, so relation consistency is guaranteed and memory clean up can be simply done by calling the destroy() method of the root object. Due to the tree-like or directory-like structure of the database and of the objects respectively, when destroying an arbitrary object, all subjects are destroyed too, and memory clean up is done by the library itself.

Devices, in turn, have a std::vector containig *Task* elements.

[2]The constructor which takes no arguments, for example Device()

[3]outside is meaning from a class which does not stand into an inherit or a friend relationship to the implementing class

Task

A task is an abstract representation of a function or a sequence of functions (which could also be an entire program). This is the part where most of the designing time elapsed.
A Task can represent many different things, therefore 3 types of tasks are distinguished:

- physical task
- virtual task
- logical task

A ***physical task*** is the low level representation. It's value is the actual ASCII command string which will be sent to the corresponding measurement instrument. In most cases the entire ASCII command is not formed from a single task element but together with taskparameters or other tasks. So there is another distinction between *root tasks* and *sub-tasks*. There are command protocol specifications which show a tree-like command set, therefore many tasks have the same prefix task which describes a taskgroup or whatever else.

Example commands are:

```
STATUS:OPERATION:ENABLE 2
STATUS:OPERATION:ENABLE?
STATUS:PRESET
STATUS:QUEUE:ENABLE
STATUS:QUEUE:ENABLE?
```

Without explaining the semantics of these commands, we can describe this command set as a tree structure with a root element containing the value *STATUS* having the sub-tasks *OPERATION*, *PRESET* and *QUEUE*. Apart from the *PRESET* task, the nodes have children too. The colons between the commands can be seen as part of the root command or as a prefix of it's children. If also STATUS? would be a valid command, it would not be a good idea to make the colon part of the root command since a second root node labeled STATUS? would be needed. Figure 3.3 shows the tree representation of the command set listened above.

When we load the available tasks from the database we have the possibility to load just a single task or the entire subtree recursively by setting the appropriate flag. Now a single path in the command tree is selected to form the command string before sending it to the device (figure 3.4).

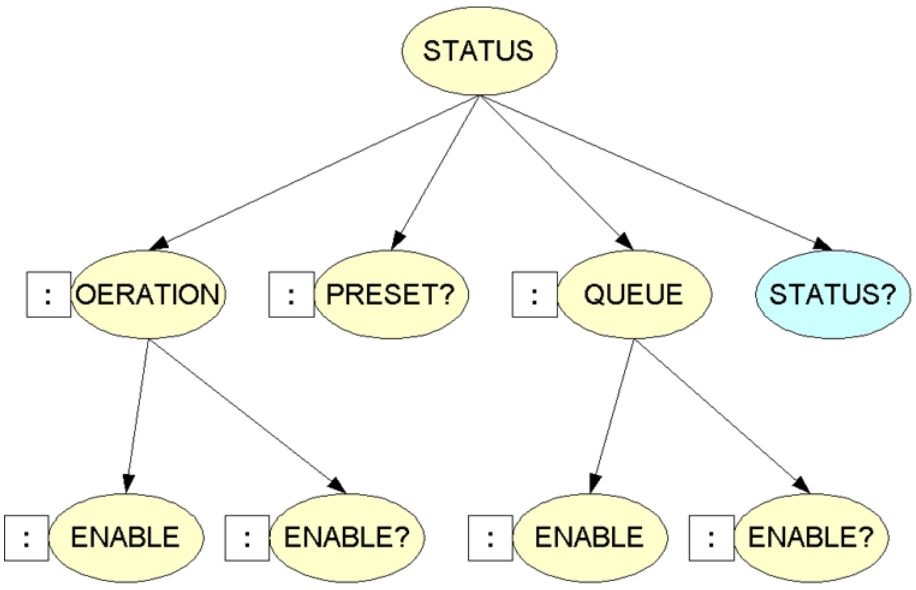


Figure 3.3: The command set as a tree representation

Rendering the command
After a tree has been selected the entire command string can be obtained by calling the task's render(SymbolTable∗) method. Either a pointer to a SymbolTable or null can be passed. If a SymbolTable∗ is passed, the symbols in the task parameter (see 3.2.1) get substituted by the associated values.

Listing 3.1: The render function of a task

```
std::string Task::render(SymbolTable *st) {
  std::string s = "";
  s += this->prefix;
  s += this->value;
  if (!this->subtaskList.empty()) {
    s += subtaskList.at(0)->render(st);
  }
  if (!this->parameterList.empty()) {
    for (std::vector<TaskParameter*>::iterator
        i = parameterList.begin(); i != parameterList.end();
         i++) {
      s += (*i)->render(st);
    }
  }

  s += this->postfix;
  return s;
}
```

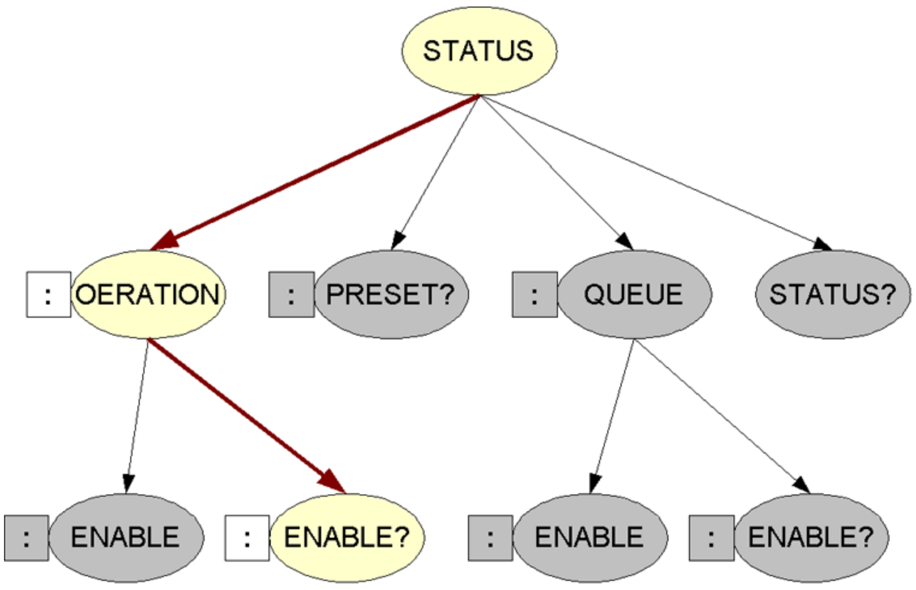


Figure 3.4: A selected path in the command tree

The render - function takes a symbol table as an argument, which will be passed to the render - function of the TaskParameter class which will be described below. We start with an empty string and add the prefix variable which can be empty or not. After that, the actual value (or the value of this node) is added to the string before adding the value recursively of it's child. Since we have chosen a unique path in our command tree, every task has at most one child node.

After that we have an iteration over a list of elements of type *TaskParameter*. This sounds a little confusing at the first moment. The fact is that an invariant[4] prohibids a task having subtasks and taskparameters at the same time. So if the subtask is rendered, we can be sure that the parameterlist is empty, since only a leaf node can have taskparameters.

A ***virtual task*** describes a set of tasks or a sequence of phsysical tasks between programming statements implemented by a device-module[5] developer. In chapter 4 we show the development of a new DLL-module.

[4]A constraint that ensures a consistent state of an object

[5]A device-module is considered as the DLL implementation belonging to a measurement device.

