Realization of a framework for rapid development of distributed data-stream processing systems

Master's thesis

Moritz Fišer

Institute of Microwave and Photonic Engineering
Graz University of Technology

in cooperation with JOANNEUM RESEARCH

Supervisors:
Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Erich Leitgeb
Dipl.-Ing. Dr.techn. Franz Graf

Graz, February 2014

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz,	Date	Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am	Datum	Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgements

First I would like to express my sincere appreciation to Erich Leitgeb for his supervision and useful comments, as well as his cordial, welcoming disposition. He has been very supportive of my work and always took the time to answer any of my questions on short notice.

I would like to show my deepest gratitude to Franz Graf, who has been supporting me for many years, not only in the course of this thesis. I have greatly benefited from his experience and guidance. Not only is he very competent professionally, but he always has an open ear for the people around him and is the first one to lend a helping hand, even when it comes to carrying couches up the stairway.

I want to thank Bernhard Rettenbacher, name giver and father of the original *Chronos* framework, and Susanne Rexeis for valuable discussions concerning my work, as well as Harald Rainer for enduring them without a grumble (mostly).

Above all I would like to thank all of my family, particularly my parents Inge and Ilja Fišer, who gave me the opportunity to move to a new city for my studies and always supported me with valuable advice and encouragement. Without them none of this would have been possible.

Graz, February 2014 Moritz Fišer

Abstract

For this thesis a software framework with the purpose of simplifying and standardizing the work-flow of transforming a prototyped set of algorithms into a fully-fledged and expandable real-time stream processing system was to be designed and implemented. The system should be kept general and scalable to large, multimodal data stream applications, capable of being integrated into existing infrastructure. The software itself is closed source, however, an in-depth overview of its substance and a background on the underlying technologies will be given. The framework's use cases and work-flow as well as design criteria are going to be exemplified and the distinction to related frameworks characterized. After fundamentals have been established, the architecture is going to be described in greater detail, followed by the specifications needed to configure and utilize the framework for custom use. To further promote an understanding of the subject, three real-world applications employing the framework will be presented.

Zusammenfassung

Im Zuge dieser Diplomarbeit war ein Software-Framework zu entwickeln, das den Prozess der Transformation eines prototypisch entworfenen Signalverarbeitungsalgorithmus in ein voll funktionsfähiges und erweiterbares Echtzeit-System vereinfachen und vereinheitlichen sollte. Das Framework sollte möglichst allgemein gehalten sein, auf große, multimodale Datenstromapplikationen skalieren und in bestehende Infrastrukturen integrierbar sein. Die im Zuge dieser Arbeit entstandenen Quellcodes sind geschützt und nicht für eine Veröffentlichung vorgesehen. Es werden jedoch die wesentlichen Inhalte, Schnittstellen, sowie die verwendeten Technologien vorgestellt und erläutert. Die wichtigsten Entwurfskriterien für die Entwicklung, sowie Anwendungsfälle des Frameworks werden beschrieben und die Abgrenzung zu verwandten Frameworks charakterisiert. Nach Vermittlung der notwendigen Grundkenntnisse werden die Software-Architektur und die für den Einsatz in eigenen Applikationen nötige Vorgehensweise näher erläutert. Um das Verständnis für die Funktionalität des Frameworks zu vertiefen, werden drei in die Realität umgesetzte Anwendungen vorgestellt.

Contents

List of Figures

1. Introduction

This chapter will motivate this work and provide the reader with an introduction to the problem domain. A short review of the most relevant topics and related frameworks will be given.

1.1. Motivation

Development of a novel hardware/software product employing signal processing and pattern recognition algorithms is an intricate process and generally cycles through several stages. After a theoretical concept has been laid out, a first algorithmic prototype has to be developed. It is not recommendable to employ a low-level language like C to tackle this task right from the start, as it is time-consuming and cumbersome to fine-tune parameters or change algorithmic components as well as to easily get test data into or out of the algorithm. On these grounds, prototyping environments (e.g. MATLAB, Octave, SciPy) alleviating a lot of the work required to set up a first demonstration system are available on the market. However, it can be difficult to integrate these kinds of prototyped demonstrators into a real-world environment, employing various hard- and software-interfaces and requiring the algorithm to perform in an on-line manner. A classical approach would be to finalize the algorithm inside of a prototyping environment and in a second phase reimplement it in a for real-time applications more appropriate programming language. Subsequently the algorithm has to be embedded into a custom application, serving as an interface to the user and to external components like capturing-hardware or automation systems. At this point, the algorithm can be tested for meeting on-line requirements, that is - Is the algorithm capable of processing an incoming data stream in real-time? If the answer is no, the algorithm has to be revised and possibly reimplemented. It can be difficult to meet the on-line constraint if computations to be made are complex or the number of channels to process is large. Should an adaption of algorithms become necessary, it is sensible to go back into the prototyping environment, where the effect of algorithmic adjustments can easily be monitored. After the prototype has been adjusted, changes need to be translated to the on-line implementation once more. Keeping the prototype implementation and the on-line version coherent is an important but tedious task and can be prone to errors. Traditional development processes keep cycling between the two stages of prototyping and adaption of the on-line system until aspired goals are met.

In this thesis a software framework, *Chronos Realtime*, was to be developed, that would concentrate on finding a common denominator between different applications of on-line signal processing and pattern recognition systems and enable a consistent and straight-forward way to make the move from a prototyped algorithm to a comprehensive on-line system as little time-consuming as possible. The name *Chronos* stems from an already existent, house-internal framework developed at Joanneum Research. It has been used to support implementation of signal processing algorithms purely in MATLAB. *Chronos Realtime* is however not based on the original *Chronos* framework and is not to be understood as replacement, but as a complement. The two are related only semantically, covering different parts of the development cycle. Background of this work was most notably the development of various acoustical monitoring systems. However, much attention has been paid to the criterion of keeping the applicability of the framework as general as possible and hence suitable to a wide variety of use cases. In principle any kind of data may be processed if suitable extension modules are implemented. A multimodal use-case, integrating both audio and image streams has already been realized successfully (see section 6.2.2).

1.2. Software Framework

Seeking a better understanding of the subject, the term *software framework* shall be discussed first. As there is no unique and generally accepted definition available, the most distinctive characteristics shall be highlighted. In general, a framework is a real or conceptual structure intended to serve as a support or guide for the building of something that expands the structure into something useful [13f]. Similar to a software library, it offers a collection of reusable components, providing functionality common to a class of problems. However, a framework goes further than that. In contrast to an ordinary library, it also provides the user with a superordinate application architecture. The overall program's flow of control is laid out by the framework, which is referred to as inversion of control. This promotes a more consistent way of coding and facilitates the readability of application code for external developers familiar with the framework. Specific applications are customized by extension and configuration. Frequently, frameworks also tend to be more extensive than libraries, providing a holistic workflow and possibly a set of external tools to aid in the development of an application within the problem domain.

1.3. Problem Domain

The main objective of *Chronos Realtime* is to aid in development of big-scale on-line stream and signal processing systems. On-line can be understood as operating in real-time, but is not to be confused with the property of hard real-time, a common term in informatics [Tan09]. More specifically, the system is designed to meet soft real-time requirements, meaning occasional excessive processing delays are tolerable but should be kept to a minimum. The overall system is expected to be capable of managing the incurring amount of data without falling back over time. A minor amount of non-deterministic lag is acceptable. In the following, on-line and real-time are going to be used interchangeably. Big-scale in this context means that the number of signals to be processed in parallel could potentially be very large, reaching up to thousands. These systems typically need to be embedded into an existing infrastructure to interface with the outside world. Despite the heterogeneity of possible applications, they usually share common operational procedures. The various building blocks of a distributed *Chronos* application are going to be referred to as modules in the upcoming chapters. A module generally consists of a separate executable, implementing a framework-specific communication protocol and running on an arbitrary workstation connected to a computer network.

1.3.1. Real-Time Digital Signal Processing

Digital signal processing is concerned with the analysis or modification of digitized signals, most often motivated by the objective of extracting or enhancing contained information of interest. A signal defines the variation of some physical quantity as a function of one or more independent variables over time. If both time and signal values are discretized, the signal is referred to as digital. These signals generally originate from A/D converted sensor measurements. In this context a single reading of a sensor at one time-point is referred to as sample (not to be confused with the definition of a sample in the field of statistics) and is most often acquired at a fixed rate, the so called *sampling rate*. To enable processing of a signal in real-time, the applied algorithms need to be adequately fast to process at least as many input samples as are acquired within a certain epoch. When the sampling rate is higher than e.g. $>$ 20 Hz and the signal is one-dimensional, samples are commonly grouped together into so called *frames* and processed at once. To add to the confusion, in image processing the term *frame* usually refers to one image at a particular point in time (i.e. one two-dimensional sample). *Chronos Realtime* however is consistent with both of these nomenclatures, i.e. when processing audio signals, a frame consists of a consecutive sequence of samples (potentially of multiple channels sampled at the same time interval from different sensors), when processing image sequences a frame consists of a single image (or multiple images sampled at the same point in time from different sensors).

Figure 1.1 illustrates the basic structure of a simple but typical real-time signal processing application within the context of this framework. Input data is acquired from various sensors (in this case multiple microphones) and processed by one or several custom algorithms generating an application specific output. The output could either belong to the same domain as the input, or represent a different signal type, like a classification result. Depending on the setting, the output might either be

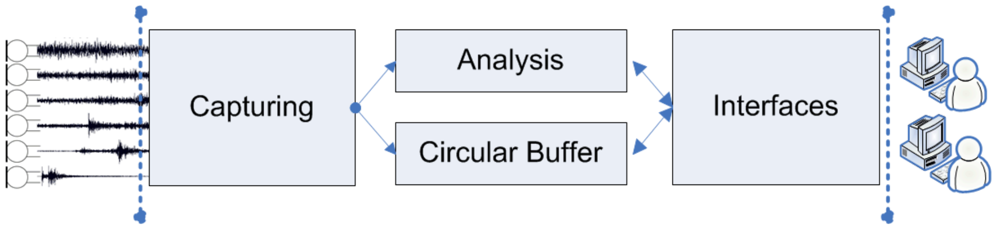


Figure 1.1.: Block Diagram of a Simple Processing Application

post-processed, forwarded to some sort of interface or cause specific actions within the scope of application. Data Acquisition, Processing and User Interfaces could potentially be located on different machines, according to hardware requirements, available processing power and geographical situation. Apart from the processing algorithms themselves, auxiliary components are often needed to fulfill application requirements. In the illustrated example the captured signals are additionally relayed to a circular buffer, enabling a user to listen to data from the past (e.g. after the analysis components have detected an event). The framework should not impose any restriction on these possibilities but promote an easy integration of heterogeneous components.

1.3.2. Algorithm Prototyping

Algorithm prototyping in the context of this work refers to the process of signal analysis and development of new solutions to problems within the domain of stream- and signal-processing. Most commonly these tasks are carried out by means of a specialized prototyping environment, consisting of a high-level scripting language and a set of domain-specific tools and libraries. Within such an environment, algorithms may be developed and tested quite conveniently. However, often execution times are prohibitively slow and integration of low-level device drivers and protocols might prove difficult. Making an algorithm capable of on-line processing requires additional consideration. This work suggests a way to enable a rapid development and integration of prototyped algorithms into *Chronos Realtime* using Simulink (see sections 2.2.1 and 3). Support of further prototyping environments (e.g. Python/SciPy) is conceivable as a future extension.

1.4. Design Goals

Before engaging in the design process of a piece of software, it is important to establish a set of conceptual priorities. Since different design goals often tend to be conflicting, the final decision can only result in a compromise. A typical example of such a trade-off would be execution speed versus memory consumption. For a software library or framework, these decisions generally tend to be even more tenuous, since final applications are not known during design phase. Therefore the most important goals shall be presented and discussed in the following.

1.4.1. Flexibility

Flexibility and generality are inarguably the primary objectives of any software framework. It is nevertheless a very sensible topic, as flexibility and ease of use often are conflicting attributes. A framework should put as little restrictions on its field of application as possible. On the one hand, common functionalities of applications should be encapsulated, on the other hand the desirable level of generality is very difficult to determine for unknown future applications. A too generic and flexible design can lead to an overly complex framework, drastically reducing its benefit over a custom solution. Going too far in the other direction however, might narrow down the application domain to only a very specific set of problems. Main area of application and background of this work was the domain of acoustic signal processing systems. However, a major design goal was to equally facilitate more interactive use cases and ones involving a variety of sensor signals. This is due to multi-modal processing and sensor fusion being important factors in achieving more reliable machine-made decisions. Therefore the framework's interfaces have not been devised specifically for audio signals, but are formulated so as to facilitate arbitrary signal types (as long as a sequential processing of these signals is reasonable).

1.4.2. Scalability

Signal processing and pattern recognition tasks can become very computationally expensive for complex problems. Particularly when a multitude of different signals needs to be processed simultaneously, even a powerful workstation may reach its limits. To tackle this problem, one of the main goals of this work is to innately support the distribution of an application over several computing devices. This way, an algorithm may be scaled to very large problems by simply adding more hardware, an approach commonly referred to as cluster or grid computing. The proposed targets for applications of this framework are standard PCs and workstations, however, any platform running Windows, Linux or Java is a potential candidate. The complexities generally arising from a distributed scenario should be encapsulated as much as possible by the framework. Processing of additional channels or separating an algorithm into several stages should require no knowledge about the internals of distributed computing by an application engineer. Her job is to devise how an algorithm might be split into several parallelizable parts on a higher level.

1.4.3. Extensibility

It should be possible to extend the framework through custom components without requiring too much effort or intricate knowledge of its internals. While a solid foundation of a common range of functions should be inherently provided, every novel application brings new requirements into the equation that cannot be accounted for during framework design. Interfaces and inheritable classes should therefore be designed in an understandable way that is easily extendible by an application developer. Integration of third party tools and libraries should be promoted by the architecture.

1.4.4. Cross-Platform, Cross-Language

Cross-Platform in context of *Chronos Realtime* does not only refer to the ability of compiling the whole application for several operating systems, but to actually distribute a single application across different platforms. This is provided for through the ICE middleware (section 2.1.2), available for Windows, Linux, MacOS as well as in plain source code. It allows multiple machines connected to a common network to work together on the same problem. Different hardware- and software-platforms tend to have distinctive advantages and disadvantages. It can therefore proof beneficial to choose the platform most fit for a particular application component. For example it might be advantageous to use MacOS for the task of audio capturing and Windows for providing a user interface in one specific use-case. The same argumentation applies to usage of different programming languages. While not one of the main priorities, this design goal plays a role in architecture and coding standard choices and is non-negligible.

1.4.5. Robustness

In certain areas, particularly monitoring scenarios, applications are expected to run for a very long time, potentially years non-stop. Downtimes may not only be expensive for the responsible party, but also impair safety of the monitored system. Distributed systems are typically susceptible to a lot of different error scenarios, some of which are difficult to predict or test beforehand. Clean error-handling possibilities are therefore a requirement that needs to be thought of at design time. If one succeeds in overcoming the associated difficulties, distribution can lead to increased robustness of the overall system. If only one component fails, all other independent components can continue to carry out their work without interruption. Critical components may be mirrored, so in case one component fails, the redundant one may take over on-the-fly. *Chronos Realtime* should be able to handle network failures and partial application crashes and never expect reliable communication between components. If errors occur they are to be logged accordingly. More advanced features like component mirroring are not part of the first version but are considered subject of future work.

1.5. Related Frameworks

1.5.1. CLAM

CLAM (**C**++ **L**ibrary for **A**udio and **M**usic) is an object-oriented open source research and application framework that originated at the Music Technology Group at Universitat Pompeu Fabra, Barcelona, with the aim to be a lingua franca for several projects that were taking place in the group [Ama04] [AA05] [13b]. As the name implies its main purpose lies in the development of audio and music applications. Like *Chronos Realtime* it offers stream based processing fit for real-time implementations. The project has been in development since 2000 and profits from the many contributions of different developers since, therefore offering a comprehensive set of tools. However, CLAM is licensed under the GPL scheme and thus unfit for many commercial applications (inquiries about the dual license scheme have remained unsuccessful). The main differentiating factors of *Chronos Realtime* are its inherent distribution and support of other media types than audio. Integration into the MATLAB/Simulink prototyping environment is also not given by CLAM. However, it would be feasible to add this functionality to the framework, due to its open and extendible nature.

1.5.2. DataTurbine

DataTurbine [Fou+12] [13d] is a JAVA based open source real-time data streaming middleware. Similarly to *Chronos Realtime* it may be used to read data from a sensor and stream it over a network to a number of recipients. Its focus however lies on the collection and reliable transport of sensor data and less on rapid prototyping and real-time signal processing. To this end it also follows a different network topology. Although similar from a user's point of view, as there are so called *Sources* (corresponding to *Data Providers* in *Chronos Realtime*) and *Sinks* (corresponding to *Data Clients* in *Chronos Realtime*), which may be "connected" to each other independently of their location within the network, in DataTurbine the flow of data takes an indirect route via a so called *DataTurbine Server*. This server acts as a circular buffer for data streams arriving from the sources, decoupling sources and sinks from each other. This approach has some advantages with regard to data safety and error handling compared to a direct streaming solution (as employed in *Chronos Realtime*), but adds some overhead and delay to the overall stream. Particularly if a *DataTurbine Server* is not located on the same machine as the respective sink, the stream has to take an extra indirection via an additional host. It should be mentioned that *Chronos Realtime* offers a circular buffer component, which may be used to emulate a behaviour similar to DataTurbine. In principle DataTurbine could have been adopted as a streaming backend for *Chronos Realtime*, however ZeroC's ICE offers better flexibility and performance in that respect. As a result there is a certain overlap in the application range of DataTurbine and *Chronos Realtime*.

1.6. Example Applications

To gain further insight into why the developed framework might be useful, it is advisable to skim through section 6.2, where a few example applications have been described. These applications are real-world use cases that have been implemented using *Chronos Realtime* or a precursor thereof. An introduction to the different use-cases will be given and the involved components as well as their interaction between each other will be covered.

2. Employed Technologies

In this chapter, an overview of technologies employed in this thesis shall be given. It is not meant as an extensive review of the different topics, but rather a justification as to why these particular technologies have been chosen. A brief overview will however be given on each subject, to provide the reader with sufficient knowledge for subsequent chapters.