Virtual tasks represent already implemented functionalities. It would also be possible to achieve the same functionality by merging physical and logical tasks which increases complexity notably. Not every user has the sufficient knowledge about every single command accepted by a measurement device, and so it should be possible for him to use the virtual tasks doing the work.

A ***logical task*** belongs to a predefined set of tasks which control a logical sequence of the other 2 task types described above. This type of task becomes interpreted by the later described *TaskHandler* and becomes then mapped to a logical function which controls the sequence.

In logical task we use the symbol $ to identify a variable and the symbol # to identifiy a reference. A reference means a link to an already present result in the database. This will be described later when talking about *Results*.

1. For loops

   A for loop is identified by the value *FOR* and takes 4 parameters:

   ```
   FOR(FROM, TO, STEP, VAR_NAME)
   ```

   The *VAR_NAME* parameter is the symbolic name used for the loop counter. If we use $i as the variable name, we can use this value in any task belonging to the scope of the loop.

2. If statements

   An *IF* statement takes 3 arguments

   ```
   IF(VAL_1, OP, VAL_2)
   ```

   where *VAL_1* and *VAL_2* are number representations, variables or references to number representation. The *OP* parameter is the comparing operator out of the set $\{==, !=, <=, <, >, >=\}$

3. Else statements

   An else statement simply describes the branch that is executed when the evaluation of the last if statement returned a negative value.

TaskParameter

Each leaf task can have a list of taskparameters. Consider a command string formed up of STATUS:ENABLE <NRf> where <NRf> is a number representation format and takes an integer argument between an upper and a lower bound. Then, <NRf> can be a simple number, a variable representing a number or even a reference to a result in a number representation format.

We get the a list of parameters by calling the task's getParameterList() function which returns a copy of a vector containing pointers to *TaskParameter* classes. See the API for more details.

Like the *Task* class, it has a prefix and a postfix variable which is used when rendering the command. Consider the following ASCII command:

$$ROUTE : CLOSE < clist >$$

where $< clist >$ defines a list of channels in the form $< clist >= (@SCH)$ where S = mainframe slot number and CH = switching module channel number.

So we would have 2 task nodes, one for *ROUTE* and the other for *[:]CLOSE* and in turn a taskparameter which describes the list for the channels to close[6]. We don't want to ask too much of so to the user by asking him to enter the right syntax for the task parameter. He just has to enter a comma separated list of *SCH's* in the form 101, 201, 202 and the belonging syntax (prefix and postfix) is added automatically. Listing 3.2 shows the function that does the work.

Listing 3.2: The render function of a taskparameter

```
std::string TaskParameter::render(SymbolTable *st) {
  using namespace std;

  string s;
  string val;

  // add prefix
  s += this->prefix;

  // if SymbolTable is not NULL
  if (st != NULL) {
    s += this->parseValue(st);
  }
  // if SymbolTable is NULL, render normally!
  else {
    s += this->getValue();
  }
```

[6]ROUTE:CLOSE belongs to the command set of the Keithley Multimeter/Switch and closes the channels

```
  // add postfix
  s += this->postfix;
  return s;
}
```

The function is called from the task's render function. The *symboltable* can either be NULL or a *symboltable* containing symbol to value mappings which in turn may be used as variables in the taskparameter. If the SymbolTable is NULL, the value is rendered without symbol substitution, otherwise the value is parsed using the parseValue(SymbolTable*) function shown in the following listing:

Listing 3.3: The function for parsing the taskparameter's value

```
std::string TaskParameter::parseValue(SymbolTable *st) {
  using namespace std;
  string retval;

  string::size_type loc_ref = value.find("#");
  string::size_type loc_var = value.find("$");
  string::size_type loc_ind = value.find("[");

  // Search for a Reference
  if (loc_ref != string::npos) {
    string var;
    string index;
    string ref;
    // Search for an Index
    if(loc_ind != string::npos) {

      // this parameter is an indexed reference
      if (loc_var != string::npos) {

        // this parameter is indexed with a variable
        while (loc_var < value.size()) {
          char c = value.at(loc_var);
          if (c == ']') break;
          if (c != ' ' && c != '[') var += c;
          loc_var++;
        }
      }
      while(loc_ind < value.size()) {
        char c = value.at(loc_ind);
        if (c == ']') break;
        if (c != ' ' && c != '[') index += c;
        loc_ind ++;
      }
    }

    while (loc_ref < value.size()) {
      char c = value.at(loc_ref);
      if (c == '$' || c == '[') break;
      if (c != ' ' && c != '#') ref += c;
      loc_ref ++;
    }

    int i = 1;
    if (!index.empty()) {
      if (index.find("$") != string::npos) {
        i = atoi(st->get(index).c_str());
      }
      else {
        i = atoi(index.c_str());
      }
    }

    Result *res = Result::loadFromDb(st->getIdMeasurement(),
              ref, i);
    if (res) {
      retval = res->getValue();
    }
    res->destroy();
  }
  // Search for a Variable
  else if (loc_var != string::npos) {
    // this parameter is a variable
    retval = st->get(value);
  }
  else {
    // this parameter is a normal value, no parsing.
    retval = value;
  }
  return retval;
}
```

A taskparameter has three special character symbols to influence the rendering output. These special symbols are:

Dollar sign \$: marks a variable, i.e. \$i in a for loop.

Rhomb #: reference to a result in database, i.e. #1.2.1

Angular brackets [ ]: for indexing a reference, i.e. #1.2.1[2] or #1.2.1[\$i] (indexed).

The above code listing first uses the find method of the value string which returns a pointer unequal to `string::n_pos` if the character given as argument (in our case one out of $, #, [) was found. If a reference sign (#) was found, we also check if there is also an index present. If it is, we check if the reference is indexed with a variable or a constant and store them into the index string which is used to get the value from the symboltable if it contains the variable name, or simply parsed into an integer[7]. All gained information is used to load the referenced data from the database by creating a result object and returning its value.

In the other case, the parameter value only contains a variable which is used to get the actual value from the symboltable. If neither a reference, nor a variable sign has been found, the value becomes returned as it is and no processing is needed.

Measurement

Once the system has loaded successfully, we can start creating a measurement. A measurement is identified by a unique id in the system, has a name and an optional description. A start and a stop timestamp should give information about the execution duration.

```
Measurement m("Hello measurement!", "Hello description!");
```

Between others, the class provides methods like execute() and stopExecute() for starting and aborting a measurement. We'll see the usage of them later.

HStation

The classes with an *H* prefix are derivated classes and describe the actual usage in a measurement, while the base[8] classes describe the available objects[9] in the system.

When creating an object of type *HStation*, we say that we are using this station in our measurement, so we create a measurement station the following way:

```
Station *station = Station::loadFromDb(1, true);
HStation hStation(&m, station);
```

[7]An index is always an integer

[8]A base class is the class where it is inherited from

[9]An available object is one out of a station, a device, a task and a taskparameter

First we need to search for an available station in the database. With the assumption we already know about the available stations in the system, we load the station with id=1. The second parameter sets recursive to *true* meaning that all subelements (devices, tasks and taskparameters) are also loaded.

Now we can use the available station to create a measurement station. The measurement object is also passed to the constructor, so that the class automatically attaches the station to the measurement.

HDevice

Once having attached a station to a measurement, we can use this station to attach a device. For this purpose we select an available device and pass it as a parameter to the constructor.

```
Device *device = station->getDevice(0);  // logical device
HDevice hDevice(&hStation, device);
```

Here, the same happens as when creating a measurement station. An available device is selected from the station and used as a parameter when constructing a device object used for the measurement.

HTask

We follow the technique and choose a task offered by our available device and use it for the measurement.

```
Task *taskLoop = device->getTask(0);
HTask hLoop1(&hDevice, taskLoop);
```

Consider the first task of our selected device beeing a loop task explained above in the section of logical tasks; we are now going to create a loop by the construction with the device and selected loop task as parameters.

The next step is to complete our structure by adding the taskparameters needed by this task. After that, we assign a sequence id which controls the position in our task tree or the execution order respectively. More about that later.

HTaskParameter

Assuming that we know about the available task parameters of a given task, we use them to create the actual task parameter for the measurement task.

```
HTaskParameter htp(&measurementTask, availableTaskParameter);
hWaitTaskParameter1.setValue("101, 201, 202");
```

Result

A result always belongs to a task, be it pysical or virtual.

Listing 3.4: Creating and saving a result

```
/* Create a Result, set Value and save! */
Result result(htask);
result.setValue(res);
result.Write2db();
```

3.2.2 Connection classes

For this thesis, all devices work with the same connection type (IEEE488 Bus), but the system has been designed in a way, that it would be easily possible to add another connection for communication between devices and the computer. Figure 3.5 is showing an abstract class which should act like an interface for extending classes and forcing them to be a singleton classes by providing a static `Instance()` function.

There must be an instance for every connection type, so it is not possible to hold the static instance object in the Connection class, but every extending class must do it by itself. While there are some possibilities in the Java programming language, the C++ language is very severe and this is the only solution I've found.

Figure 3.5 shows the *IEEE488* connection class which is actually an implementation of the singleton pattern but in the same time an object of type *Connection*. This matter of fact is very useful, since every connection can be accessed over the *Connection* class[10]. On the other way, a programmer of DLL modules has the possibility to use connection specific functions when creating virtual tasks. The IEEE488 connection for example has the additional functions `ppoll(int channel)` and `spoll(int channel)` for a serial or parallel polling. These 2 functions are not implemented in a standard connection since it is a connection dependent behavior.

[10]When the TaskMachine sends or reads from a device, it doesn't care of which connection is actually being used. This is done by dynamic binding of virtual functions

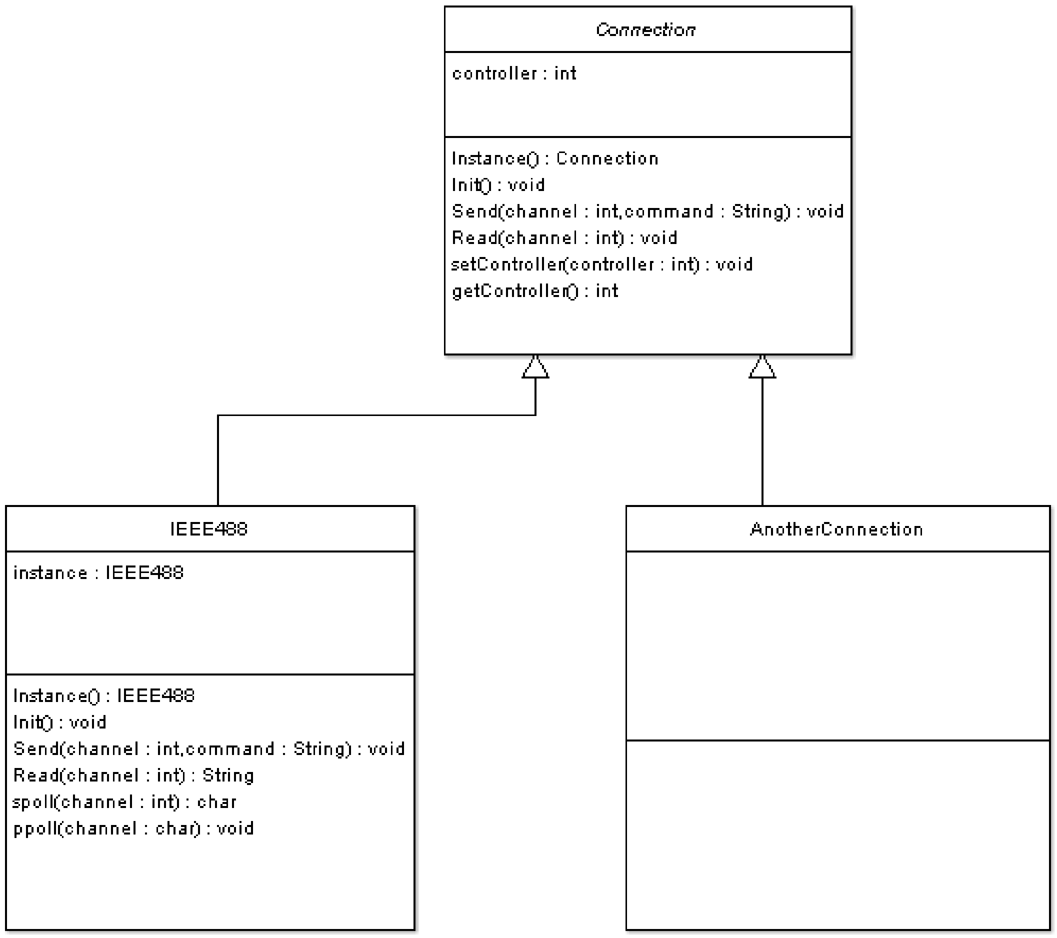

Connection
controller : int
Instance() : Connection
Init() : void
Send(channel : int,command : String) : void
Read(channel : int) : void
setController(controller : int) : void
getController() : int
IEEE488
instance : IEEE488
Instance() : IEEE488
Init() : void
Send(channel : int,command : String) : void
Read(channel : int) : String
spoll(channel : int) : char
ppoll(channel : char) : void
AnotherConnection

Figure 3.5: The connection interface

3.3 Using Task++

Now let's do a simple example. We will create a for loop and load a reference from database. We take the assumption that the references are already stored.