2.1. Distributed Computing

Distributed computing is a field of computer science that studies distributed systems. A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. [Cou+11]

2.1.1. Ethernet

Ethernet (IEEE 802.3) is a networking standard developed for use in **L**ocal **A**rea **N**etworks. Originally it was specified for a bandwidth of 10 MBit/s in the year 1983 [Tan02], but due to its sweeping success has been revised several times. Today 1 GBit/s is standard for home and office workstations, with 100 GBit/s (IEEE 802.3ba) on the horizon. It is due to this tremendous speed increase, that distributed multimedia applications have gained a lot of attraction in recent years. Formerly the high data rates prohibited an extensive distribution and favored either local or expensive specialized solutions. An uncompressed PCM audio stream of CD quality (44100 Hz, 16 bit) accounts for a bitrate of 705.6 kBit/s per channel. This suggests that an idle 1 Gbit link would theoretically be capable of streaming slightly more than 1400 channels simultaneously. However there generally are several layers of protocols running on top of Ethernet handling the connection and streaming process on a higher level, which causes significant overhead. Despite this, the aforementioned numbers serve to exemplify the potential of audio streaming via Ethernet, even more so under the advent of 100 GBit technology. For video the rationale is similar, although video will usually still have to undergo some sort of compression for streaming. A conventional DVD quality video stream (MPEG2 encoded) has a maximum bitrate of 9.8 MBit/s, so around 100 of them could be transmitted in parallel via a 1 GBit link, again ignoring protocol overhead, packet loss and similar network specific obstacles. However, for real-time systems often lower quality streams are sufficient, leading to an even higher number of channels. The practically achievable goodput is strongly implementation dependent and has been measured for *Chronos Realtime* in different scenarios (see section 4.4) using the system clock. Due to the described advances in network technology, Ethernet streaming has recently become a big topic in the field of professional audio broadcasting. Protocol standards and products have been developed based on well established IP and Ethernet technology to replace the traditional analog distribution of audio with digital networking solutions. These new technologies are readily available and have already been successfully applied in live performances. Well-established examples are the commercial product called *Dante* by Audinate Pty Ltd [13a] and the open standard *RAVENNA* [13j].

2.1.2. Object-Oriented Middleware - ZeroC ICE

While Ethernet enables a basic network communication on a low level, in a distributed application it is desirable to use higher level abstractions to encapsulate the details of underlying protocols. Modern operating systems often provide implementations of different **R**emote **P**rocedure **C**all routines. They extend the well-known procedure call abstraction to distributed systems, trying to make a remote procedure invocation behave as if it were a local one [VKZ05]. Unfortunately they are platform specific and generally not designed for object oriented architectures. Therefore higher level software solutions have been developed to provide services that go beyond those offered directly by the operating system. These solutions are referred to as middleware, or more specifically distributed object middleware. The middleware sits in between the application and operating system layer and encapsulates the communication process between different applications that might be running on separate machines. No knowledge of where exactly a call is executed is required, as the interfaces generally behave is if they were local calls. However, one major difference to a regular function call that needs some extra consideration in any application using a middleware solution is the possibility of network failures. A robust distributed application must always consider the possibility of network failures and therefore requires very careful design, particularly in presence of multiple threads of execution.

Many different middleware architectures exist, e.g. message-oriented, message-passing, SQL-oriented and others. One common kind of architecture is the **O**bject **R**equest **B**roker. It allows an object-oriented treatment of remote invocations by introducing so called distributed objects. The most well-known representative is CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecure), which is an open standard defined by the **O**bject **M**anagement **G**roup. It enables separate pieces of software written in different programming languages and running on different hosts to work together as a single application. However, due to inconsistencies within the open implementations of the standard and partially poor documentation, an alternative called **I**nternet **C**ommunications **E**ngine by ZeroC Inc. [13l] has been evaluated for this work. ICE has many similarities to CORBA (an extensive comparison may be found in [Hen04]), but offers a simpler and more efficient design and is available under the GNU GPL as well as proprietary license models for commercial use. It supports the C++, Java, .Net, Ruby, Actionscript, Objective-C, Python and PHP programming languages on today's most common operating systems Linux, Solaris, Windows and MacOS X, as well as iOS and Android. Therefore a distributed application may take advantage of the benefits different languages and operating systems have to offer, to aid in tackling the problem at hand. ICE is intuitive to use and offers comparatively fast communication between different hosts. These properties make it appear to be a good backbone for a distributed real-time system. UDP as well as TCP may be utilized as a transport layer protocol. ICE uses an IDL (**I**nterface **D**efinition **L**anguage) for a programming language-independent specification of object interfaces. These specifications may then be used to automatically generate stub code in a desired language. The actual definition of declared methods is realizable by subclassing. Once instantiated and registered with the ICE runtime, a user-defined class becomes a servant, ready to accept remote or local method invocations. For a client to be able to call operations on this servant, it must hold a corresponding proxy object. A proxy is an artefact that is local to the client's address space and represents the (possibly remote) ICE object for the client [13e]. The necessary proxy code is implicitly generated by the aforementioned meta-compilation step. A proxy may be instantiated via explicit specification of a host address and port, as well as via a well-known name that may be resolved by a lookup service. Proxies may also be used as parameters or return values of remote methods, which is a feature the *Chronos Registry* (section 4.2) makes heavy use of. ICE comes with a set of additional tools and services useful for distributed application development. One that should be mentioned explicitly due to its importance in the current work is IceGrid. IceGrid is the location and activation service for ICE applications [13e]. On the one hand it may be used to resolve addresses and ports of servants via name lookup, on the other hand it is able to remotely start, stop and distribute executables on an arbitrary number of hosts. Additional features are replication and load balancing as well as status monitoring of registered components. To enable this functional range, each involved host has to run a local service or daemon called *IceGridNode*. The configuration of which executables to start, where to find them and where to deploy them is specified via XML files.

2.2. Programming Languages, Tools and Libraries

The practical implementation of this work makes use of several tools and libraries, which will be described in the following.

2.2.1. MATLAB/Simulink

MATLAB by Mathworks is a well-established prototyping environment for engineering tasks. Although there are free alternatives, MATLAB currently offers one of the most comprehensive and best supported overall packages for digital signal processing. In principle it consists of a powerful high-level scripting language optimized for linear algebra and numerical calculations and an extensive library of predefined functions, partitioned into various ”toolboxes” for a wide range of different applications.

The advantage of this kind of prototyping environment is the flexibility it offers during algorithm design. Algorithms and data can easily be inspected step by step during runtime, visualized, imported and exported. It is quite time-efficient to try different ideas, due to the huge set of predefined functions available. However, once an algorithm is ready for a first tryout in a larger system, it can often be difficult to integrate into a real-time environment. The algorithm has to be transformed into a variant capable of real-time processing and of being integrated into some target application. This usually means reimplementation in a lower level language like C and adaption to the requirements of on-line processing. Not only is this step error-prone, but it also requires a lot of development time and most likely an additional, specialized programmer.

Simulink is an add-on product to MATLAB, extending its functionality by a graphical programming language tool for modeling and simulation. It offers tight integration with the MATLAB environment and may be scripted from it. Simulink is widely used in control theory and digital signal processing for multidomain simulation and Model-Based Design [13k]. The models are inherently suitable for real-time operation, given sufficiently powerful processing hardware. However, to make use of multithreading from within Simulink, the additionally available *parallel computing toolbox* is required. Attention has been paid in this work, to rely on as few required toolboxes as possible. A basic MATLAB/Simulink installation plus Simulink Coder (former Realtime-Workshop) suffice to make full use of the developed extensions. However, the signal processing toolbox is highly recommended and is employed in some of the examples presented later.

The decisive benefit of using Simulink in combination with Simulink Coder is the possibility of generating C/C++ code directly from prototyped models. This meta-compilation process may be influenced by developing a custom ”target”, which has been done for *Chronos Realtime* (see chapter 3). Vice versa it is possible to integrate custom C/C++ code into Simulink via the "s-function" API. This enables integration of third party libraries into the prototyping environment and finally into generated code (see section 3.2 for an example). Signal processing models developed in Simulink are inherently on-line capable due to their fundamental architecture. This makes them ideal for integration into *Chronos Realtime* as building blocks for more complex applications.

2.2.2. XML/XSD

The word XML serves as an abbreviation for the E**x**tensible **M**arkup **L**anuage, which is used for describing data in a structured form. The design intent behind XML is that the format should be both human- and machine-readable. Therefore it perfectly lends itself as a means for writing configuration files. To further enhance the usefulness of XML in that respect, so called XML schema languages have been defined. These languages allow their users to express a set of rules to which an XML document should conform. In other words, they enable an application developer to specify what a syntactically correct XML configuration file for a particular application has to look like. The created specification aids a user in writing correct configuration files by enabling her to automatically validate and check for schema conformity using an XML editor. One of the most well-known XML schema languages is XSD (**X**ML **S**chema **D**efinition). Within *Chronos Realtime* all necessary configuration files for assembling an application are kept in XML where corresponding XSD files are supplied. This is true for the *Data Module* specific *Module Configuration* and *Channel Configuration* files (sections 4.2.3 and 4.2.4) as well as for the configuration of ZeroC's IceGrid applications.

2.2.3. C++

C++ is an object-oriented general-purpose programming language, originally developed as an enhancement to C, beginning in 1979 at Bell Labs. It is a very feature rich language, offering a lot of flexibility. Compilers are available for a wide variety of hard- and software platforms. Although a *Chronos Realtime* application may be extended by modules written in any language supported by ICE, the core components have been implemented using C++. According to [HS11], out of all supported languages ICE offers the best performance in combination with C++. As compilers have been slowly adapting the new features of C++11 in the course of this work, some of the older components do not make use of these features and implement techniques that could be written more elegantly by today's standards. The newer or refactored components however already depend on some additions of the new language standard and therefore require a C++11 compliant compiler.

2.2.4. Boost

Boost is the name of a comprehensive publicly available set of modern C++ libraries, designed to augment the C++ standard libraries with an additional range of high quality content. Boost plays a major role in the standardization of C++ today, as those parts that have stood the test of time stand a good chance of being integrated into the C++ standard libraries in a consistent or slightly modified way. Boost therefore offers a sensible and compatible way to integrate modern language features into a C++ program, even if the employed compiler or standard libraries do not natively support them at the time. *Chronos Realtime* makes extensive use of Boost throughout the entire range of its C++ source code.

2.2.5. SQLite

A *Chronos* application may assume different states, depending on currently running modules and active streams between these modules. This state may either be conditioned exactly as intended by the user, reside in some form of transition or assume an error state (e.g. due to a network or software failure). It is controlled and supervised by the *Chronos Registry* (see section 4.2). To enable a robust operation, it is advisable to not only keep the current application state in memory, but to also make it persistent by employing a database, accessing a persistent storage device. This way, even in the event of a blackout or *Registry* crash, the application may automatically continue its work after a restart or reboot. SQLite is a C library implementing a serverless SQL (**S**tructured **Q**uery **L**anguage) database engine. It is well-suited for the application at hand, as it is self-contained and may be integrated directly into the *Chronos Registry* executable. The library is well-established and offers good and stable performance for smaller databases. It is managed via standard SQL statements and may therefore be replaced by an external relational database system without too many modifications. SQL is a special-purpose programming language designed for managing data in relational database management systems. SQLite has been chosen for simplicity and offers sufficient performance for current applications. It may be replaced by a different database system in a future version of the *Registry*.

2.2.6. Portaudio

As the main focus of this work lies on audio processing, the functionality to capture and playback audio via corresponding hardware has been integrated into the core of *Chronos Realtime* (see sections 5.1 and 5.2). To achieve this goal in a potentially portable way, the cross-platform C library portaudio [13h] has been used. It supports audio recording and playback on most common operating system, utilizing several backends, including jack, ALSA and ASIO. To enable ASIO support the library had to be compiled in combination with the ASIO software development kit available from Steinberg, an important step in achieving low-latency audio using a Windows operating system.

2.2.7. PerClass

PerClass is a commercial software package available for Windows, MacOS and Linux, allowing for rapid development of custom machine learning and pattern recognition solutions. It has its origins in the PRTools toolbox for MATLAB, developed at the Delft University of Technology. As such there is still a certain level of similarity and compatibility between the two. PerClass likewise consists of a MATLAB toolbox for prototyping purposes, but also enables the user to compile a complex classification pipeline (consisting of e.g. preprocessing, dimensionality reduction and various classifier hierarchies) into a single file that may be addressed from a C/C++ application. To this end PerClass ships with a C runtime library that allows for execution of trained classifiers outside of the MATLAB environment. To close the circle, for *Chronos Realtime* a Simulink block has been developed (see section 3.2), able to execute PerClass pipelines during simulation as well as from programs generated via Simulink's code generation process.

2.2.8. Qt

Qt is a cross-platform, open-source C++ framework for creating GUI applications. It first became publicly available in May 1995 [BS06] and has been in development since. It offers a very comprehensive set of tools for GUI development in C++ and has therefore been used to implement the *Controller GUI* of *Chronos Realtime* (see section 5.5).

2.2.9. CppUnit

CppUnit is a unit testing framework for C++ code [13c] inspired by the original JUnit framework for Java. It has been employed to test individual code fragments for correct execution. This is particularly useful when compiling for different platforms, since certain errors might not be visible on the platform used for development. Furthermore it is highly recommendable to devise unit tests for methods making use of third party interfaces, since these might be subject to change or show otherwise unexpected behavior.

2.2.10. CMake

CMake [14] is a cross-platform build system for C++. It provides a simple language used to control the software compilation process. From the more or less generic CMake configuration files, native makefiles and workspaces may be generated to enable different tool chains to compile the entire *Chronos Realtime* framework. This procedure severely eases the expenditure of work required to maintain cross-platform support over the course of development.

3. From MATLAB/Simulink to Chronos Realtime

This chapter will introduce the Simulink part of the framework. Simulink offers a tight integration with MATLAB, which makes it quite suitable for transforming an offline MATLAB algorithm into a real-time-enabled model. Prototyping an algorithm from scratch however may also quite conveniently be done directly using Simulink. To support a simple integration of Simulink models into a *Chronos Realtime* application, the basic functionality of Simulink has been extended as a part of this thesis. This was achieved by implementing a so called ”real-time workshop target” (see section 3.1). In addition, a PerClass Simulink block has been developed (see section 3.2), allowing for a quick realization of pattern recognition tasks.

3.1. Chronos DLL Target

As mentioned in the introductory section 2.2.1, it is possible to generate C code from a Simulink Model, and to compile this code into a standalone executable, using the *Simulink-Coder* add-on (formerly part of the *Real-Time Workshop*). There are several different target platforms available by default, one of which may be selected by choosing a specific *system target file*. *Target files* influence how the generated code will be structured. This enables the code generation process to be adapted to different environments. If full integration of a compiled model into a bigger application is desired, the shipped system target files are only of limited use. Fortunately it is possible to implement custom *target files* by hand, allowing for great flexibility. Models may consist of any standard Simulink blocks and custom blocks (written in either C/C++ or the embedded MATLAB language). The newly developed *target* does not impose any additional restrictions on the code generation process compared to the default ones shipped with Simulink-Coder.

Figure 3.1 illustrates the procedure of how an executable is created from a Simulink model using custom *target files*. The template makefile, as well as the model code have to be supplied for implementation of a new system target. Specialized versions of these files have been developed as a part of this work to enable a straightforward integration of Simulink models into *Chronos Realtime* applications. Instead of generating a stand-alone executable, the approach that has been taken to tackle the problem

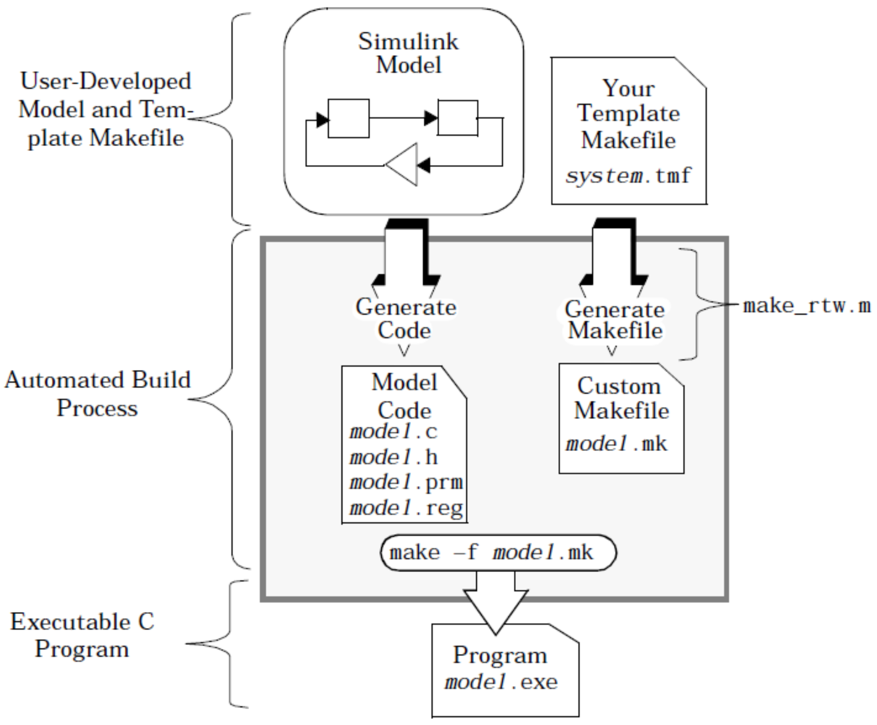


Figure 3.1.: Simulink Code Generation Process [The99]

An executable is compiled from a Simulink Model using instructions from a custom *Template Makefile* and automatically generated source code files of integratability, was to create a shared C++ library providing an API that would allow for supplying external data to the model, reading data from the model as well as for adjusting runtime parameters. This approach not only opens the possibility of integrating the model into a frame application, but also of exchanging models during runtime. The *target* has been termed *Chronos Realtime DLL Target*, where DLL stands for dynamic-link library. Typically this is the name associated with Microsoft's implementation of shared libraries, however, care has been taken to maintain a compatibility with all platforms supported by Simulink (Windows, MacOS, Linux). The terms *shared library* and *dll* will be used interchangeably in the following, as they refer to the same basic concept. The library resulting from the build process is named according to the convention of the given platform. The easiest way to understand the workflow involved in creating a *Chronos* shared library from Simulink is by following a simple example, as shown in section 3.1.1. Furthermore, the API of the generated library and how it may be used from C++ is described in section 3.1.2. Note however, that for the integration of a compiled model into a *Chronos Realtime* application, no programming is necessary, as there already exists a generic and ready-to-use *Chronos Realtime* module for this task (see section 5.3).