Listing 3.5: An example of using Task++

```
// Get Available Data (Stations, Devices,
// Tasks and TaskParameters)
Station *station = Station::loadFromDb(1, true);

// Load the devices we need
Device *logical = station->getDevice(0);
Device *keithley2750 = station->getDevice(1);
Device *solartron1286 = station->getDevice(2);

// Load the tasks we need
Task *_loop     = logical->getTask(0);
Task *_if       = logical->getTask(1);
Task *_else     = logical->getTask(2);
Task *_wait     = logical->getTask(4);
Task *v_task    = keithley2750->getTask(3);
Task *p_task    = solartron1286->getTask(1);

// Define the taskparameters we need
TaskParameter *loop_from = _loop->getParameter(0);
TaskParameter *loop_to   = _loop->getParameter(1);
TaskParameter *loop_step = _loop->getParameter(2);
TaskParameter *loop_var  = _loop->getParameter(3);

TaskParameter *if_arg1    = _if->getParameter(0);
TaskParameter *if_op      = _if->getParameter(1);
TaskParameter *if_arg2    = _if->getParameter(2);
TaskParameter *_wait_time = _wait->getParameter(0);

TaskParameter *delay      = v_task->getParameter(0);

// Create a new measurement
Measurement *m =
  new Measurement("M1", "First Measurement");

// Define a new measurement station (HStation)
HStation *h_station = new HStation(m, station);

// Define new measurement devices (HDevice)
HDevice *h_logical   = new HDevice(h_station, logical);
HDevice *h_keithley  =
  new HDevice(h_station, keithley2750);
```

```
HDevice *h_solartron =
  new HDevice(h_station, solartron1286);

// Define new measurement tasks (HTask)
HTask *h_loop1 = new HTask(h_logical, _loop);
HTask *h_loop2 = new HTask(h_logical, _loop);
HTask *h_if    = new HTask(h_logical, _if);
HTask *h_else  = new HTask(h_logical, _else);
HTask *h_wait  = new HTask(h_logical, _wait);
HTask *h_v_task = new HTask(h_keithley, v_task);
HTask *h_p_task = new HTask(h_solartron, p_task);

// Loop 1 Parameters
HTaskParameter *h_loop1_from =
  new HTaskParameter(h_loop1, loop_from);
HTaskParameter *h_loop1_to   =
  new HTaskParameter(h_loop1, loop_to);
HTaskParameter *h_loop1_step =
  new HTaskParameter(h_loop1, loop_step);
HTaskParameter *h_loop1_var  =
  new HTaskParameter(h_loop1, loop_var);

// Loop 2 Parameters
HTaskParameter *h_loop2_from =
  new HTaskParameter(h_loop2, loop_from);
HTaskParameter *h_loop2_to   =
  new HTaskParameter(h_loop2, loop_to);
HTaskParameter *h_loop2_step =
  new HTaskParameter(h_loop2, loop_step);
HTaskParameter *h_loop2_var  =
  new HTaskParameter(h_loop2, loop_var);

// If 1 Parameters
HTaskParameter *h_if_arg1 =
  new HTaskParameter(h_if, if_arg1);
HTaskParameter *h_if_op   =
  new HTaskParameter(h_if, if_op);
HTaskParameter *h_if_arg2 =
  new HTaskParameter(h_if, if_arg2);

// Wait task parameter
HTaskParameter *h_wait_time =
  new HTaskParameter(h_wait, _wait_time);


// Virtual task parameter
HTaskParameter *h_delay =
  new HTaskParameter(h_v_task, delay);
```

```
// Set loop1 parameters
h_loop1_from->setValue("1");
h_loop1_to->setValue("20");
h_loop1_step->setValue("1");
h_loop1_var->setValue("$i");

// Set if parameters
h_if_arg1->setValue("$i");
h_if_op->setValue(Logic::OP_GREATER);
h_if_arg2->setValue("10");

// Set loop2 parameters
h_loop2_from->setValue("500");
h_loop2_from->setType(TaskParameter::TYPE_VARIABLE);
h_loop2_to->setValue("1000");
h_loop2_step->setValue("100");
h_loop2_var->setValue("$j");

// Set v_task parameters (delay)
h_delay->setValue("$j");
h_delay->setType(TaskParameter::TYPE_VARIABLE);

// Set wait time
h_wait_time->setValue("500");

// Set Sequence Id to the Tasks
// SequenceID defines execution order

h_loop1->setSequenceId("1");

  h_if->setSequenceId("1.1");
    h_loop2->setSequenceId("1.1.1");
      h_v_task->setSequenceId("1.1.1.1");

  h_else->setSequenceId("1.2");
    h_wait->setSequenceId("1.2.1");
    h_p_task->setSequenceId("1.2.2");


try {
  m->write2db(true);
  std::cout << "\n\n\n";
  tree<HTask*> tr = m->getTaskTree();
  Util::Instance()->printTaskTree(&tr);
  m->execute();
}
catch(std::exception &ex) {
  std::cout << ex.what() << std::endl;
  }
```

The code listing shown above is actually what the GUI does and produces the following which can by graphically understood like shon in figure 3.6. At the first moment, this short syntax seems to need much code. Consider that in the same moment, the object tree becomes also stored in the database and all relations are set correctly.

Listing 3.6: The console output of the tasktree

```
0 − # ROOT #
1 − FOR(from 1 to 20 ; step 1; variable $i)
1.1 − IF($i > 10)
1.1.1 − FOR(from 500 to 1000 ; step 100; variable $j)
1.1.1.1 − KEITHLEY2750V1($j)
1.2 − ELSE
1.2.1 − WAIT(500)
1.2.2 − BK
```

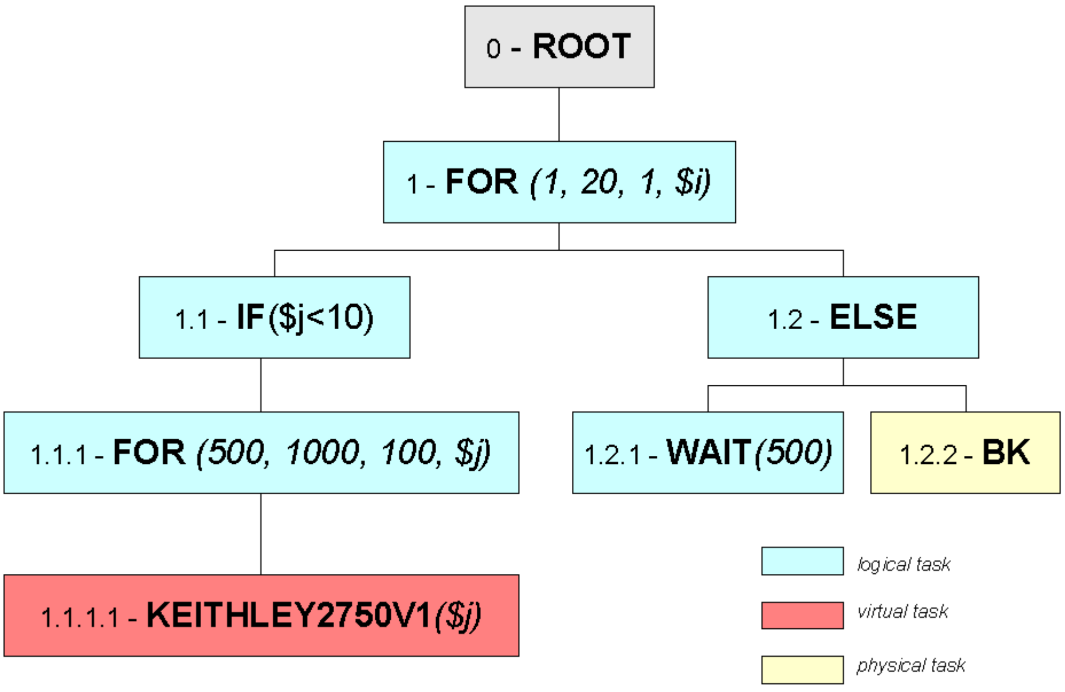
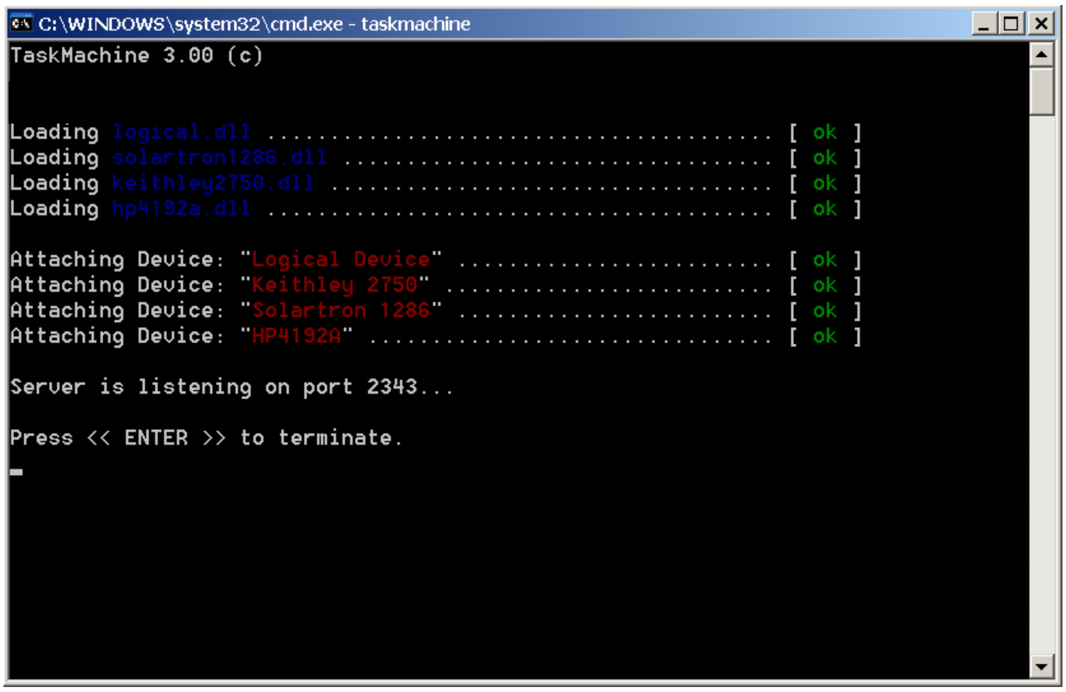


Figure 3.6: The Syntaxtree

3.4 The TaskMachine

The controller module in the system was formerly called *HardwareController*. It has been programmed by Kalchgruber Martin. In order to satisfy the urgently needed functionality, only a subset of task was possible to execute. Is very hard coded and does not feature the design of a good object oriented programming.

The TaskMachine is the actual reimplementation of the *HardwareController*. It has been a brand new start using some requirement knowledge gained while analyzing the old program. The TaskMachine is a server-side program running on a windows system. Once started successfully, it is listening for a *START* signal of a measurement from any client in the network.



```
TaskMachine 3.00 (c)


Loading logical.dll ......................................... [ ok ]
Loading solartron1286.dll ................................... [ ok ]
Loading keithley2750.dll .................................... [ ok ]
Loading hp4192a.dll ......................................... [ ok ]

Attaching Device: "Logical Device" .......................... [ ok ]
Attaching Device: "Keithley 2750" ........................... [ ok ]
Attaching Device: "Solartron 1286" .......................... [ ok ]
Attaching Device: "HP4192A" ................................. [ ok ]

Server is listening on port 2343...

Press << ENTER >> to terminate.
```

Figure 3.7: The TaskMachine

3.4.1 Startup

In figure 3.8 we see that the *TaskMachine* works in singleton modus[11] We can obtain our singleton class by calling the TaskMachine::Instance() method. Be-

[11] Singleton is a design pattern that ensures the existence of maximal one instance of a class.

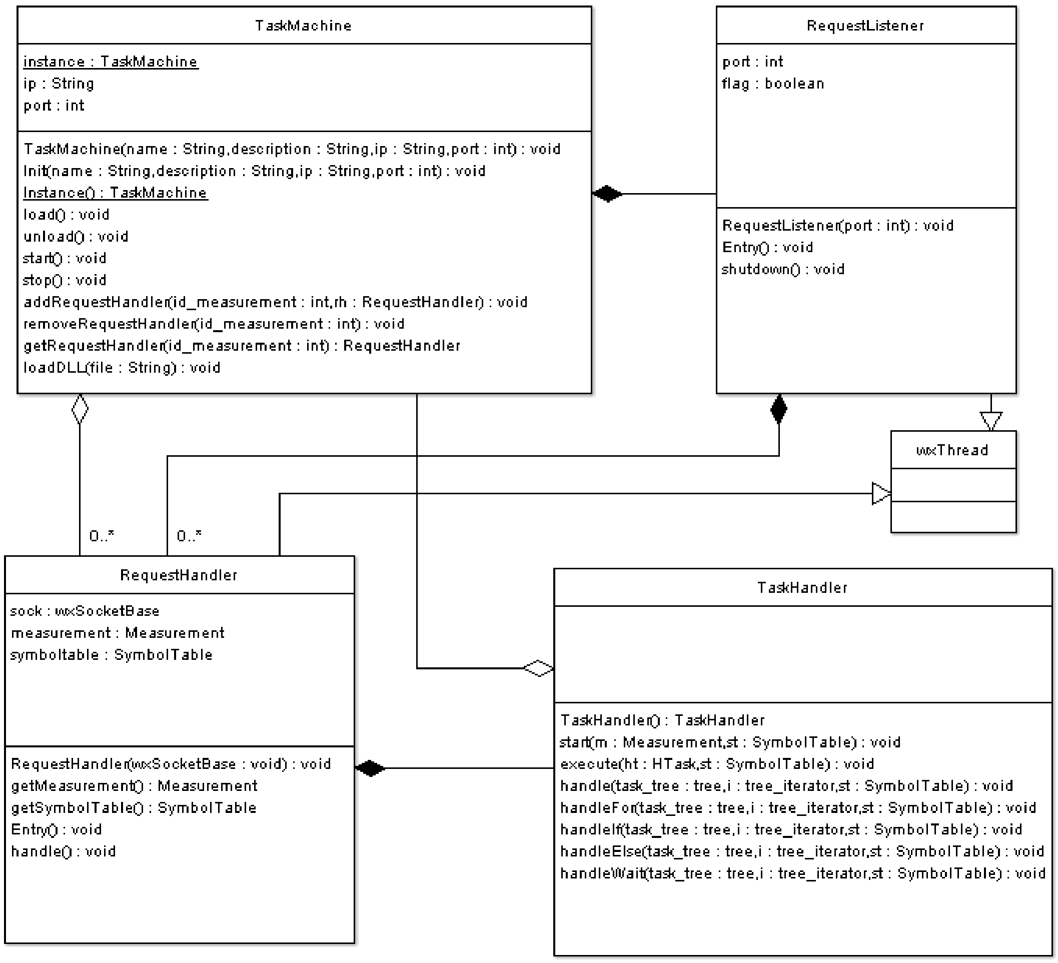

TaskMachine
instance : TaskMachine
ip : String
port : int
TaskMachine(name : String,description : String,ip : String,port : int) : void
Init(name : String,description : String,ip : String,port : int) : void
Instance() : TaskMachine
load() : void
unload() : void
start() : void
stop() : void
addRequestHandler(id_measurement : int,rh : RequestHandler) : void
removeRequestHandler(id_measurement : int) : void
getRequestHandler(id_measurement : int) : RequestHandler
loadDLL(file : String) : void
RequestListener
port : int
flag : boolean
RequestListener(port : int) : void
Entry() : void
shutdown() : void
wxThread
0..*
0..*
RequestHandler
sock : wxSocketBase
measurement : Measurement
symboltable : SymbolTable
RequestHandler(wxSocketBase : void) : void
getMeasurement() : Measurement
getSymbolTable() : SymbolTable
Entry() : void
handle() : void
TaskHandler
TaskHandler() : TaskHandler
start(m : Measurement,st : SymbolTable) : void
execute(ht : HTask,st : SymbolTable) : void
handle(task_tree : tree,i : tree_iterator,st : SymbolTable) : void
handleFor(task_tree : tree,i : tree_iterator,st : SymbolTable) : void
handleIf(task_tree : tree,i : tree_iterator,st : SymbolTable) : void
handleElse(task_tree : tree,i : tree_iterator,st : SymbolTable) : void
handleWait(task_tree : tree,i : tree_iterator,st : SymbolTable) : void

Figure 3.8: Class diagram for the TaskMachine

fore starting the *TaskMachine* we need to initialize it with some parameters which are read from the *taskmachine.ini* configuration file shown in listing 3.7.