3.1.1. Example: Creating a shared library from a Simulink Model

A simple example to illustrate the process of compiling a Simulink model into a Chronos shared library shall be given. It assumes for the *Chronos Realtime* framework to be set up correctly within the MATLAB environment (instructions on how these practical steps are carried out are included in the source code package and are not part of this document). The demonstration model *GainPan16* (figure 3.4) is going to be used throughout this thesis to clarify different concepts. The function of the model may be described as receiving a stream of 16-bit stereo audio data and applying to it a certain gain and balance configuration, similar to a stereo channel of a simple digital mixer. The first question that arises is how to get the data into and out of the model, or put differently, how to define the boundaries between the compiled model and the application driving it. To this end a custom "blockset"[1] has been developed, providing a source and a sink for use with the *Chronos Realtime DLL Target* (see fig. 3.2).

When used in a standalone model executed from within the Simulink environment, the behavior of these two blocks is not particularly meaningful. The source copies any data received at the input to its output, thus acting as a bypass (sometimes useful for testing purposes). If the input is kept open, the output will contain all zeros. The sink simply discards all received data in a standalone model. However, when compiled into a *Chronos Realtime DLL Target*, the two blocks serve as an interface between a C++ application and the Simulink model, as defined in the library API (see section 3.1.2). To generate an operational model, Simulink needs to know the exact signal properties

[1]The term *blockset* in Simulink refers to a collection of semantically related building blocks

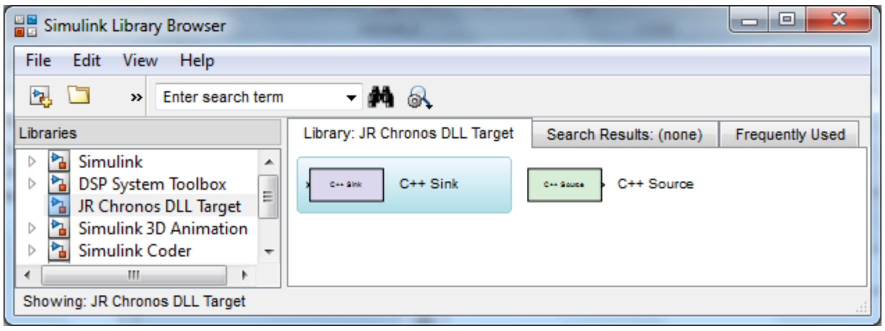
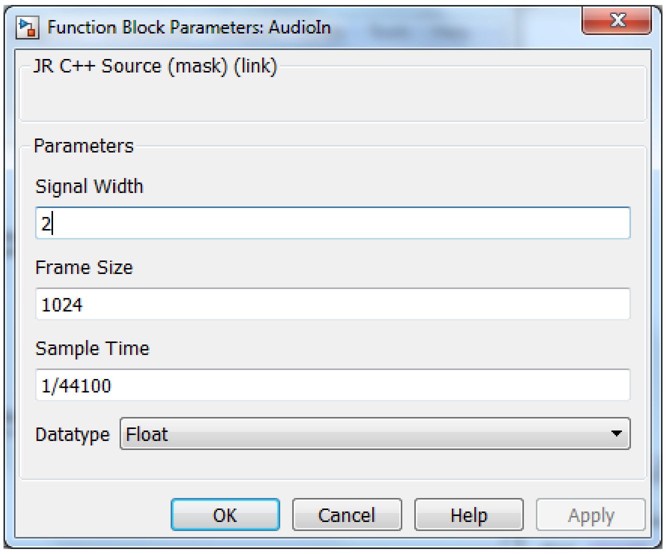


Figure 3.2.: Chronos DLL Blockset

Two custom Simulink blocks have been implemented to serve as an interface to the C++ API

of the input signal. These may be specified separately for each C++ *Source* via an input mask (shown in figure 3.3). The parameters of a C++ *Sink* are derived automatically and do not need to be specified. Using the described custom input and output blocks,


Function Block Parameters: AudioIn

JR C++ Source (mask) (link)

Parameters

Signal Width: 2

Frame Size: 1024

Sample Time: 1/44100

Datatype: Float

Figure 3.3.: C++ *Source* Input Mask

By double-clicking the block icon, its parameters may be configured

it is possible to proceed by building the desired algorithmic content as with any other model. The complete example is shown in fig. 3.4. The 16-bit audio input coming from the C++ *Source* **AudioIn** is first converted to double precision. Then a gain is applied to both channels, which in the shown figure is set to 1. The scaled signal enters the **Panorama** block, which is controlled by the **pPan** parameter. The **Panorama** block is a subsystem defined such that depending on the value of its second inport, the balance between the two channels of the first inport may be controlled. The meaningful range of values for **pPan** is defined to be between +1.0 (first channel only) and -1.0

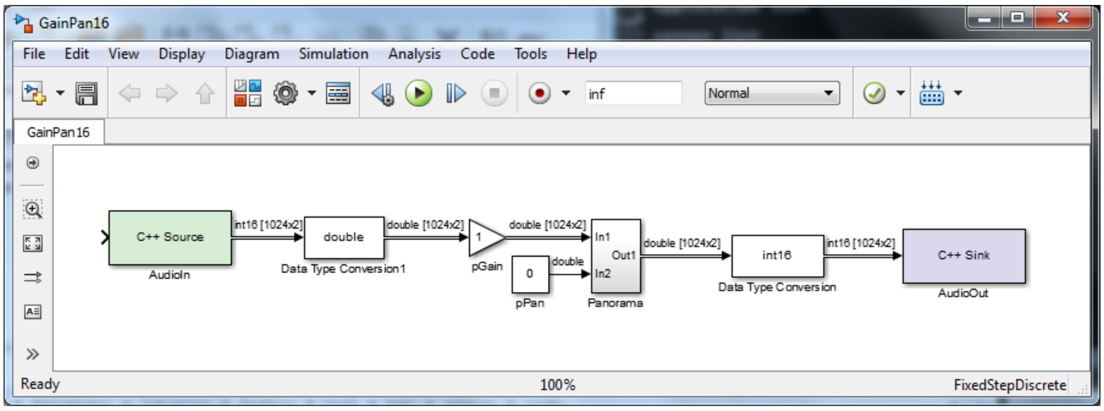


Figure 3.4.: *GainPan16* Simulink Model

A simple example to illustrate the described process

(second channel only). Finally the output of the **Panorama** block is converted back to a 16-bit integer signal and forwarded into a C++ *Sink* named **AudioOut**. Although non-essential for understanding the presented material, the internals of the **Panorama** subsystem are depicted in figure 3.5 for completeness.

Once a model with desired properties has been created and tested in the Simulink environment, the build procedure may be initiated. To this end a couple of settings in the configuration parameters dialog of the model have to be adjusted. For the purpose of real-time digital signal processing, the solver should be set to fixed-step, as a variable step-size can not be mapped to a real-time clock. In the code generation tab, *jr_dll.tlc* has to be chosen as system target file and C++ as language. Figures 3.6 and 3.7 illustrate what the settings dialog should look like.

If everything is set up correctly, the build process may be launched. As a result, a shared library (e.g. GainPan16.dll or GainPan16.so) will be created and stored in the current working directory. The generated library may then be loaded into a *Simulink Module*, one of the framework's core components (see section 5.3), which serves as a proxy for integration into a more elaborate *Chronos Realtime* application. The *Simulink Module* is able to process arbitrary *Chronos Streams* (section 4.4) using the generated file. For clarification, the whole workflow of Simulink integration is once more summarized visually (see fig. 3.8).

A library generated by the described process however is not bound exclusively to being embedded into a *Simulink Module*. It may similarly be loaded from a custom stand-alone application. Section 3.1.2 describes how to interface the library from C++ directly, using its exported API.

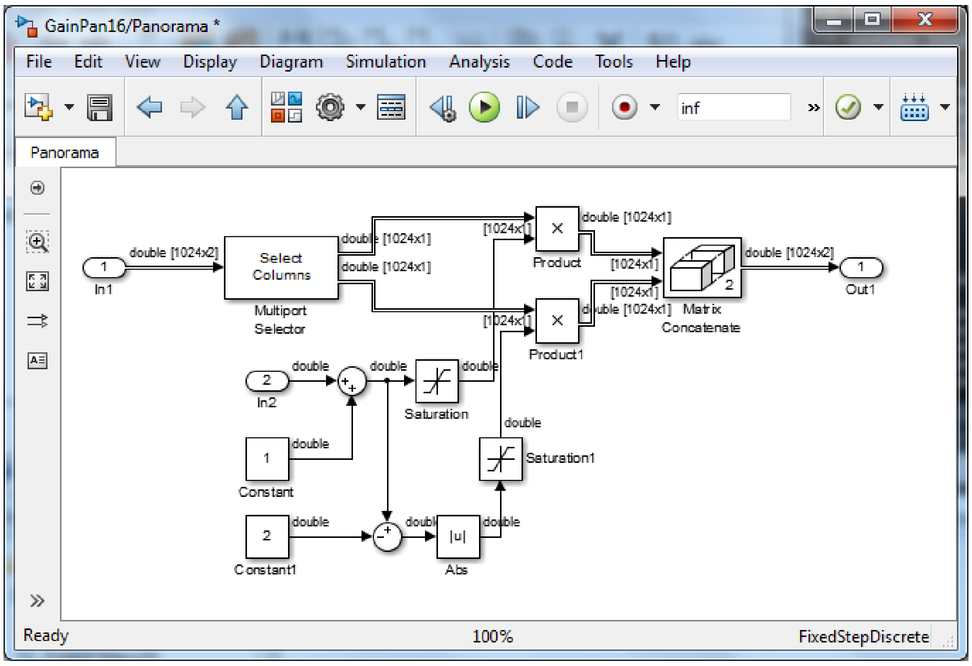
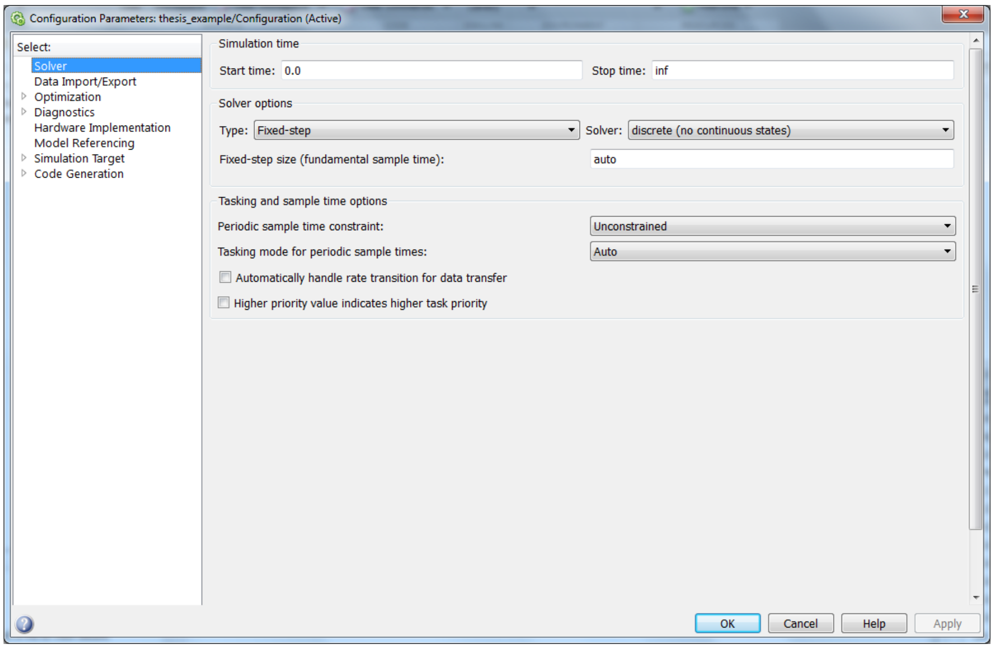


Figure 3.5.: Panorama Subsystem


Configuration Parameters: thesis_example/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation

Simulation time

Start time: 0.0 Stop time: inf

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Fixed-step size (fundamental sample time): auto

Tasking and sample time options

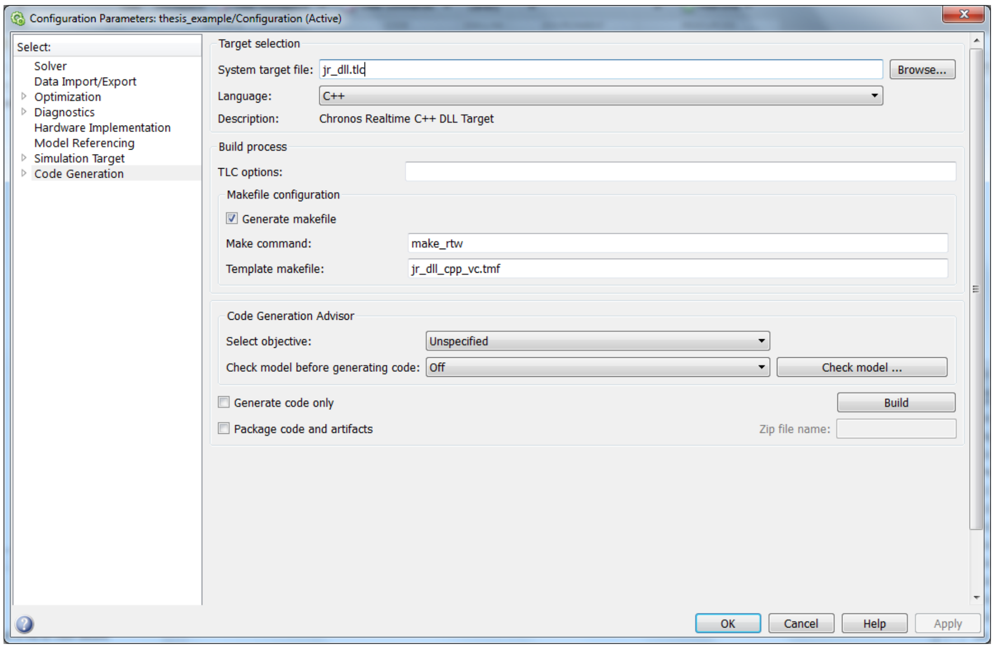
Periodic sample time constraint: Unconstrained

Tasking mode for periodic sample times: Auto

☐ Automatically handle rate transition for data transfer

☐ Higher priority value indicates higher task priority

Figure 3.6.: Configuration Parameters - Solver Tab


Configuration Parameters: thesis_example/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation

Target selection

System target file: jr_dll.tlc

Language: C++

Description: Chronos Realtime C++ DLL Target

Build process

TLC options:

Makefile configuration

[✓] Generate makefile

Make command: make_rtw

Template makefile: jr_dll_cpp_vc.tmf

Code Generation Advisor

Select objective: Unspecified

Check model before generating code: Off

[ ] Generate code only

[ ] Package code and artifacts

Zip file name:

Figure 3.7.: Configuration Parameters - Code Generation Tab

3.1.2. C++ API

This section will give a little more insight on the way the Chronos DLL target is implemented and how to interface with a generated dll. Care has been taken to make integration into a C++ project as straightforward as possible. Only the model-independent header file "simulink_model.h" has to be included to access the API. No additional linking or any model-specific files are required for compilation of the frame application. The generated dll exports an object oriented C++ interface. Two classes of particular importance are `SimulinkModel` and its template subclass `SimulinkModel::Port`. Their class diagrams are depicted in fig. 3.9 and 3.10 respectively and may be referred to as an aid in understanding the remainder of this section.

The `SimulinkModel` class defines a static creator method `CreateModelInstance`, used for instantiation of a model from a particular dll, specified by its filename. It is possible to create multiple instances of the same model, or several different models within one application. Once constructed, a `SimulinkModel` object provides all methods necessary to interact with the underlying algorithm.