Listing 3.7: The TaskMachine's configuration file

```
; TaskMachine (c) Configuration File
[Station]
Name=My Station
Description=A Station for Measurements
IP=192.168.0.10
Port=2343
Number=1
```

```
[Database]
db_host=192.168.0.11
db_port=3306
db_name=taskpp
db_user=taskpp
db_pass=taskpp
```

The configuration file gives the *TaskMachine* a name, a description, an ip address, a port and a unique number in the system. It also provides database access information. After the database connection has been established successfully, we simply call the TaskMachine::start() method to start the *TaskMachine*.

When starting up the *TaskMachine*, the program loades the dynamic libraries containing the devices functionality. For this purpose the TaskMachine scans the configuration directory */conf* which should contain a configuration file for each device attached onto the station the TaskMachine is running on.

The configuration file is in the windows .ini format and contains the following information divided into 2 sections:

Device: The settings for the device

- **Name:** The name of the device (e.g. Solartron 1286)
- **Description:** A short description which helps the user to understand or to remember it's functionality.
- **Number:** A unique number given to the device onto the station. Devices on different stations may have the same numbers.
- **dll:** This is the main configuration of .ini file. It tells the TaskMachine where to find the dynamic library that implements the device functionality. More about this in a later section.

Connection: The settings for the connection

- **Name:** The name of the connection this device uses to communicate with the TaskMachine. At the moment only the IEEE488 connection is available. Aliases are IEEE488.1 and GPIB.
- **Channel:** Depending on the communication interface the device is using to talk to the computer, a channel can be assigned when the connection supports the devices to talk onto different channel. I.e. the GPIB bus allows the assignment of 30 different channels.

Listing 3.8: Sample configuration file for a device

```
; Configuration File for Device
; KEITHLEY 2750

[Device]
Name=Keithley 2750
Description=Model 2750 Multimeter / Switch System
Number=2


; DLL File implementing the funcionality
dll=keithley2750.dll

[Connection]
Name=IEEE488
; Channel on GPIB Bus
Channel=16
```

Once we know the location of our dynamic link library (DLL), we make use of Run-Time Dynamic Linking to load the library into runtime. This is done with the function LoadLibrary or LoadLibraryEx respectively. After a successful loading of the library, we use GetProcAddress to get the function pointer of the exported function.

Listing 3.9: Run-time linking to the DLL

```
HINSTANCE device = LoadLibrary(fileSrc.str().c_str());

  if (device) {
    _getDllDevice = (getDllDevice)
          GetProcAddress(device, "getDllDevice");

    if (_getDllDevice) {
      DllDevice *d = _getDllDevice();
      d->setChannel(channel);
      instance->dllMap[u->int2string(id)] = d;
      cout << " [ " << cc::fg_green << "ok"
           << cc::fg_white
           << " ]" << endl;
    }
    else {
      std::cout << " [ "<< cc::fg_red << "failed"
                << cc::fg_white
                << " ]\n";
    }
  }
  else {
    std::cout << " [ "<< cc::fg_red << "failed"
              << cc::fg_white << " ]\n";
  }
}
```

3.4.2 Running

Once the *TaskMachine* has been started successfully, the *RequestListener* (see class diagram in figure 3.8) is listening on the port we set in the ini configuration file. As soon as a connection is accepted, the *RequestListener* delegates its work to the *RequestHandler* which reads the command buffer from the socket and handles accordingly. For the moment, only the commands **START** and **STOP** with the measurements id appended are recognized.

Listing 3.10: Handling a request and executing a measurement

```
// Get the singleton instance of the TaskMachine
TaskMachine *tm = TaskMachine::Instance();

// Get the measurement id
int mId = atoi(s.substr(5).c_str());

// Add the RequestHandler to the TaskMachine
tm->addRequestHandler(mId, this);

// Load the entire measurement from database
m = Measurement::loadFromDb(mId, true);

// Create a new SymbolTable for this measurement
st = new SymbolTable(mId);

// Start the measurement
taskHandler->start(m, st);

// Remove the measurement from the TaskMachine
tm->removeRequestHandler(mId);

// Clean up memory
m->destroy();
delete st;
```

3.4.3 Shutdown

For simplicity reasons, the *TaskMachine* works in command line modus and can be terminated by pressing enter or the CTRL-C signal. All devices become detached from the *TaskMachine* until the *TaskMachine* removes itself from memory.

3.5 The TaskDLL framework

TaskDLL is the framework which is used when creating new DLL modules for new devices. It is a very basic library and is essentially an implementation of the command pattern.

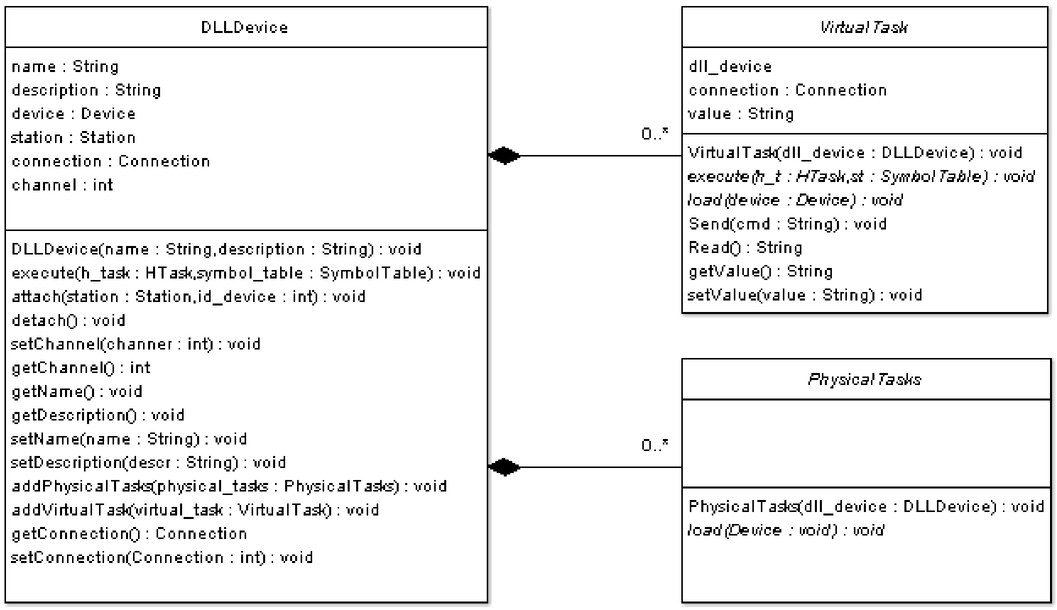

DLLDevice
name : String
description : String
device : Device
station : Station
connection : Connection
channel : int
DLLDevice(name : String,description : String) : void
execute(h_task : HTask,symbol_table : SymbolTable) : void
attach(station : Station,id_device : int) : void
detach() : void
setChannel(channer : int) : void
getChannel() : int
getName() : void
getDescription() : void
setName(name : String) : void
setDescription(descr : String) : void
addPhysicalTasks(physical_tasks : PhysicalTasks) : void
addVirtualTask(virtual_task : VirtualTask) : void
getConnection() : Connection
setConnection(Connection : int) : void
0..*
VirtualTask
dll_device
connection : Connection
value : String
VirtualTask(dll_device : DLLDevice) : void
execute(h_t : HTask,st : SymbolTable) : void
load(device : Device) : void
Send(cmd : String) : void
Read() : String
getValue() : String
setValue(value : String) : void
0..*
PhysicalTasks
PhysicalTasks(dll_device : DLLDevice) : void
load(Device : void) : void

Figure 3.9: Class diagram for the TaskDLL framework

Each DLL file scanned by the *TaskMachine* provides an exported function named getDLLDevice(). This function is implemented in the DLL generator and simply returns a *DLLDevice* object. The creation of DLL modules is shown in the next chapter.