Every C++ *Source* and C++ *Sink* present in the original Simulink model becomes a `SimulinkModel::Port` in its C++ counterpart and may be referred to by its path and name. For example, the C++ *Source* shown in fig. 3.4 is addressed by the name '<Root>/AudioIn'. <Root> refers to the lowest layer of the model. Every subsys-

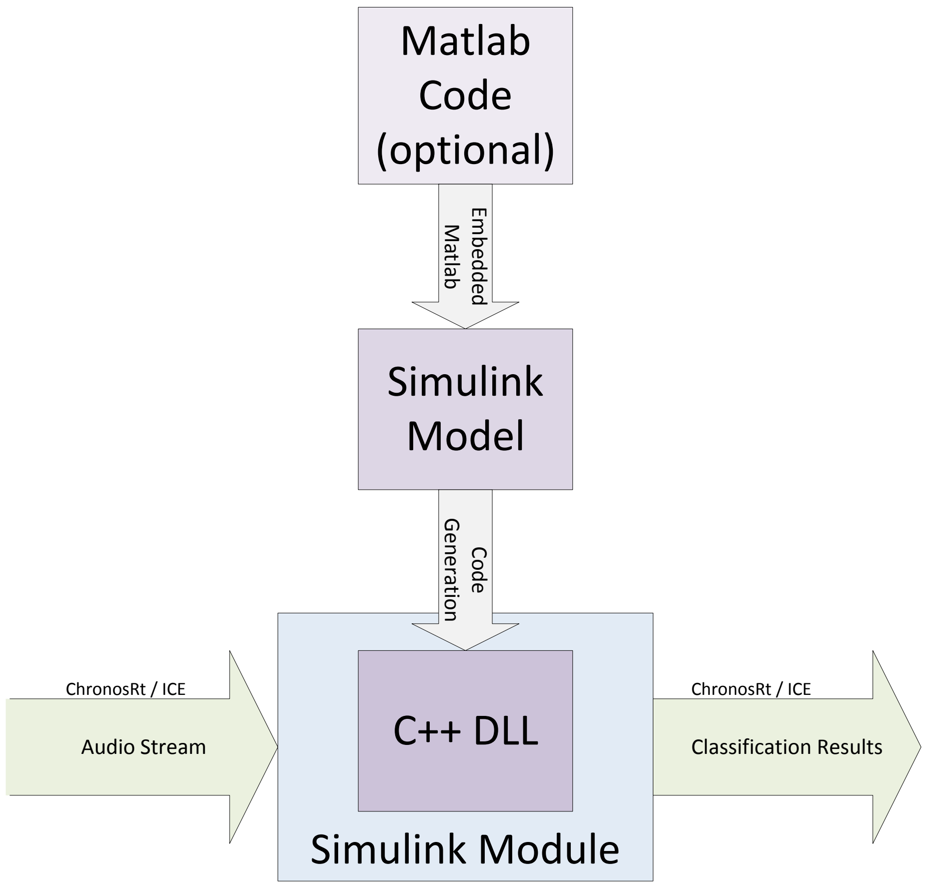
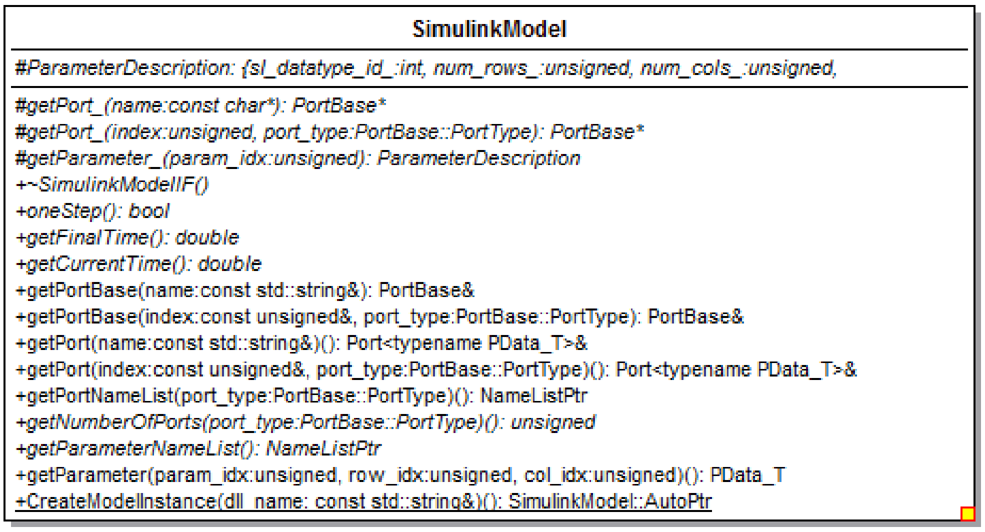


Figure 3.8.: From Matlab/Simulink to Chronos Realtime
To integrate a Simulink Model into Chronos Realtime it has to be compiled into a DLL and loaded into an instance of a *Simulink Module* executable. Matlab code may be integrated into the model if it complies with the Embedded Matlab language subset.



```
SimulinkModel
#ParameterDescription: {sl_datatype_id_:int, num_rows_:unsigned, num_cols_:unsigned,
#getPort_(name:const char*): PortBase*
#getPort_(index:unsigned, port_type:PortBase::PortType): PortBase*
#getParameter_(param_idx:unsigned): ParameterDescription
+~SimulinkModelIF()
+oneStep(): bool
+getFinalTime(): double
+getCurrentTime(): double
+getPortBase(name:const std::string&): PortBase&
+getPortBase(index:const unsigned&, port_type:PortBase::PortType): PortBase&
+getPort(name:const std::string&)(): Port<typename PData_T>&
+getPort(index:const unsigned&, port_type:PortBase::PortType)(): Port<typename PData_T>&
+getPortNameList(port_type:PortBase::PortType)(): NameListPtr
+getNumberOfPorts(port_type:PortBase::PortType)(): unsigned
+getParameterNameList(): NameListPtr
+getParameter(param_idx:unsigned, row_idx:unsigned, col_idx:unsigned)(): PData_T
+CreateModelInstance(dll_name: const std::string&)(): SimulinkModel::AutoPtr
```

Figure 3.9.: SimulinkModel Class Interface

tem adds one hierarchical layer. If, hypothetically, the mentioned 'C++ Source' was located in the 'Panorama' subsystem instead, it would be addressed by the path '<Root>/Panorama/AudioIn'. Specification of the full path is necessary, since the name of a block in Simulink doesn't need to be unique within the whole model, but only on the same layer - similar to how most modern filesystems work. When unsure about the path or name of a particular port, `getPortNameList` may be used to find the names of all available ports. Once a `SimulinkModel::Port` has been retrieved, it may be used to get data into or out of the model, depending on its `PortType`. Note that a `SimulinkModel` object does not hold a thread of its own, but needs to be driven externally by user code. Every time `SimulinkModel::oneStep` is called, simulation time of the model moves forward by its smallest internal sample time step. It is important to keep in mind that in a multi-rate system, not every in- and outport will be updated at every sample instant. For this reason the `Port` class offers a method `PortBase::isUpdated` to query for the availability of new input data at an input port, or new output data at an output port. By utilizing these methods an unknown Simulink model may be handled quite conveniently and generically. Note that for an input port to the model (`PortBase::SL_IN`), the next frame of data has to be written to the port as soon as `PortBase::isUpdated` returns false. Data has to be written using the `PData_T*` returned by `Port<PData_T>::getDataToWrite`. On the other hand, for an output port of the model (`PortBase::SL_OUT`), the next frame of data should be read as soon as `PortBase::isUpdated` returns true. Data has to be read using the `PData_T*` returned by `Port<PData_T>::getDataToRead`. In principle model ports could also be dealt with according to their respective sample times, instead of calling `PortBase::isUpdated` every time step. However, this approach would make a correct execution of a general model a lot more involved.

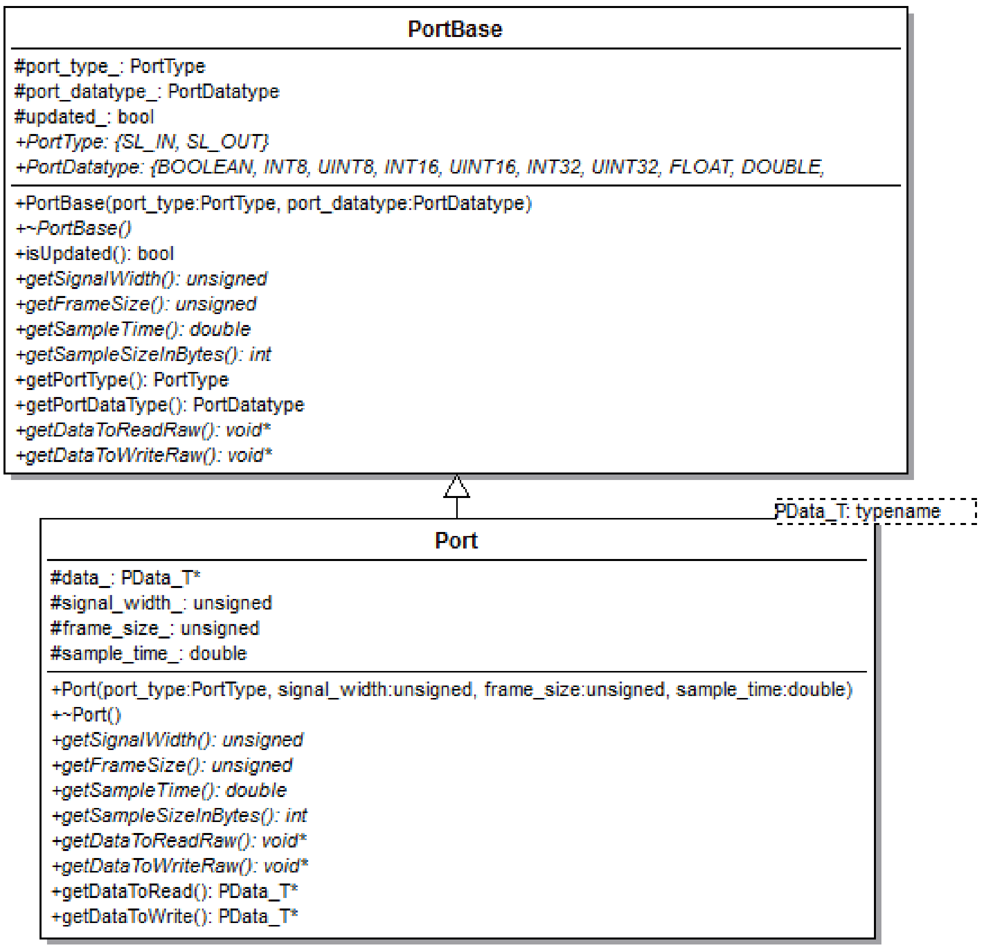

PortBase
#port_type_: PortType
#port_datatype_: PortDatatype
#updated_: bool
+PortType: {SL_IN, SL_OUT}
+PortDatatype: {BOOLEAN, INT8, UINT8, INT16, UINT16, INT32, UINT32, FLOAT, DOUBLE,
+PortBase(port_type:PortType, port_datatype:PortDatatype)
+~PortBase()
+isUpdated(): bool
+getSignalWidth(): unsigned
+getFrameSize(): unsigned
+getSampleTime(): double
+getSampleSizeInBytes(): int
+getPortType(): PortType
+getPortDataType(): PortDatatype
+getDataToReadRaw(): void*
+getDataToWriteRaw(): void*
PData_T: typename
Port
#data_: PData_T*
#signal_width_: unsigned
#frame_size_: unsigned
#sample_time_: double
+Port(port_type:PortType, signal_width:unsigned, frame_size:unsigned, sample_time:double)
+~Port()
+getSignalWidth(): unsigned
+getFrameSize(): unsigned
+getSampleTime(): double
+getSampleSizeInBytes(): int
+getDataToReadRaw(): void*
+getDataToWriteRaw(): void*
+getDataToRead(): PData_T*
+getDataToWrite(): PData_T*

Figure 3.10.: SimulinkModel::Port Class Interface

Listing 3.1 shows a simple example of how the *GainPan16* model, developed in section 3.1.1 could be driven directly from a C++ application. Although fully functional, note that for a general multi-rate system, having multiple input and output ports, this loop would need to be a lot more intricate, requiring an appropriate flow-control algorithm. However, the basic principles for interfacing the model remain the same in all cases.

Listing 3.1 Chronos DLL target C++ example

```cpp
#include <iostream>
#include <simulink_model.h>

int main(int argc, char *argv[])
{
  try
  {
    auto our_model = SimulinkModel::CreateModelInstance("GainPan16.dll");
    auto audio_in = our_model->getPort<short>("<Root>/AudioIn");
    auto audio_out = our_model->getPort<short>("<Root>/AudioOut");
    while(our_model->getCurrentTime() < our_model->getFinalTime())
    {
      std::vector<short> some_data = generateData<short>(audio_in.getFrameSize() *
          audio_in.getSignalWidth());
        // let's pretend the generateData function generates N input
        // samples, where N is specified by the first parameter, and the
        // sample datatype is given by the template argument
      std::copy(some_data.begin(), some_data.end(), audio_in.getDataToWrite());
        // since GainPan16 is a single-rate system
        // we can read and write data on every time step
        // otherwise we should only write data if !isUpdated()
        // after calling getDataToWrite(), isUpdated() will return true
        // until the written data has been consumed by the model

      if(!our_model->oneStep())
        break; // something went wrong

      std::copy(audio_out.getDataToRead(), audio_out.getDataToRead() + audio_out.
          getFrameSize() * audio_out.getSignalWidth(), some_data.begin()) // for
          convenience we just reuse the some_data vector
        // if it was a multi-rate system we should check if
        // the port isUpdated() before reading from it
    }
    // note that no cleanup is required, the dll and all
    // allocated memory are freed automagically
  }
  catch(const std::exception &exc)
  {
    std::cerr << "Exception: " << exc.what() << std::endl;
  }
  return 0;
}
```

One more feature of the API worth mentioning is the ability to access tunable parameters of the model. The method `getParameter` returns a reference to the parameter specified by the given index. Which parameter corresponds to which index may be found using `getParameterNameList`. The parameter reference is directly modifiable by use of the assignment operator. Any applied changes take effect on the

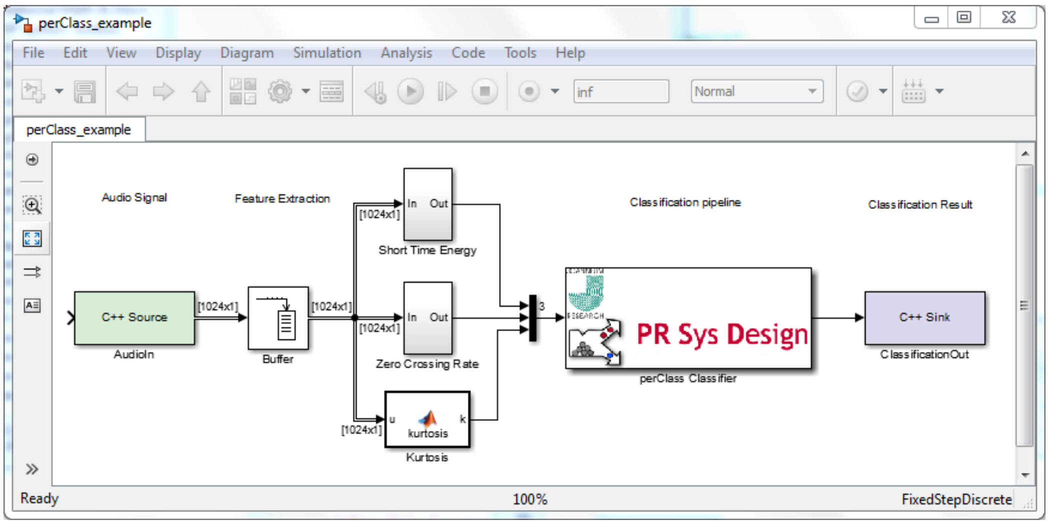


Figure 3.11.: PerClass Simulink Model

Example illustrating the use of the developed PerClass Simulink block

next call of `oneStep`. Since every parameter in Simulink may potentially be a matrix, `getParameter` may optionally be supplied with a row and column index, enabling a direct access to an arbitrary matrix element. Note that the indices in this case are zero-based, as common in C++.

3.2. PerClass Integration

PerClass is a toolbox used for classification and pattern recognition tasks (see section 2.2.7). A perClass pipeline, taking features as inputs and producing classification results on its outputs, may be developed and exported to a .ppl file using MATLAB. A custom Simulink block for loading and executing these .ppl files directly into a model has been implemented in the course of this thesis. From there the pipelines may easily be integrated into a Chronos Realtime application, by making use of the formerly described DLL target (see section 3.1). Figure 3.11 illustrates a simple, but typical instance of a model that may be used for a classification task within a *Chronos Realtime* application. To make the example more instructive, a simple feature extraction stage has been included (alternatively the features might already have been calculated in an external Chronos module). The incoming signal is reframed using a hop size as specified by the buffer parametrization. Three simple features are calculated and sent into the classification pipeline. To configure the perClass block, only the license path and the .ppl file to be used need to be specified. Finally the classification result is relayed via a C++ *Sink* (as explained in section 3.1.1).

4. Framework Application Architecture

This chapter will provide a detailed overview of the architecture and application flow of a *Chronos Realtime* application[1]. The currently implemented core components will be introduced and their role within the framework explained. Simplified UML-like (**U**nified **M**odeling **L**anguage) class- and sequence-diagrams will be used to summarize and exemplify the most important interfaces. It should be noted that these diagrams are used for documentation of existing code only and are not complete in the sense of a software architecture model.

4.1. Fundamental Concepts

To better understand how the different components play together, it is expedient to first gain an insight into underlying concepts. The general flow of a *Chronos Realtime* application may be characterized as follows:

- **acquire** input data
- **process** acquired data
- **provide** processed/transformed data or control signals

In most cases input data will be captured on-line from hardware sensors like microphones, accelerometers or video cameras. However, the data may similarly stem from a TCP stream or a file. The data entering the system is then processed as defined by the application engineer. For instance the data might run through a pattern recognition pipeline, providing classification results on its output. These results might either be written directly into a file or passed on to some sort of external interface. The processing step can however be made arbitrarily complex and potentially consist of several hierarchically structured stages. In *Chronos Realtime* data is generally handled as stream and referred to as *Data Stream* (section 4.4). Components that handle *Data Streams* in one way or the other are referred to as *Data Modules* (section 4.3). A *Data Module* supplying data streams to other modules is called *Data Provider*, a *Data Module* taking streams as input *Data Client*. A combination of both is referred to as *Data Transducer* (see fig. 4.1).

[1]It is important to note that the *Chronos Realtime* framework is under constant development. The current version (and the one described within this document) is 2.0

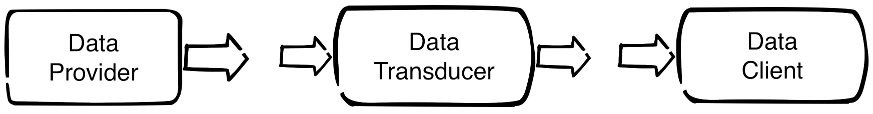


Figure 4.1.: Data Module Types

Generally a *Data Module* is an executable of its own, albeit it is be possible for one executable to contain several `DataModule` objects, if it appears sensible to do so. Each *Data Module* implements one specific functionality and may potentially be executed on a separate machine. To keep it general, a *Data Module* does not have any apriori knowledge about which other *Data Modules* it is connected to. In other words, a general *Data Module* will run idle after startup, effectively doing nothing until prompted to do otherwise. The component responsible for this task is the *Chronos Registry* (section 4.2). It is the only component invariably present in any *Chronos Realtime* application and is in charge of configuring *Data Modules* and establishing streams in between them. The *Registry* itself is configured by the user via schema based XML configuration files. It is also possible to add configurations during run-time via API calls. The subsequent sections will delve into this subject in more detail.

Since a *Chronos Realtime* application generally consists of several executables on potentially more than one machine it is conceivably useful to have a way of starting and stopping all of them via a single command. This may be achieved by setting up an IceGrid application (part of the ICE middleware, see section 2.1.2). An IceGrid application is configured through an XML file, specifying which executables to run on which machine. Each involved machine, also referred to as node, needs to host an IceGrid daemon with sufficient rights to launch respective executables (called *servers* within IceGrid). IceGrid may also be used to monitor the different nodes and states of each server.

Functionality of a *Chronos Realtime* application is generally not dependent on the time-frame or order in which its different components are started. It is therefore entirely possible to launch some or all of the involved executables by hand if desired. The design is robust towards component failures due to hard- or software crashes, in that a restarted machine or module may be reintegrated into a running system at any point in time. This is due to a *Data Module* never expecting the presence of any other module (except the *Registry*) and the inherent expectation that any remote method invocation might potentially fail at any point in time.

4.2. Chronos Registry

The *Chronos Registry* constitutes the heart of any *Chronos Realtime* application. It is responsible for configuring *Data Modules* and setting up streams in between them.