Listing 3.11: Attaching a device

```
void DllDevice::attach(Station *station, int id_device) {
  using namespace std;

  // Avoid attaching a device twice
  if (device == 0) {
    // set station the DLL device belongs to
    this->station = station;

    // Create the device and set the number (id_device)
    this->device = new Device(station, name, description);
    this->device->setIdDevice(id_device);
    // Load physical Tasks
    for (std::vector<PhysicalTasks*>::iterator
         i = p_tasks.begin();
         i != p_tasks.end(); ++i) {
      (*i)->load(device);
    }

    // Load virtual Tasks
    for (std::vector<VirtualTask*>::iterator
                i = v_tasks.begin();
         i != v_tasks.end(); ++i) {
      (*i)->load(device);
    }
    // Write the device to database
    this->device->write2db(true);
  }
}
```

Once we have the DLLDevice object, we can attach it to the station which loaded the DLL file by calling its `attach(Station*, int)` method with our station object as the first argument and the device number (gained from the devices .ini file) as second argument. We only load the device if we already didn't (`if (device==0)`). We create the device, set the id and load all belonging virtual and physical tasks before writing it to the database. From this moment on, all information about the device can be accessed by the GUI client.

Chapter 4

Creating DLL modules

The creation of DLL modules is the point this project builds up. When creating DLL modules, we make use of the *TaskDLL* framework described previously.

4.1 Overview

Physical tasks: We create the physical tasks by extending the class PhysicalTasks of the namespace taskdll and implementing it's load(Device∗) function. The loading procedure of a device has been shown above and for each *PhysicalTaks* element present, the load function becomes executed. Creating physical tasks should not really be the problem, since the load implementation only contains an instantiation list of tasks and taskparameters, where the tasks values correspond to the ASCII command sent to the device. An example is shown later.

Virtual task(s): These type of tasks just differ from physical tasks in the way that sending and reading from/to the device must be implemented by the programmer. A virtual task can be considered as a logical sequence of some physical tasks. In this case, for each virtual task, the class *VirtualTask* must be extended and the functions load(Device∗) and execute(HTask∗, SymbolTable∗) must be implemented. The load function acts in the same way as the function of the physical tasks does.

4.2 Creating physical tasks

Consider a subset of the command set for the Keithley Multimeter/Switch System shown in chapter 3:

Listing 4.1: A command subset of Keithley2750

```
STATUS:OPERATION:ENABLE 2
STATUS:OPERATION:ENABLE?
STATUS:PRESET
STATUS:QUEUE:ENABLE
STATUS:QUEUE:ENABLE?
```

These commands are considered to be *physical tasks*, since they are sent to and interpreted from the device directly. In this case, the DLL file does not execute the commands but it acts as an information holder. The information is held in an object oriented way.

Listing 4.2: Header file for the physical tasks

```
#include "taskdll.h"
#include "task++.h"

using namespace taskdll;
using namespace taskpp;

class Keithley2750P1 : public PhysicalTasks {
  public:
    inline Keithley2750P1(DllDevice *dllDevice)
      : PhysicalTasks(dllDevice) { }

    void load(Device *device);
};
```

Listing 4.3: Implementation of the physical tasks

```
#include "Keithley2750P1.h"


void Keithley2750P1::load(Device *device) {
/* Create some handy shortforms ;-) */
  typedef TaskParameter TP;
  typedef Task T;
  char INT = TaskParameter::TYPE_INTEGER;
  char p = Task::TYPE_PHYSICAL;
  Device *d = device;

  /* STATUS  branch */
  T *status = new T(device, "Status", "STATUS");
    /* OPERATION branch */
    T *op = new T(status, "Operation", "OPERATION",  p, ":");
      T *en = new T(op, "Enable", "ENABLE", p, ":");
      T *en1_= new T(op, "Enable?", "ENABLE?", p, ":");

    T *preset = new T(status, "Preset", "PRESET", p, ":");
    /* QUEUE branch */
    T *queue = new T(status, "Queue", "QUEUE", p, ":");
      T *en2 = new T(queue, "Enable", "ENABLE", p, ":");
      T *en2_= new T(queue, "Enable?", "ENABLE?", p, ":");
      TP *tp_en1 = new TP(en1, "Slot number", "", INT, " ", "");
    TP *tp_en2 = new TP(en2, "Queue number", "", INT, " ", "");
}
```

So what is happening here? Every header file of a *PhysicalTasks* derivation is same except the name of the class. The implementation of the physical tasks (shown in listing 4.3) includes the according header file. The load(Device*) function becomes overridden which follows the command pattern approach. The first part of the function just contains some shortform definitions which make life easier (instead of writing Task, we just write T, etc.). After that, we create a task by telling to the constructor that this task belongs to our device, which has been passed as a parameter to the function. The task is named "Status" and it's physical value[1] is *STATUS*.

For the command set shown in listing 4.1, the "Status" command defines a command group and so cannot stand alone. Another command group is the "Operation" command which is a subgroup of the "Status" command. We create this command group by telling to the constructor that this task belongs to the "Status" task. The API automatically hooks it into the "Status" task. This approach could be repeated infinitely. It is not always the best solution to nest the commands. Every command could also be created with a single task, setting the task's value to the entire ASCII string (i.e.: STATUS:QUEUE:ENABLE?). It depends on the command set's complexity which approach to use.

Creating a TaskParameter follows the same approach: the constructor is told which task it belongs to, the name, a preset value which in most cases is empy, the type, a prefix and a postfix. See the Task++ API for details. We don't have to worry about memory clean up, since this is done automatically from the TaskMachine when the device gets detached.

4.3 Creating virtual tasks

Listing 4.4: Header file of a virtual task

```
#include "taskdll.h"
```

[1]The pysical value is the ASCII command (or command part) sent directly to the device

```
#include "task++.h"

using namespace taskpp;
using namespace taskdll;

class Solartron1286V1 : public VirtualTask {

public:
  inline Solartron1286V1(DllDevice *dllDevice)
    : VirtualTask(dllDevice) { }

  void load(Device *device);
  void execute(HTask *htask, SymbolTable *st);

};
```

Listing 4.5: Implementation of a virtual task

```
void Solartron1286V1::execute(HTask *htask, SymbolTable *st) {
  using namespace std;
  typedef TaskParameter P;
  Util *u = Util::Instance();

  // Get (H)TaskParameters from (H)Task.
  P *p1 = htask->getParameter(0);
  P *p2 = htask->getParameter(1);

  int sample_count = atoi(p1->parseValue(st).c_str());
  int sample_interval = atoi(p2->parseValue(st).c_str());

  Send("BK4");
  Sleep(1000);
  Send("GP1");
  Send("TR1");
  Send("PW1");
  Send("RU1");

  for (int i = 0; i<sample_count; i++) {
    /* Read data */
    string res = Read();

    /* Create a Result, set value and save! */
    Result result(htask);
    result.setValue(res);
    result.Write2db();

    Sleep(sample_interval);
  }
}
```

Creating virtual tasks is slightly more different to the creation of physical tasks. Basically, for each virtual task a new class must be generated (every command is encapsulated into an object[Vli95]). Generally, this requires a little bit of typing work, but the abstraction is high enough that a code generator or a class generating tool could assume the work. But now let's see how the things work.