4.2.1. Overview

Since a *Chronos Realtime* application is generally composed of a number of generic modules, an authority is required to make them cooperate suitably for a specific task at hand. This is the *Chronos Registry*'s main purpose. The knowledge of how exactly the modules need to be interconnected has to be provided by the user or application engineer. Static configurations (streams that should be active from start-up to tear-down) may be read from configuration files on disk, dynamic configurations (on-demand streams) will generally be supplied to the Registry via its API. The format in which the *Registry* expects its configuration files is specified according to purpose-built XML schema definitions. Two different types of configuration files are to be distinguished:

- *Module Configurations* (section 4.2.3)
- *Channel Configurations* (section 4.2.4)

All configurations accepted by the *Registry* are stored in a database (section 4.2.5) to make them permanent. This way the desired application state may be restored on reboot, in case of a crash or power outage. *Data Modules* do not need to remember their states as they will be reconfigured by the Registry upon incidental restarts.

Every *Data Module* is required to register at the *Registry* after start-up, announcing its name and the ICE specific `DataModule`-proxy (which is containing network address). For this to be possible, the *Registry*'s location must be known beforehand. This is achieved by utilizing the lookup service provided by IceGrid. The *Chronos Registry* itself must be registered as a "well-known object" [13e] named *CXR* by convention. Every other server launched within IceGrid is then able to automatically retrieve the *Registry*'s proxy. A component started outside of the grid needs to be provided with the address of the IceGrid locator via command line parameter or configuration file, as it would otherwise not be able to resolve the *Registry*'s network address.

Once a *Data Module* has successfully registered, it will receive its configuration by the *Registry* via corresponding API calls. Each *Data Module* is identified by a unique name, which may be set via command line arguments when launching the respective executable. The supplied identifiers are used by the *Registry* to assign correct configurations to the different *Data Modules*. Note that the configurations held by the *Registry* are independent of hardware/network structure. In other words, it is irrelevant to the application, which machine a particular module is running on. Since modules are identified by name, the exact distribution of executables among different machines does not have to be laid out beforehand and may change dynamically during run-time.

Figure 4.2 exemplifies the process of *Data Module* registration. Communication is initiated by the *Data Module*, in this example a *Data Client*[2], by calling the *Registry* method

[2]the fragments demonstrated here equally apply to both, *Data Clients* and *Data Providers*

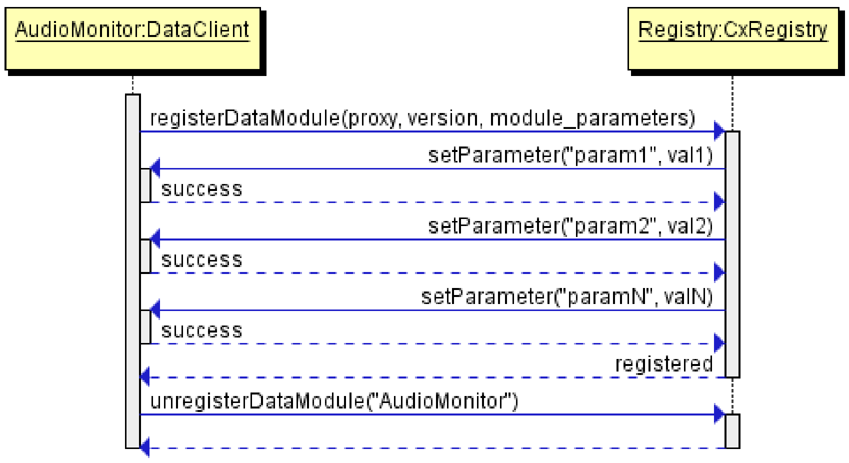

AudioMonitor:DataClient
Registry:CxRegistry
registerDataModule(proxy, version, module_parameters)
setParameter("param1", val1)
success
setParameter("param2", val2)
success
setParameter("paramN", valN)
success
registered
unregisterDataModule("AudioMonitor")

Figure 4.2.: *Data Module* Registration

The sequence diagram shows the basic process of a *Data Module* registering at and getting configured by the *Chronos Registry*

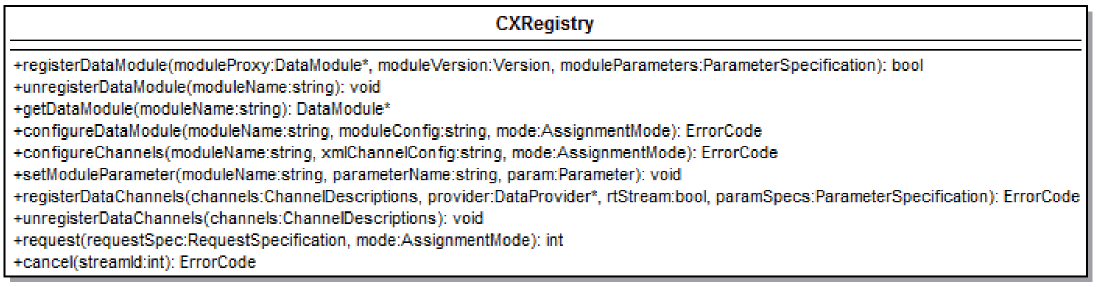
`registerDataModule`[3]. The *Registry* stores the given information in its database and looks for module configuration entries matching the provided identifier (which is embedded into the supplied proxy parameter). If the query is successful, the module is configured by an according number of calls to the `setParameter` method. A successful configuration completes the general registration process and the module is ready for operation. In this case *Registry* proceeds to check for any stream-related configuration assignments associated with the newly available module. These may trigger one ore several of three different actions:

- Ask the newly registered module to publish specific streams
- Tell the newly registered module to stream data to another, already registered module
- Tell an already registered module to stream data to the newly registered module

Section 4.4 will elaborate on the process of streaming in *Chronos Realtime*.

At some point, a registered *Data Module* might have to terminate. Before shutting down, the module is required to call `unregisterDataModule`, to facilitate a clean continuation of application flow. However, due to the fact that a variety of failures may occur at any point in a distributed system, the unregister call may become lost. It is therefore allowed to re-register a module with an identifier already present in the *Registry*'s database. In this case, the *Registry* assumes that the formerly registered module has become inoperative and carries out a complete deregistration before accepting the new module with the same identifier.

[3]methods and parameters applied here are described in more detail in sections 4.2.2 & 4.3



```
CXRegistry

+registerDataModule(moduleProxy:DataModule*, moduleVersion:Version, moduleParameters:ParameterSpecification): bool
+unregisterDataModule(moduleName:string): void
+getDataModule(moduleName:string): DataModule*
+configureDataModule(moduleName:string, moduleConfig:string, mode:AssignmentMode): ErrorCode
+configureChannels(moduleName:string, xmlChannelConfig:string, mode:AssignmentMode): ErrorCode
+setModuleParameter(moduleName:string, parameterName:string, param:Parameter): void
+registerDataChannels(channels:ChannelDescriptions, provider:DataProvider*, rtStream:bool, paramSpecs:ParameterSpecification): ErrorCode
+unregisterDataChannels(channels:ChannelDescriptions): void
+request(requestSpec:RequestSpecification, mode:AssignmentMode): int
+cancel(streamId:int): ErrorCode
```

Figure 4.3.: *Chronos Registry* ICE API

Due to the central role the *Registry* plays within a *Chronos* application, it is crucial to ensure a stable and responsive behavior. To this end, a set of worker threads is maintained internally, asynchronously handling each API call via a queuing system. This paradigm is also referred to as the worker-crew model [GSS02], which has been slightly extended to incorporate a finite state machine [HS01] for handling different execution states. On the one hand it allows for a fast response time and independent handling of parallel requests, on the other hand, resources are reserved ahead of time, preventing failures in times of heavy load (legitimate or due to a malicious attack). The maximum job queue length may be restricted so that excessive requests could potentially get rejected. This however is not a situation that should occur under normal operating conditions.

4.2.2. ICE API

All of the *Registry*'s API calls relevant to a *Data Module* developer are documented in figure 4.3. In the following they will be itemized and their purpose quickly summarized.

- `registerDataModule`: Invoked by *Data Modules* to register at startup.
  - **Parameters**:
    1. `moduleProxy`: Contains identifier and address of the module to register.
    2. `moduleVersion`: Contains version of the module to register. Allows the *Registry* to detect and handle outdated *Data Modules* accordingly.
    3. `moduleParameters`: Describes module specific parameters by name and type. Additionally a parameter may be marked as required or optional. As long as not all required parameters have been configured, the *Registry* will not permit any stream configurations involving this module.
- `unregisterDataModule`: Invoked by *Data Modules* to unregister on shutdown.
  - **Parameters**:
    1. `moduleName`: Identifier of the module to unregister.

- `getDataModule`: Used to retrieve the proxy of an already registered *Data Module* by name. Calling this method from other *Data Modules* should generally be avoided since it negatively impacts generality. However, in rare cases it might be necessary for an application specific module to make use of this feature.
  - **Parameters**:
    1. `moduleName`: Identifier of the module whose proxy should be retrieved.
- `configureDataModule`: Used to add a *Module Configuration* for the specified *Data Module*.
  - **Parameters**:
    1. `moduleName`: Identifies the module to configure.
    2. `moduleConfig`: XML string containing the desired *Module Configuration* (see section 4.2.3).
    3. `mode`: `AssignmentMode` for the given configuration. Possible modes are:
       * `IMMEDIATE`: Only use this configuration if the module is currently registered
       * `ONCE`: Make sure to apply this configuration either immediately or next time the module registers
       * `PERMANENT`: Keep this configuration permanently, irrespective of how often the concerned module registers and unregisters
- `configureChannels`: Used to add a *Channel Configuration* for the specified *Data Module*
  - **Parameters**:
    1. `moduleName`: Identifies the module to configure.
    2. `xmlChannelConfig`: XML string containing the desired *Channel Configuration* (see section 4.2.4).
    3. `mode`: `AssignmentMode` for the given configuration. Possible values are described under `configureDataModule`.
- `setModuleParameter`: Used to set a module parameter directly, instead of using an XML string.
  - **Parameters**:
    1. `moduleName`: Identifies the module to configure.
    2. `parameterName`: Name of the parameter to set.
    3. `param`: Value of the parameter to set. The appropriate `Parameter` subclass has to be chosen according to the its data type.
- `registerDataChannels`: Invoked by *Data Providers* to make their provided channels available for subscription.
  - **Parameters**:
    1. `channels`: Global identifiers of the channels to register.
    2. `provider`: Proxy of the *Data Provider*.
    3. `rtStream`: Indicates if the stream is subject to real-time constraints.

4. `paramSpecs`: Describes `RequestParameters` that are supported or required when subscribing to the specified channels.

- `unregisterDataChannels`: Allows a *Data Provider* to unregister formerly registered channels.
  - **Parameters**:
    1. `channels`: Global identifiers of the channels to unregister.
- `request`: Invoked to supply a *Data Client* with input streams.
  - **Parameters**:
    1. `requestSpec`: The `RequestSpecification` holds all parameters necessary for initialization of one or several data streams to a given `DataClient` interface.
    2. `mode`: `AssignmentMode` for the given request. Possible values are described under `configureDataModule`.
- `cancel`: Used to cancel one specific data stream.
  - **Parameters**:
    1. `streamId`: Identifier of the stream to be canceled. If the stream does not exist, the invocation will be ignored

4.2.3. Module Configuration

Some *Data Modules* may require configuration of certain parameters before being able to process data. For example, an *AudioCaptureUnit* (section 5.1) needs to know which audio device to capture from. The exact parameters required by one particular module are very specific to its type, but the principle of configuration can be generalized. An XML schema definition has been developed to enable a uniform approach to module configuration. The *Registry* is capable of parsing XML strings complying with this schema and storing the information in its database. If the respective module is registered at the time the *Registry* has extracted the relevant information, it will be configured immediately, otherwise configuration is part of the registration procedure.

Listing 4.1 Module Configuration example for an AudioCaptureUnit

```
<ModuleConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Parameter name="Device" xsi:type="String">ASIO Fireface USB</Parameter>
  <Parameter name="SampleRate" xsi:type="Float64">48000</SampleRate>
</ModuleConfig>
```

Listing 4.1 shows an example of a *Module Configuration* for an *AudioCaptureUnit*. As can be seen, the format is kept very straightforward. The XML file consists of a sequence of `Parameter` elements, enclosed by a `ModuleConfig` element. `Parameter` simply defines a key/value pair, where the "value" may assume one of several data types. The "key" is specified by the `name` attribute. The "value" corresponds to the text enclosed by the `Parameter` element and is interpreted according to the type defined by the `xsi:type` attribute. There are five different data types allowed in this context:

- **String** (Text)
- **Float32** (32-bit floating point)
- **Float64** (64-bit floating point)
- **Int32** (32-bit integer)
- **Int64** (64-bit integer)

Which parameters are allowed or even required is defined by the *Data Module* developer. Generally they should be known to a *Chronos* application engineer beforehand (e.g. by documentation), however, their specification is transmitted to the *Registry* during registration, so information about them may be queried from the database if necessary.

4.2.4. Channel Configuration

Everything related to streaming between different *Data Modules* is defined by the so called *Channel Configuration*. Similar to the *Module Configuration* an XML schema has been defined, sufficiently expressive to carry out the entire streaming setup in human readable form. It does however offer a more complex functional range and will therefore be presented in several, increasingly elaborate examples. When it comes to streaming, a clear distinction has to be made between *Data Providers* and *Data Clients*. The former ones are able to offer and dispatch data streams, the latter ones request and process them.

Initially, provider-specific configurations shall be illustrated, as these are responsible for making streams available to other modules in the first place. The question might come to mind as to why provided streams need configuration, since a provider could simply register all streams it is capable of supplying. However, apart from the fact that it often is not clear what exactly to provide even for a quite specific component, another important aspect is the need to assign a unique identifier to every subscribable stream. This has to be done by the application engineer, since identifiers cannot be auto-generated in a semantically sound way for an unknown application. In other words, it is necessary to tell a provider what exactly to provide and which names to use for the provided channels. This may be achieved using the `Provide` tags within a *Channel Configuration* XML string. Listing 4.2 shows a first, simple example for configuration of an *AudioCaptureUnit*.

Listing 4.2 Channel Configuration example-1

```
<ChannelConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Provide>
    <Channel>
      <GlobalID>Accel0_Z</GlobalID>
      <DataType>AUDIO</DataType>
      <LocalDescription>
        <Parameter name="Channel" xsi:type="Int32">0</Parameter>
      </LocalDescription>
    </Channel>
    <Channel>
      <GlobalID>Accel1_Z</GlobalID>
      <DataType>AUDIO</DataType>
      <LocalDescription>
        <Parameter name="Channel" xsi:type="Int32">1</Parameter>
      </LocalDescription>
    </Channel>
    <Channel>
      <GlobalID>Accel2_Z</GlobalID>
      <DataType>AUDIO</DataType>
      <LocalDescription>
        <Parameter name="Channel" xsi:type="Int32">2</Parameter>
      </LocalDescription>
    </Channel>
  </Provide>
</ChannelConfig>
```

The example describes the case where three tri-axial accelerometers [Gau02] are connected to an audio interface. The z-axes of the different sensors are connected to the physical channels 0, 1 and 2 of the audio interface. Every `Channel` defined in the configuration may be subscribed to separately. The `GlobalID` and `DataType` values characterize the channel globally within *Chronos Realtime*. No other channel is allowed to share the same `GlobalID/DataType` tuple. Valid enumeration values within the `DataType` element are:

- AUDIO
- VIDEO
- EVENT
- CONTROL
- CUSTOM

This distinction facilitates certain *Data Clients* in interpreting the incoming data correctly and similarly aid the user in interpreting the configuration file.

The `LocalDescription` element defines the local mapping of the channel. It is specific to the provider implementation and defined by a set of parameters. In the case of an *AudioCaptureUnit* it simply consists of a single integer parameter, corresponding to the channel number within the audio device driver. Other providers might need different local descriptions, possibly consisting of several parameters. As in the *Module Configuration* case, a parameter may take on one of the five data types:

- **String** (Text)
- **Float32** (32-bit floating point)
- **Float64** (64-bit floating point)
- **Int32** (32-bit integer)
- **Int64** (64-bit integer)

For large-scale applications it can become very tedious to itemize every single channel by hand. Most of the time, a high quantity of similar sensors are processed by a *Chronos Realtime* system, hence it seems natural to index them by numbers, as seen in the example at hand. To alleviate the problem of unmaintainably large configuration files, a short-cut method has been devised. A more compact version of the previous example, giving rise to the exact same outcome, is illustrated in listing 4.3. It should be noted, that not a single line needs to be added to provide an arbitrary number of like-wise channels.

Listing 4.3 Channel Configuration example-2

```
<ChannelConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <IndexDefinition name="chIdx">0:2</IndexDefinition>
  <Provide>
    <ForEach index="chIdx">
      <Channel>
        <GlobalID>Accel[chIdx]_Z</GlobalID>
        <DataType>AUDIO</DataType>
        <LocalDescription>
          <Parameter name="Channel" xsi:type="Int32" indexed-by="chIdx"/>
        </LocalDescription>
      </Channel>
    </ForEach>
  </Provide>
</ChannelConfig>
```

Two new element types have been introduced for this purpose, `IndexDefinition` and `ForEach`. These constructs play together to enable a short-cut method for channel enumeration. `IndexDefinition` allows the user to define a sequence of integer numbers in a MATLAB-like syntax. It may consist of white-space separated integers or sequence expressions as the one shown in the example listing. There is no limit to how many different indices may be specified within one configuration, but their names must be unique, as is enforced by the schema definition. Some more valid examples are shown in listing 4.4, all expanding to the same sequence of numbers. For more information on the colon-syntax, [Xen99] or a similar guide may be consulted.

Listing 4.4 IndexDefinition example

```
<IndexDefinition name="chIdx">0 1 2 4 6</IndexDefinition>
<IndexDefinition name="chIdx2">0:2 4 6</IndexDefinition>
<IndexDefinition name="chIdx3">0:1 2:2:6</IndexDefinition>
```

The concept of the `ForEach` element is borrowed from the well-known *for each* construct, common in many high-level programming languages. In the context of *Chronos Channel*

Configurations it is meant to read: For every element in the given index, repeat the channels enclosed by the `ForEach` block and replace designated expressions by the corresponding index value. In other words, the enclosed channels are repeated as many times, as the index has elements.