The header file of a virtual task is almost the same as the one for physical tasks, in addition there is an *execute* function which takes an HTask and a SymbolTable as parameters. While the *PhysicalTasks* class only contains a list of task objects which are attached to the device when calling the load function, the load function of the class *VirtualTask* only loads one virtual task. The implementation is shown in listing 4.5. In the load function we create a single Task which can also have TaskParameters. The value given to the tasks **must be unique** for this DLL since this value acts as an identifier for our virtual task. The creation of a TaskParameter should be a known process. When the load function becomes executed while the TaskMachine is starting, the virtual task becomes written to the database and is shown as available in the GUI.

The execute function contains the actual implementation of the virtual task. It is a mixture of ASCII commands sent to device as well as C++ code. When in the GUI the virtual task is selected, all of the stuff in the execute function becomes executed. We have the possibility of fetching task parameters and creating results. See the Task++ API for details.

4.4 Creating the DLL

In an early discussion we talked about the need of exporting the function which the TaskMachine's device loader uses as the entry point to the DLL file. For this purpose we make use of the exporting directive defined in the preprocessor variable DEVICEDLL_EXPORT.

Let's assume we're having a list of physical tasks and two virtual task implementations. So we would have three classes. Listing 4.6 shows how a DLL is created and how the tasks are attached to it.

Listing 4.6: Exporting the linking function

```
#include "taskdll.h"
#include "Keithley2750.h"
#include "Keithley2750V1.h"
```

```
#include "Keithley2750V2.h"

EXPORT_DLL DllDevice *getDllDevice() {
  // Create a new DLL
  DllDevice *dllDevice = new DllDevice();
  // Set the connection
  dllDevice->setConnection(IEEE488::Instance());

  // Create the Tasks
  PhysicalTasks *pt = new Keithley2750(dllDevice);
  VirtualTask *vt1 = new Keithley2750V1(dllDevice);
  VirtualTask *vt2 = new Keithley2750V2(dllDevice);

  return dllDevice;
}
```

After a succesful compilation, the DLL file has been created and can be copied into the TaskMachine's /devices directory. A configuration file must be created and copied into the /conf directory (see 3.8).

Chapter 5

Conclusions

In this thesis, many aspects of modularization have been shown. After a detailed description of the communication bus in chapter 1, chapter 2 shows the basic aspects of linking. Both, dynamic linking and static linking have been used for the creation of the software this thesis is about. Since a main aspect is the creation of DLL modules, we also explained the distinction between *Load-time-* and *Run-time* dynamic linking. The benefits of *Load-time* dynamic linking have been used by the creation of reusable objects, saying the *Task++* and the *TaskDLL* API which is implemented once and used by the TaskMachine and the client GUI respectively. Changes done in the implementation do not affect the API and in turn the functionality of the using components. So the library can simply be replaced. This feature covers the aspects of *Load-Time Dynamic Linking* while the creation of DLL modules covers the aspects of *Run-Time Dynamic Linking*.

Chapter 3 shows the most important implementation aspects and the underlying database. This chapter is important for those who want expand or refactorize the functionality of the basic modules *Task++* and *TaskDLL*. It is also important to the person who integrates the functionality into the GUI client. At this moment, the assembly of the client with the *Task++* components was not completely done, but our test clients proved the functionality.

Chapter 4 gives an introduction to the creation of DLL modules and some examples. This chapter helps to give a basic overview and should be the starting point of the creation of an instruction manual which gives a detailed overview how to create DLL's.

List of Figures

Listings

Bibliography

[Coc96a] Adrian Cockcroft. What are the tunable kernel parameters for solaris 2? Technical report, `http://sunsite.cs.msu.su/sunworldonline/swol-01-1996/swol-01-perf.html`, January 1996.

[Coc96b] Adrian Cockcroft. Which is better, static or dynamic linking? Technical report, `http://sunsite.uakom.sk/sunworldonline/swol-02-1996/swol-02-perf.html`, 1996.

[Dre] Ulrich Drepper. Static linking considered harmful. Technical report, Red Hat, `http://people.redhat.com/drepper/no_static_linking.html`.

[eia99] Interface standard for nominal 3 v/3.3 v supply digital integrated circuits. Technical report, Electronic Industries Alliance, 1999.

[Gro02] Sandeep Grover. Linkers and Loaders. Technical report, `http://www.linuxjournal.com/article/6463`, 2002.

[Ins] National Instruments. History of GPIB. Technical report, `http://zone.ni.com/devzone/cda/tut/p/id/3419`.

[Ins88] Solartron Instruments. *1286 Electrochemical Interface - Operating Manual.* Solartron Instruments, 1988.

[Ins03] National Instruments. The HS488 protocol. Technical report, `http://zone.ni.com/devzone/cda/tut/p/id/4283`, 2003.

[Kal06] Martin Kalchgruber. Messablaufsteuerung, 2006.

[Kei03] Keithley. *Model 2750 Multimeter/Switch System User's Manual*, 2003.

[Mic] Microsoft. Visual studio, microsoft portable executable and common object file format specification. Technical report, `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, 2006.

[Mic06] Microsoft. What is a DLL? Technical report, http://support.microsoft.com/kb/815065/EN-US/, May 2006.

[Pur99] Andy Purcell. The search for a GPIB replacement. *Autotestcon, 1999. IEEE Systems Readiness Technology Conference, 1999. IEEE*, pages 169–177, 30. aug. - 02.sep. 1999.

[Ryn05] John Rynland. Can LXI replace GPIB? *Autotestcon, 2005. IEEE*, pages 739–743, 26-29 sept. 2005.

[Sof05] Tech Soft. HTBasic GPIB tutorial. Technical report, http://www.techsoft.de/htbasic/tutgpibm.htm?tutgpib.htm, 2005.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4 edition, 2000.

[Str02] Bjarne Stroustrup. Sibling rivalry: C and C++. *The C/C++ Users Journal*, July, August, and September 2002.

[Tec96] Agilent Technologies. *HP 4192A LF Impedance Analyzer - Operation Manual*, 12 1996.

[Vli95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissiedes. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[wik06a] Wikipedia: C++. http://en.wikipedia.org/wiki/C++, 11 2006.

[wik06b] Wikipedia: IEEE-488. http://en.wikipedia.org/wiki/IEEE_488, 11 2006.

[wik06c] Wikipedia: Labview. http://en.wikipedia.org/wiki/Labview, 11 2006.

[wik06d] Wikipedia: MySQL. http://en.wikipedia.org/wiki/MySQL, 12 2006.

[wik06e] Wikipedia: Software. http://en.wikipedia.org/wiki/Software, 11 2006.

[Zso] Pápay Zsolt. GPIB tutorial. Technical report, http://www.hit.bme.hu/~papay/edu/GPIB/tutor.htm.