An expression of the form `[index-name]` within a `GlobalID` element will be expanded to the respective values of the specified index. However, only an index traversed by an enclosing `ForEach` (as specified by its `index` attribute) may be used, or the expression will be ignored. The same holds true for `Parameter` elements of type `String`. For example the following parameter could be specified within the `LocalDescription` of listing 4.3:

```
<Parameter name="PortName" xsi:type="String">AudioIn[chIdx]</Parameter>
```

As before, the [chIdx] expression is expanded accordingly. This differs from the way other parameter types incorporate the indexing mechanism, as can be seen in listing 4.3. Instead of the expression in square brackets, an `indexed-by` attribute is adopted. The reasons for this seeming inconsistency are twofold. One of them is schema validation. Would it be permissible to enter a string as a value of a numeric type, like:

```
<Parameter name="Channel" xsi:type="Int32">[chIdx]</Parameter>
```

it would not be possible to validate the data type correctly in case no indexing is used, making the configuration more prone to errors. Another reason is the ambiguity in interpretation of composite expressions. For example it is not immediately clear what an expression like:

```
<Parameter name="Channel" xsi:type="Int32">4[chIdx]6</Parameter>
```

should represent, as it is not natively interpreted as a string.

The functional range of described `ForEach` elements has further been extended by the possibility of nesting, which can be a useful in situations when multi-indexing is needed (A nested `ForEach` loop is illustrated in listing A.1). Although the concept is quite useful as presented, another feature has been implemented to make it more powerful. It is called *indirect indexing* and has been devised to cover the following use-case: Consider the tri-axial sensors to be connected to the audio interface in a different way this time. Instead of using the physical channels 0, 1 and 2, they are connected to 2, 5 and 8 (as the X- and Y-axes have been plugged in between). Therefore a second index needs to be defined, however, it needs to be addressed by the same `ForEach` block as the index `chIdx`. To express this relation, the symbol @ has been introduced as a way to chain an index expression to a specific `ForEach`. It is written `[index1@index2]` and can be read as 'insert the value that index1 takes at the position of the currently running index2 counter'. In other words, index1 and index2 are synchronously traversed element by element. Listing 4.5 displays the use of indirect indexing as described. Note that in this example `devIdx` must obviously contain at least as many elements as `chIdx`, otherwise a runtime error will occur. Semantic errors like this are beyond the scope of schema validation.

Listing 4.5 Channel Configuration example-3

```xml
<ChannelConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <IndexDefinition name="chIdx">0:2</IndexDefinition>
  <IndexDefinition name="devIdx">2:3:8</IndexDefinition>
  <Provide>
    <ForEach index="chIdx">
      <Channel>
        <GlobalID>Accel[chIdx]_Z</GlobalID>
        <DataType>AUDIO</DataType>
        <LocalDescription>
          <Parameter name="Channel" xsi:type="Int32" indexed-by="chIdx@devIdx"/>
        </LocalDescription>
      </Channel>
    </ForEach>
  </Provide>
</ChannelConfig>
```

This finishes the discussion on provider configuration and leads to examination of its counterpart, the request configuration. It is responsible for the actual establishment of data streams between two modules. The syntax is strongly related to the one introduced for providers, with a few extensions and modifications. A first example (listing 4.6) will demonstrate how a client would request the signal of the 0th acceleration sensor's z-axis, as it has been defined in the previous examples.

Listing 4.6 Channel Request example-1

```xml
<ChannelConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Request>
    <Channel>
      <GlobalID>Accel0_Z</GlobalID>
      <DataType>AUDIO</DataType>
      <ChannelSpecifics type="TimeSeries">
        <SampleRate>48000</SampleRate>
        <SampleFormat>INT16</SampleFormat>
        <FrameSize>1024</FrameSize>
        <StreamFormat>PCM</StreamFormat>
      </ChannelSpecifics>
      <ProcessingParameters>
        <Parameter name="Channel" xsi:type="Int32">0</Parameter>
      </ProcessingParameters>
    </Channel>
  </Request>
</ChannelConfig>
```

As before, the `GlobalID` and `DataType` elements are present to identify the channels, except they are enclosed by a `Request` element in this case. The *Registry* uses these tuples to look up the corresponding providers and to relay stream establishment to them. If the channels have not been registered, or a provider is unavailable at request time, the registry will store the request and prompt the provider to handle it as soon as it becomes available. Entirely new is the `ChannelSpecifics` element, used to describe what kind of stream the client would like to receive in more detail. The allowed child elements are dependent on the `type` attribute. Table 4.1 gives an overview of the currently supported combinations.

Table 4.1.: ChannelSpecifics and child elements currently allowed in an XML channel config

type	allowed child elements
TimeSeries	SampleRate SampleFormat (in Hz) FrameSize (in samples) StreamFormat
ImageSequence	Width (in pixels) Height (in pixels) MaxFrameSize (in bytes) FramesPerSecond Compression

The effect induced by these specifications is the guarantee that either the stream will comply exactly to the given requirements, or a `FormatNotSupported` exception will get thrown by the provider and an according error logged by the *Registry*. A provider might be capable of adapting to some or all of the specified attributes, but it is not obligated to. For example, an *AudioCaptureUnit* may only sample at one specific sample-rate at a time, which is specified in its *Module Configuration* and cannot serve streams at differing rates. The application engineer is responsible for ensuring a compatible overall configuration. However, in case of a configuration error, the logging mechanism of the framework will generally be able to pinpoint the source of failure quite accurately, to aid in solving potential problems.

Many providers are able to choose `ChannelSpecifics` for a stream automatically, so most of the time, they could be omitted from the *Channel Configuration*. Note however, that in doing so, the application is prone to unintentional behavior (e.g. due to a wrong assumption or due to changed default behavior induced by a software update) whose cause might be difficult to track down. Therefore it is recommendable to always fully specify the `ChannelSpecifics` for each channel.

For audio or other one-dimensional streams, `TimeSeries` specifics are used. The `SampleRate` and `FrameSize` elements are self-explaining. The `SampleFormat` expresses the datatype of the samples within a frame of data and might take on one of the following enumeration values:

- **INT8** (8-bit integer)
- **INT16** (16-bit integer)
- **INT24** (24-bit integer)

- **INT32** (32-bit integer)
- **FLOAT32** (32-bit floating point)
- **FLOAT64** (64-bit floating point)

Note that 24-bit integers are explicitly supported, since it is a very common resolution in professional audio applications and transmitting these streams in a 32-bit framing would be wasteful.

Finally, `StreamFormat` may contain the name of a codec, in case the stream does not conform to the default *Chronos* streaming format. This feature is intended for future extensions and has not been put to practical use yet. Similarly, the `Compression` element for image sequences specifies the format used to encode the images contained in the frames to stream.

This leads to the next major new element within the example listing, the `Processing-Parameters`. They can be thought of as being the counterpart to the formerly mentioned `LocalDescriptions`, but for use with *Data Clients*. Commonly they are used to map a channel to a client-specific resource. In listing 4.6, which is a configuration for an audio playback module, the *processing parameter* `Channel` is used to map the data of *Accelo_Z* to the 0th channel of the audio output device. However, the semantics of these *processing parameters* may go beyond just establishing a channel mapping. A client might define any number of allowed parameters, affecting the way the received stream is going to be processed.

As long as a stream is sufficiently characterized by its provided *Channel* specification, the elements discussed so far suffice to assemble any sort of streaming configuration between all registered *Data Modules*. However, there are situations when the `GlobalID` and `DataType` parameters are not sufficient to specify the exact content a stream should contain. This is frequently the case for *Data Providers* offering data not captured in an on-line manner, but rather data from a permanent storage or buffer. An example of this would be a file reader, providing audio from a number of WAVE files. The files might potentially be very large and a client might not want to receive all of the data, but only specific channels or time segments. To support this possibility the request specification has been extended by the `RequestParameters` element. This allows for an arbitrarily detailed characterization of what exactly should be supplied. A provider may support any number of request parameters, but in general it is recommendable to abstain from requiring any more than absolutely necessary, as it negatively affects generality.

A final example displaying a complete *Channel Configuration* XML file can be found in the appendix (listing A.1). It is meant to display all discussed features by example of a *Simulink Module Data Transducer* (see section 5.3). It has been constructed to give an overview and does not necessarily constitute the simplest possible or recommendable solution when it comes to application design. Specifically it would not be necessary to create a separate output port for every single band of the MFC. The corresponding simulink model addressed by the channel configuration is shown in figure 4.4. The

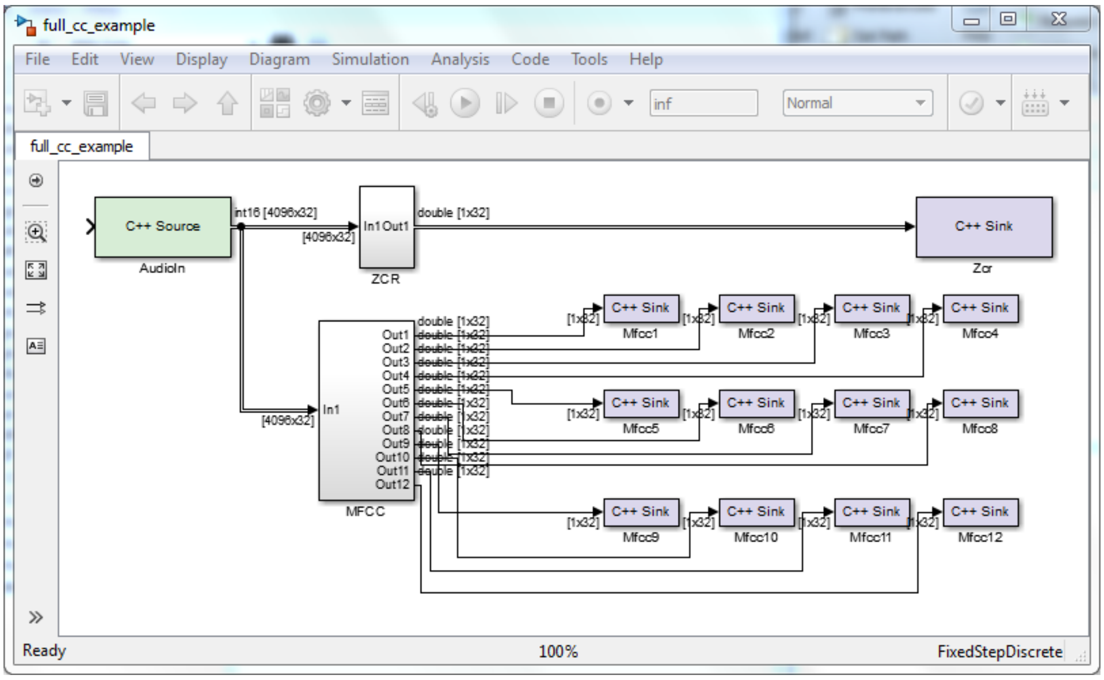


Figure 4.4.: Example of a Simulink Model used for Feature Extraction

scenario motivating this example may be described as follows: There are 16 stereo WAVE files provided by a *Chronos File Reader* (see section 5.6), so in total there are 32 single channels available. The audio data contained within these files are to be streamed to the aforementioned *Simulink Module* in real-time simulation mode (so the frames will arrive at a frequency dependent on the sample rate), which is particularly useful for demonstrations and testing. The model extracts 12 MFC coefficients and the zero-crossing rate of all 32 audio signals and makes these features available to other Chronos clients in the system. To get more insight on the exact meaning of the specific `LocalDescriptions`, `ProcessingParameters` and `RequestParameters` used in listing A.1 please refer to the corresponding *Core Data Module* descriptions in chapter 5.

4.2.5. Database Layout

The *Chronos Registry* uses a database backend to store information about all registered *Data Modules* and their respective configurations. This has been implemented to allow for a more stable operation of *Chronos* applications. The rationale behind this notion is that the *Registry* corresponds to the Achilles heel of the architecture. If any other component or even a whole server crashes, only the particular functionalities carried out by the affected modules go missing. As soon as they are back online, they will re-register and streams will be re-established. There are barely any other consequences, since the modules' states are administrated by the *Registry*. However, if the *Registry* itself or the computer it is running on crashes, no new streams may be established and no freshly started *Data Modules* may be configured during downtime. If the *Registry* kept track of *Data Modules* in memory only, it would not be able to locate the ones that have been registered before it went down, nor would it know if they had already been configured - the application state would be corrupt and the failure could only be recovered from with a full restart. On the other hand, when the state is stored on disk, the *Registry* is able to proceed with little consequence.

In its current implementation, an SQLite database has been chosen as a backend. It is a very lightweight, relational SQL database and shines in ease of use, since no additional server application is required. The whole functionality is implemented within the client and the storage consists of a simple file. The C interface has been encapsulated so as to make a potential switch to another database system unproblematic.

Figure 4.5 illustrates the devised database layout. As can be seen it consists of 7 tables representing the current state of configuration. The abbreviations **PK** and **FK** refer to the terms ***P**rimary **K**ey* and ***F**oreign **K**ey* respectively.

- **DataModules** (contains information about registered *Data Modules*)
- **NativeConfigurations** (contains XML module configurations)
- **ChannelConfigurations** (contains XML channel configurations)
- **ModuleParameters** (contains module parameters)
- **ParameterSpecs** (contains information about parameters as supplied by the registered *Data Modules*)
- **ParameterSets** (contains information about how parameters should be grouped together)
- **RegisteredChannels** (contains information about currently provided channels)

The `ModuleId` used as key in various tables is an internal identifier, created when a new *Data Module* registers. It is not to be confused with the `ModuleName`, which is the string a *Data Module* uses to identify itself. The essential difference between the two is that the `ModuleName` entries serve as an identifier for any *Data Module* that might register using this name, while the `ModuleId` refers to the one specific instance of that is currently registered. Therefore the `ModuleId` values in the database may only contain a valid value while a *Data Module* with the name `ModuleName` is actually registered.

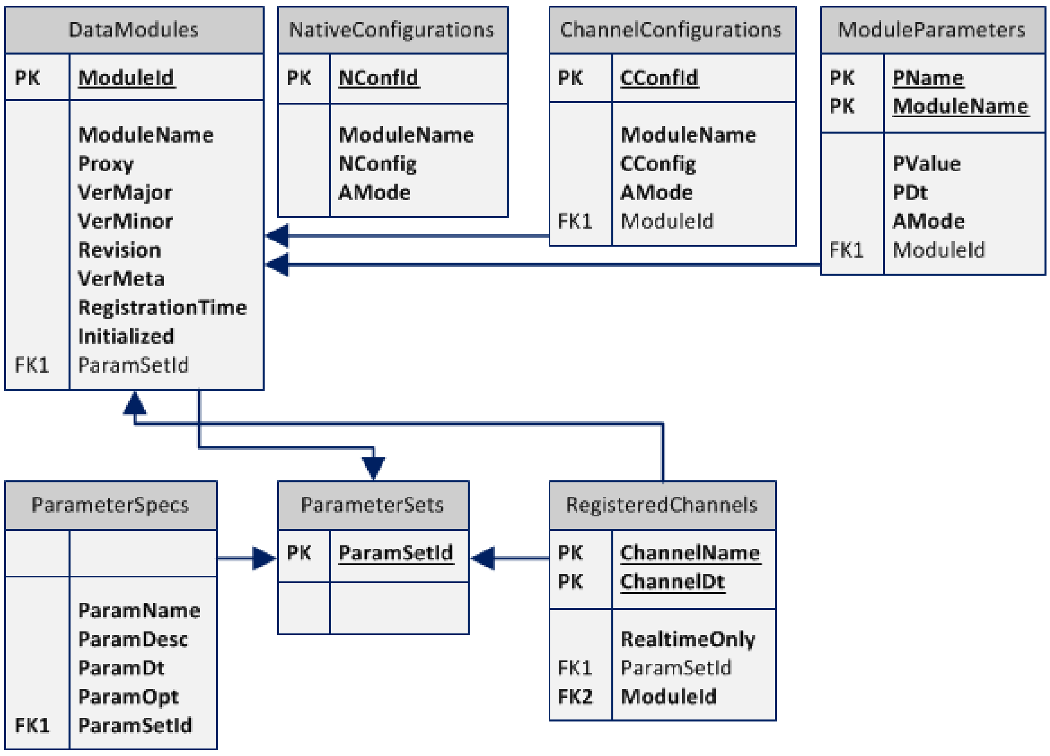


Figure 4.5.: Architecture of the SQL Database

Each box represents a table in the database, each entry a data element, the arrows represent relations between the different tables.

This mechanism simultaneously allows to ascertain that a particular configuration has already been assigned, as the *Registry* uses the `ModuleId` foreign keys to link to the corresponding primary key only after associated operations have been successful.

4.3. Data Modules

Besides the *Registry*, a *Chronos Realtime* application consists of an arbitrary number of *Data Modules*. These modules are responsible for the actual data streaming and processing procedures.

4.3.1. Overview

Data Modules are designed according to a variation of the publish/subscribe pattern [HW03], to allow for a decoupled connection establishment between individual mod-

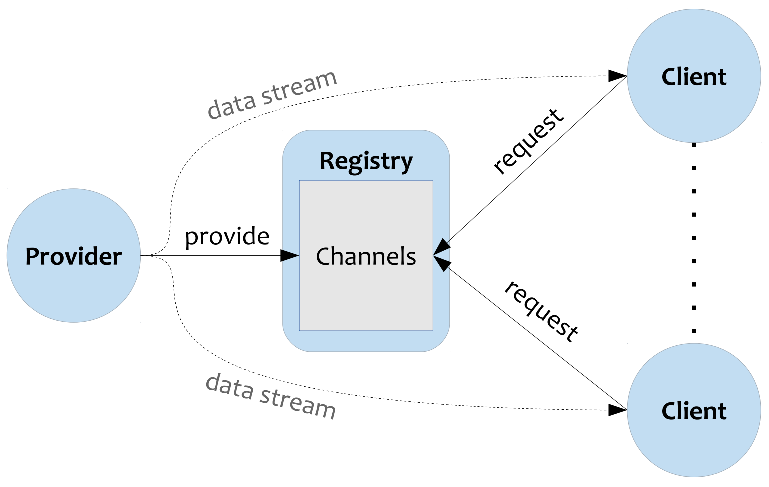


Figure 4.6.: Relation between *Registry*, *Providers* and *Clients*

Although the Registry is responsible for administration and establishment of the data streams, the streaming process itself is carried out peer-to-peer from *Providers* to *Clients*.

ules. This enables them to be implemented in an application-agnostic manner. The *Chronos Registry* plays the role of a mediator in this context.

Three different types of *Data Modules* can be distinguished:

- *Data Client*,
- *Data Provider*,
- and *Data Transducer*,

all of which implement a common `DataModule` interface (see fig. 4.7). The terminology has been changed with respect to the usual publish/subscribe wordings, as there are subtle semantic differences. In particular a stream in *Chronos Realtime* may be of finite duration and is regarded as completed after a successful transmission. It is therefore incoherent to speak of a subscription in that respect, instead the term *request* has been chosen. Furthermore it should be noted, that the *Chronos Registry* is never targeted by any data streams. It is not the case that *Providers* stream their data to the *Registry* for a further distribution to interested *Clients*. Instead the streams are established peer-to-peer. Several *Clients* may request the same channels (also referred to as topics, in the publish-subscribe nomenclature), but they all receive their data directly from the *Provider*. Figure 4.6 illustrates these relationships.

Note that *Clients* and *Providers* are still decoupled in the sense that they don't need to have any a priori knowledge of each other nor does a *Client* have to poll for availability

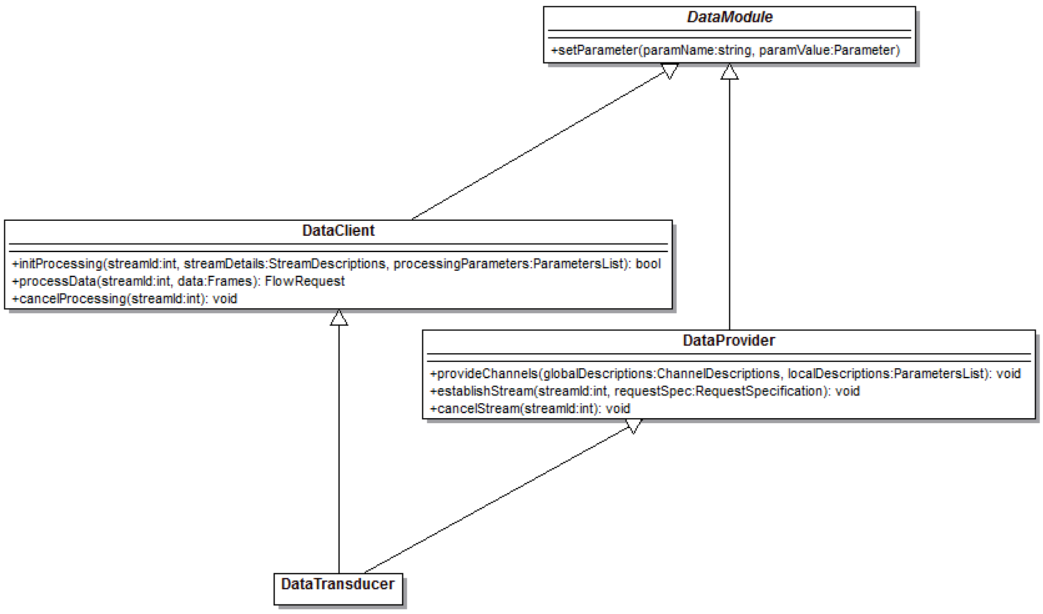

DataModule
+setParameter(paramName:string, paramValue:Parameter)
DataClient
+initProcessing(streamId:int, streamDetails:StreamDescriptions, processingParameters:ParametersList): bool
+processData(streamId:int, data:Frames): FlowRequest
+cancelProcessing(streamId:int): void
DataProvider
+provideChannels(globalDescriptions:ChannelDescriptions, localDescriptions:ParametersList): void
+establishStream(streamId:int, requestSpec:RequestSpecification): void
+cancelStream(streamId:int): void
DataTransducer

Figure 4.7.: Data Modules Class Diagram

of a specific data stream. Instead, a provider publishes its available channels (as configured by the user) at the *Registry*. Similarly, a *Client* requests desired channels from the *Registry*. The *Registry* stores the associated information and when both, *Client* and *Provider* of a particular set of channels are available at the same time, tells the *Provider* which channels to stream and where to stream them to. The implication of this mechanism is that data streams do not have to take unnecessary intermediate routes but are established directly between two participating hosts. It requires that a direct route is available and also has the disadvantage that the streaming backend of a *Provider* has to cover a more complex functionality. This problem however can be mitigated by implementation of a reusable class in the *Chronos* core library, that may be employed by all *Data Providers*. For C++ these requirements have been incorporated into the `Stream` and `RtDispatcher` classes of the core library. From a user's point of view, a *Data Module* is an executable, responsible for one particular functionality. It may be "connected" to other modules by configuration, to realize a desired signal flow. From a programmer's point of view, this executable implements at least one of the interfaces shown in fig. 4.7 and registers it at the *Chronos Registry* accordingly. Common to all *Data Module* types is the `setParameter` method, which is invoked by the *Registry* to set module specific configuration parameters, e.g. as specified by the *Module Configuration* (section 4.2.3). Subsequent scetions will elaborate on the differences between available *Data Module* types.

A *Chronos* application consists of an arbitrary number of *Data Modules*, working together to solve a problem. In general, an application engineer has to implement custom modules, according to the requirements of the particular use-case at hand.

However, a number of concepts could be identified that are recurrent in many different applications. These concepts have been distilled into a set of generic *Core Data Modules* that may be reused for a wide variety of problems. Each one of them is listed and described in chapter 5.

4.3.2. Data Provider

A *Data Provider* is responsible for supplying the application with data streams. These may stem from either a live capture (on-line) or some sort of storage (off-line). The distinction is significant, since it has implications on the streaming process. An on-line stream is inherently clocked and is transmitted exactly from the time on it has been requested. Every subscriber receives the same data at the same time and rate (provided it is able to process the stream at an adequate speed). An off-line stream on the other hand provides more freedom to a requester. Since the underlying data is permanently available, the client may not only choose which parts of the data to retrieve, but also at which rate. It is therefore possible for different clients to receive different data segments from the same storage at varying transmission rates, provided the *Data Provider* is offering this functionality via its *Request Parameters*. A typical example for an on-line stream provider is the *AudioCaptureUnit* (section 5.1). An example for an off-line stream provider is the *FileReader* (section 5.6). An off-line stream provider may offer the capability of simulating an on-line stream while this is not generally true vice-versa, since an on-line stream is potentially infinite.

On a lower level a *Data Provider* is a program implementing the `DataProvider` interface (see fig. 4.7). The interface is held very slim to keep it from being prohibitively difficult to implement in any of the supported programming languages. However, it has been found sufficiently complete to realize every component required so far. It consists of the three methods

- `provideChannels`
- `establishStream`
- `cancelStream`

All of them are invoked exclusively by the *Registry* to control the provider's behavior.

`provideChannels` is used to define what to provide and which global identifiers to use. For example a capturing module needs to know what channels to capture from a given hardware device. When the *Provider* has assured that it is capable of supplying the specified data it notifies the *Registry* by calling its `registerDataChannels` method. The main reason for this indirection is to supersede a *Provider*'s need for a configuration file parser, simplifying the implementation of a custom component.

Only after the channels have been successfully registered, a corresponding stream may be initiated. The *Registry* tells the *Provider* to do so using `establishStream`. The

`RequestSpecification` holds information about which channels to stream, including potential *Request Parameters*, a proxy of the *DataClient* to stream them to and which stream identifier to use. The stream identifier is generated by the *Registry* and globally unique within a *Chronos Application*.

Finally, the `cancelStream` operation is used to abort a stream prematurely. The *Provider* is still required to close the transmission cleanly by completing the last frame of data and attaching an `EndOfStream` marker (see section 4.4).

4.3.3. Data Client

The components at the receiving end of data streams are referred to as *Data Clients*. Most often they serve as an interface to the "outside world", i.e. data is leaving the framework's boundaries. Possible examples include relaying the received data to a different protocol or writing it to some hardware device buffer or storage. Similarly to the aforementioned *Provider*, a *Client* consists of a program implementing the `DataClient` interface (fig. 4.7), which defines the operations

- `initProcessing`,
- `processData`,
- and `cancelProcessing`.

`initProcessing` is invoked by *Data Providers* with the intention of sending a new stream of data. Its parameters contain all available information concerning the scheduled stream. Most importantly this includes the unique `streamId` (which is necessary for the *Client* to support receipt of multiple streams in parallel) and the `streamDetails` (describing the underlying data type and codec specifics). The `processingParameters` are used to supply the *Client* with additional information on how to process the stream. They are optional and may or may not be accounted for by a specific *Data Client* implementation. If a *Client* does not support any of the given `processingParameters` it is to silently ignore them. After analyzing the given information, a *Client* may choose to accept or reject the stream via the boolean return value.

If the *Client* chooses to accept the announced stream by returning `true`, the *Provider* will initiate the streaming process by consequent invocation of the `processData` method. The `streamId` parameter is used to identify the stream each transmitted data package belongs to. The `FlowRequest` return parameter is utilized for a simple type of flow control and may take on one of the values `CONTINUE`, `PAUSE` or `ABORT`. Usually the *Client* will process or buffer the data immediately and return `CONTINUE` to receive the next frame. However, if the *Provider* transmits data at a rate faster than the *Client* is able to handle, it may choose to return `PAUSE`, which makes the `Provider` poll and retransmit the last frame until the *Client* is ready again. If `processData` responds with `ABORT`, the `Provider` is to not send any more frames concerning the respective stream.

This will generally be the case only after `cancelProcessing` has been invoked on the `DataClient` or a fatal error has occurred.

Finally, `cancelProcessing` may be called by the *Registry* to immediately terminate processing of a stream, deallocating all resources reserved for this particular stream. Note that this is not the ordinary way to finish a streaming procedure, which consists of a trailing `EndOfStream` frame, attached by the *Data Provider* (see section 4.4). While a stream that is terminated from *Provider* side will still lead to a processing of all frames in the pipeline (on the network and possible software buffers), invocation of `cancelProcessing` will discard everything that has not been handled, which may be desirable in some situations.

4.3.4. Data Transducer

A *Data Transducer* is an application implementing both, the `DataClient` and `Data-Provider` interfaces. It does not provide any additional functionality. However, there is a certain interdependency between the received and provided channels, that is, the input is subject to some form of operation and made available at the output in a transformed way. This is the typical processing module in *Chronos Realtime*. As opposed to a normal *Data Provider*, its output streams are generally driven by its inputs. Therefore the rate at which a *Data Transducer* will be able to stream its data is typically dependent on the components preceding it in the data flow. This implies that if any of the preceding modules fails or becomes unavailable, the output streams of the *Data Transducer* may also be put on halt. The exact behaviour however is definable by the implementor.

4.4. Chronos Streams

Streaming is the process of sequentially transferring data from a server (source) to a client (sink). In *Chronos Realtime* the source must have `DataProvider` capabilities, while the sink needs to implement the `DataClient` interface. As opposed to a regular file, a stream does not necessarily have a determined beginning or end and is therefore potentially infinite. Another property of a stream is, that it might be restricted to run at a specific rate, either due to client or server constraints. It is therefore not expedient to transfer a stream as fast as possible, but to adjust transmission rate to a value adequate for both ends. This characteristic has been accounted for in streaming design and implementation.

Although designed with audio in mind, *Chronos Realtime* is capable of handling other types of data. Additionally, all these heterogeneous types may follow different encoding schemes. For example, in one application it might be necessary to transfer an uncompressed 24-bit audio stream, while in another case a heavily compressed stream might be sufficient or desired due to bandwidth constraints. Another valuable feature is the ability to augment a stream of data with control or meta-information (see section 4.4.2), as it allows for a frame-exact annotation or custom actions to be carried out. It would not be possible for a *Provider* to invoke operations with frame-level accuracy on a *Client*, if it wasn't by embedding them directly into the stream.

Chronos Realtime streams might stem from either an on-line or off-line source. Although conceptually similar, there exists a distinctive difference concerning availability of data. While in an on-line scenario it is clear that a client will receive a stream starting at request time, in the off-line case a client may want to retrieve arbitrary parts of available data. The implications on the provider are significant, as in the case of multiple clients, every one of them will receive the same data at the same point in time in the on-line case, hence the same packets may be distributed to every client. In the off-line scenario a provider needs to keep track of every single stream and handle clients separately. This notion explains the distinction made between the two cases. *Chronos Realtime* has been designed specifically with on-line streaming in mind, whereas off-line streaming has been added as a secondary feature. This is reflected by the architecture and implementation of several core components.

The streaming process itself, as implemented by the C++ core library, follows a semi-synchronous design. Although remote invocations are carried out asynchronously, meaning the caller does not wait for the callee's response, subsequent packets are never transferred concurrently. Instead, the next packet is prepared for transport and sent immediately as a positive response of the last call is received. This is a rather simple but powerful and safe scheme, enabling reactive flow-control. Albeit potential for optimization in certain scenarios is conceivable, the performance achieved with this approach has hitherto proven sufficient (see section 4.4.3).

4.4.1. Stream Establishment

A *Chronos Stream* is generally initiated by the *Chronos Registry*, either due to a *Channel Configuration* file or due to an invocation of the *Registry*'s `request` method. The *Registry* performs a lookup to locate the corresponding *Provider* and prompts it to begin the streaming procedure by calling `establishStream`. The *Provider* prepares the necessary resources and conveys information about the upcoming stream to the intended *Client* via the `initProcessing` operation. If the *Client* accepts the transmission, the *Provider* goes on to continuously call `processData` for as long as data is available. Since most of the information describing the stream is conveyed via `initProcessing`, the stream itself does contain only little overhead. The whole process is illustrated in figure 4.8. The format used to package the data is described in section 4.4.2.

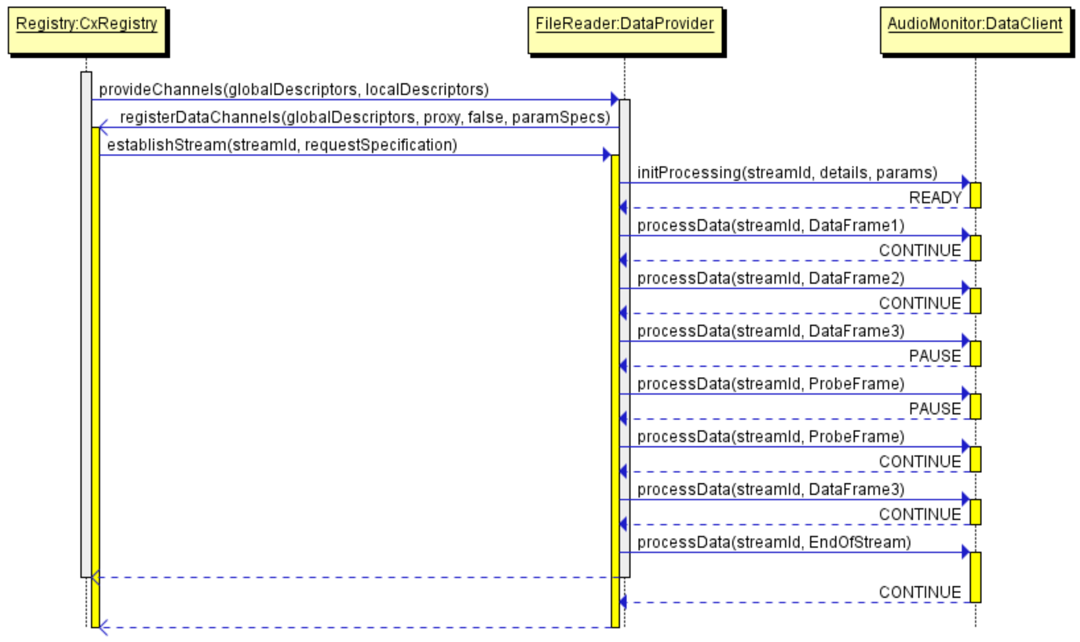

Registry:CxRegistry
FileReader:DataProvider
AudioMonitor:DataClient
provideChannels(globalDescriptors, localDescriptors)
registerDataChannels(globalDescriptors, proxy, false, paramSpecs)
establishStream(streamId, requestSpecification)
initProcessing(streamId, details, params)
READY
processData(streamId, DataFrame1)
CONTINUE
processData(streamId, DataFrame2)
CONTINUE
processData(streamId, DataFrame3)
PAUSE
processData(streamId, ProbeFrame)
PAUSE
processData(streamId, ProbeFrame)
CONTINUE
processData(streamId, DataFrame3)
CONTINUE
processData(streamId, EndOfStream)
CONTINUE

Figure 4.8.: Example: Stream Establishment and Transmission

The sequence diagram illustrates the communication involved in a typical streaming procedure

4.4.2. Streaming Format

Data in *Chronos* is generally transported in chunks called frames. The class `Frame` has been defined as a base for anything that should be carried by a *Chronos Stream*. Since there are many different types of information that may be transferred, the `Frame` class only serves as a placeholder for derived types. Its only member `mTime` carries a timestamp, which is the only information all frame types have in common. Every available frame type[4] falls into one of two categories:

- `DataFrame`
- and `ControlFrame`,

both of which are derived directly from `Frame`.

As the name implies `DataFrame` is used to transfer some type of data. Internally this data is simply stored as a vector[5] of byte arrays, which allows for maximal generality. It would have been possible to provide different types of frames for different types of data, to simplify data access. However, this approach would require a *Client* to implement different access methods for every supported data type and introduction

[4]as of *Chronos Realtime* Version 2.0

[5]The vector is used because a frame may contain multiple channels. Each entry of the vector corresponds to one channel.

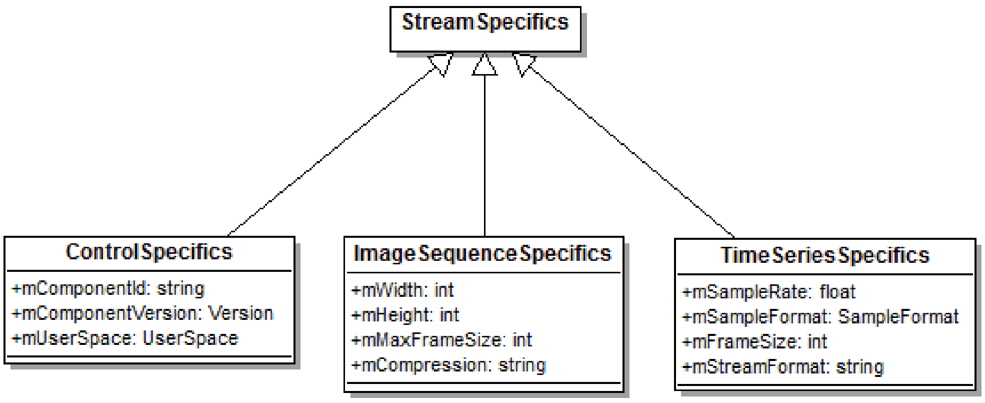


Figure 4.9.: Different Types of `StreamSpecifics`

of a new type would necessitate an upgrade of all involved components. Albeit many *Clients* need to interpret data of different types and handle them accordingly, there are components that are entirely type agnostic (e.g. a binary file writer) and therefore profit a lot from this design choice. The reason for using a vector lies in *Chronos Realtime*'s inherent multichannel support. A stream may carry an arbitrary number of synchronous data channels. These may represent all the same but also heterogeneous modalities. Particularly in audio applications it is common to transport a high number of synchronously sampled channels in parallel. The implemented structure simplifies accessing and processing these channels separately. A `DataFrame` therefore contains the member `mDataChannels`, which is indexed by a channel number to retrieve the actual `RawData` byte buffer. If the *Client* intends to process incoming data it has to interpret it according to its content type, which is specified by the parameters of `initProcessing`. The byte-order has been stipulated as little endian, to avoid CPU intensive conversions on x86 architectures.

More specifically, `initProcessing` supplies the client with `StreamDescriptions` to describe a stream's content. Every channel within a stream is represented by a separate `StreamDescription`, containing the `GlobalID`, `ChannelDataType` and a `StreamSpecifics` object. The `StreamSpecifics` class serves as a base for the derived

- `ControlSpecifics`,
- `ImageSequenceSpecifics`,
- and `TimeSeriesSpecifics`.

Each of them contains parameters relevant to interpretation of the respective stream type. The distinction occurs due to the different properties associated with the various supported types of data. A client may use the `ChannelDataType` field to infer which `StreamSpecifics` subtype to cast to. If the client operates in a non-interpreting way (e.g. a binary writer), the `StreamSpecifics` can be entirely ignored.

Figure 4.9 illustrates the different types of *Specifics*. Most attributes are self-explaining. The ones deserving some more elaboration are:

- `mUserSpace`: Used for a control channel to enable a simple form of user administration. For example a client might choose to reject certain control frames from a user space stream and accept them only from streams belonging to an administration space.
- `mSampleFormat`: Used to specify the data type carried within the `DataFrames` to enable a correct interpretation of byte arrays.
- `mStreamFormat`: String identifying the stream format. For a standard *Chronos Realtime* time series stream this is '**PCM**'. The attribute become relevant when using compressed formats for streaming.

The second big category of frames, `ControlFrame`, is used for transferring commands or other information that may influence the path of execution. A stream may constitute a pure control stream, but interleaving data frames with control frames is permitted and allows for a synchronized, deterministic interaction between the two. There are several different subtypes of control frames, serving varied purposes. The receiving client may distinguish them by dynamic casts. Due to the polymorphic design it is possible to extend the number of types in a future version of the framework without compromising the operativeness of existing clients. Currently the following `ControlFrame` subclasses are supported:

- `PropertyFrame`
- `EndOfStreamFrame`
- `DropoutFrame`
- `ProbeStream`

A `PropertyFrame` is used to transmit a key/value pair. Its main purpose is the possibility to dynamically control properties of a client. As opposed to a *Module Parameter*, the parameter name is not hard-coded into the application but might change due to module configuration. For example, a *Simulink Module* (section 5.3) may load an arbitrary plugin during runtime, as specified by its configuration. However, every plugin offers unique runtime parameters (e.g. a reverb effect is controlled very differently from a dynamics compressor). These parameters may be addressed by use of property frames. The structure of `PropertyFrame` is simple. It contains 3 members that need to be set accordingly. `mName` is a string specifying the property name. `mValue` is a byte array, specifying the associated value. Since a property might be used for a variety of use cases, `mValue` is to be interpreted as specified by `mFormat`, allowing for maximum flexibility.

An `EndOfStreamFrame` is always attached after a *Chronos Stream* transmitted its last frame of data. This enables a client to synchronously deallocate all resources associated with the respective stream. It does not supply any additional information.

A `DropoutFrame` is used to inform a client about a gap in the incoming data. This allows the client to initiate some form of recovery routine if necessary. As for every type of frame, it may simply be ignored if the client is unable to handle this situation appropriately.

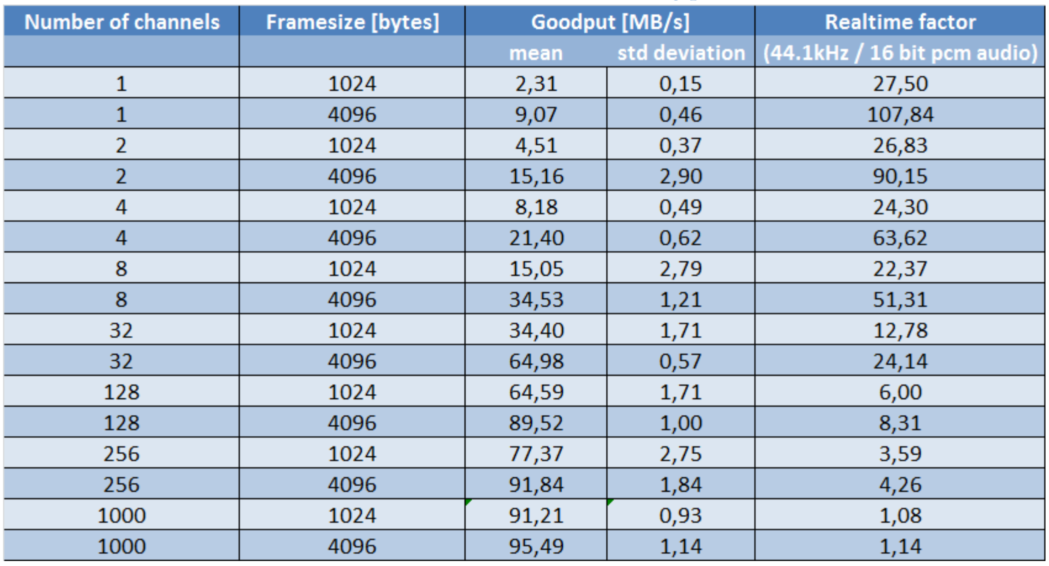
Finally, `ProbeStream` may be used by a provider to test a client's ability for processing more data. As described in section 4.3.3, the return code of `processData` contains information about a client's capability to accept incoming data. To avoid wasting bandwidth by retransmitting the same data several times during polling, a single `ProbeStream` frame may be transmitted to probe for the client's state with only little overhead. Only after the client signals readiness, the actual data needs to be retransmitted.

4.4.3. Performance

For dimensioning a big-scale system it is inevitable to have a notion of maximum achievable transfer rates, that is, how much user data may be streamed between two nodes connected to an ethernet LAN using *Chronos Realtime* in a practical setting. To this end, a number of experiments have been carried out to measure the actual goodput. For the measurements, two standard workstations were connected to form a 1 GBit network via a customary switch. To prevent distortion of the procedure, no other host or internet gateway was involved. This is the preferred way to run a *Chronos Realtime* application in general, as an interference with traffic of external sources is not desirable and may lead to unpredictable delays in adverse situations. Two *Data Modules* have been written specifically for stream testing. A *Data Provider* generating random data on-the-fly and a *Data Client* accepting any incoming streams, counting the number of received media bytes and timing the total stream duration. Both have been designed so as to use as little resources as possible (e.g. no disk access, low cpu usage), to eliminate potential bottle-necks other than the network stream itself. In the experimental set-up, the *Provider* was transferring 1 gigabyte of generated data to the *Client* executing on another host. Each measurement has been carried out at least 6 times. To simulate a real-time setting, common audio frame sizes were used and streamed individually (the "worst-case" scenario). This is for example the case if data is captured from an audio device and should be processed with as little latency as possible by another host. Note that in an off-line setting (i.e. all data is permanently available), or if latency is not an issue, data is buffered and sent in frame containers instead, which offers much higher streaming performance due to decreased protocol and synchronization overhead. However, in the case of real-time processing, maximum achievable transmission rates are insignificant as long as they surpass the rate of data generation.

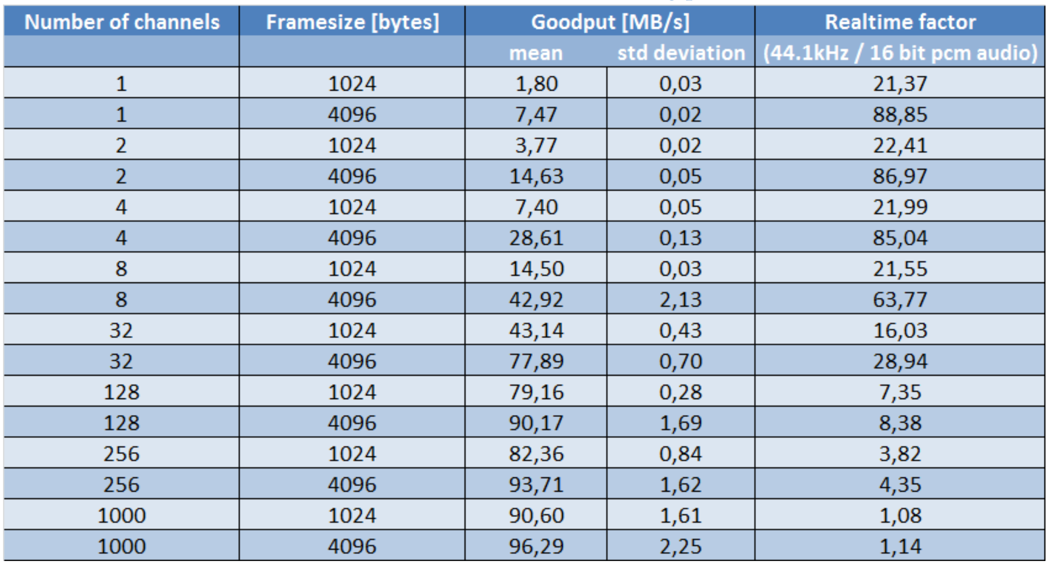
Streaming performance has been evaluated for different frame sizes and different

Table 4.2.: Microsoft Windows 7 streaming performance



Number of channels	Framesize [bytes]	Goodput [MB/s]		Realtime factor
		mean	std deviation	(44.1kHz / 16 bit pcm audio)
1	1024	2,31	0,15	27,50
1	4096	9,07	0,46	107,84
2	1024	4,51	0,37	26,83
2	4096	15,16	2,90	90,15
4	1024	8,18	0,49	24,30
4	4096	21,40	0,62	63,62
8	1024	15,05	2,79	22,37
8	4096	34,53	1,21	51,31
32	1024	34,40	1,71	12,78
32	4096	64,98	0,57	24,14
128	1024	64,59	1,71	6,00
128	4096	89,52	1,00	8,31
256	1024	77,37	2,75	3,59
256	4096	91,84	1,84	4,26
1000	1024	91,21	0,93	1,08
1000	4096	95,49	1,14	1,14

Table 4.3.: Ubuntu Studio 13.10 streaming performance



Number of channels	Framesize [bytes]	Goodput [MB/s]		Realtime factor
		mean	std deviation	(44.1kHz / 16 bit pcm audio)
1	1024	1,80	0,03	21,37
1	4096	7,47	0,02	88,85
2	1024	3,77	0,02	22,41
2	4096	14,63	0,05	86,97
4	1024	7,40	0,05	21,99
4	4096	28,61	0,13	85,04
8	1024	14,50	0,03	21,55
8	4096	42,92	2,13	63,77
32	1024	43,14	0,43	16,03
32	4096	77,89	0,70	28,94
128	1024	79,16	0,28	7,35
128	4096	90,17	1,69	8,38
256	1024	82,36	0,84	3,82
256	4096	93,71	1,62	4,35
1000	1024	90,60	1,61	1,08
1000	4096	96,29	2,25	1,14

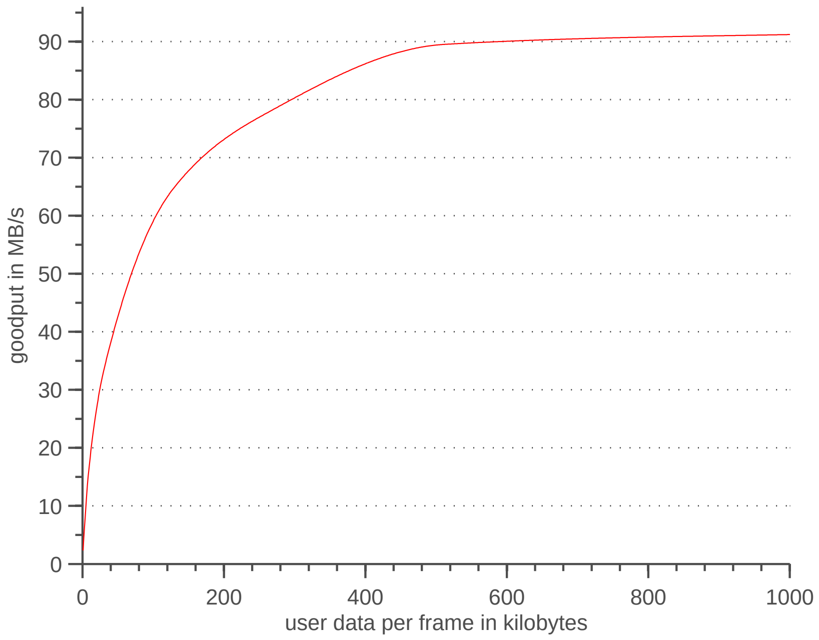


Figure 4.10.: Relation between Packet Size and Goodput

numbers of parallel channels[6]. These parameters have a substantial effect on goodput, since they determine how much data can be integrated into one remote invocation. The more data is passed at once, the better the transfer may be handled by the ICE runtime, the smaller the passed data, the more synchronization overhead caused by the *Chronos Realtime* streaming implementation is in place. Although there is room for optimization in this regard, it will be seen that for current use-cases this is not a requirement and only leads to a complication of error-handling and flow control. Tables 4.2 and 4.3 summarize the measurement results for the two test machines operating on either Microsoft Windows 7 or Linux[7] respectively. Both operating systems were deployed out of the box and have not been tuned specifically to the task. As expected, higher frame sizes and more simultaneously transmitted channels lead to higher goodput. For example a frame size of 4096 bytes has a speed-up of about factor 4 over a frame size of 1024 bytes. Similarly, 4 channels are transmitted with nearly 4 times the goodput as a single channel. This seemingly linear relation flattens off at higher package sizes and eventually plateaus at the maximum achievable rate of up to nearly 100 MB/s. The experimentally obtained plot shown in figure 4.10 may prove helpful when optimizing a distributed application with respect to its stream configuration. The user data contained within a frame (as represented by the x-axis) results from the sum of the frame sizes of all channels[8].

[6]A *Chronos Stream* is able to transmit several channels at once

[7]Ubuntu Studio 13.10 with a low latency kernel

[8]or the number of channels times frame size in case of a uniform frame size

The two operating systems show very similar characteristics for different frame sizes and an overall comparable performance. One notable distinction was that Linux showed slightly more consistent and reproducible rates on the tests, leading to an overall lower variance of goodput. To get an intuition about the practical relevance of the measured rates, the "*Realtime factor*" for an uncompressed PCM audio signal in CD-quality has been included in tables 4.2 and 4.3. The interpretation of this factor is to be understood as answer to the question *'how much faster than real-time can the data be transmitted?'*. In other words, if the factor is greater than one, the stream is real-time capable. This leads to the result that up to about 1000 CD-quality channels may safely be streamed simultaneously from one host to another on a 1 GBit ethernet using *Chronos Realtime* (the relation to other types of media streams may be easily derived from the given results). Although performance for small channel numbers and frame sizes may seem a little low, it is more than sufficient for real-time transmission. As mentioned before, when real-time is not a requirement, frames are buffered and transmitted in bigger chunks, leading to maximum throughput even for smaller frame sizes and channel numbers.

5. Chronos Core Data Modules

Chronos Realtime may be used to implement many different applications for a diverse number of scenarios and most of them will require the one or other custom *Data Module* to be developed. However, there is a handful of basic principles that tend to recur time and time again. Some components are required by most applications in the exact same or an only slightly modified way. It has been tried to single out these components by analyzing various use-cases and generalizing common schemes. In doing so, a number of concepts for reusable *Core Data Module* emerged. These concepts have been devised and implemented in C++ and are available as ready-to-use executables for a straightforward deployment in novel *Chronos Realtime* applications. For simple applications it might even be sufficient to make use of core modules exclusively. Each of them supports a set of parameters to configure their behavior in detail. When starting from the shell, they all follow the same conventions. Apart from the parameters relevant to the ICE runtime, the following command line arguments are supported:

- **-h** - shows a module specific help text if available
- **-n [module name]** - the name used by the module to register itself at the *Registry* (if this parameter is missing, a unique identifier will be generated)

Since the *Chronos Registry* is located via the IceGrid location service, it is necessary to supply the address of the location service to any *Data Module* started by hand. This is done via the command line argument **–Ice.Default.Locator**, as explained in [13e]. When started from within an IceGrid application this step is superfluous and may be omitted.

In the following sections the currently implemented *Core Data Modules* will be introduced and their functionality outlined. To enable a quick overview of the most important parameters, each one of the modules will additionally be summarized in a quick reference box. This has a very practical justification as the information most relevant for an application developer is visible at a glance. Note that the item *Platform* refers to platforms the module has been tested on - it does not necessarily imply that the module is inoperable on platforms not listed.

5.1. Audio Capturing

Quick Reference - AudioCaptureUnit

- **Type:** DataProvider
- **Purpose:** Capturing data from an audio device
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:**
  - **[Device]**[1] *(String)*: Name of the input audio device
  - **[SampleRate]** *(Float64)*: Sample rate in Hz
  - **[FrameSize]** *(Int32)*: Frame size in samples
  - **[SampleFormat]** *(String)*: Data type of individual samples
- **LocalDescriptions:**
  - **[Channel]** *(Int32)*: Channel number to capture

Every real-time signal processing system needs some sort of input data to enter the system. In object oriented design these inputs are often referred to as sources. Sources are responsible for fetching data from a low-level layer (e.g. a device driver). In case of the *AudioCaptureUnit* data is retrieved from one of the audio devices present on the system. It is responsible for continuously reading input data from the device driver and supplying it to the framework as a real-time stream. Apart from the usual challenges of network streaming, this component is particularly critical in terms of implementation because of the risk of suffering frame drops. A dropout occurs when the software is unable to read a complete buffer of data before it is overwritten by subsequently captured frames. Therefore the responsible routines have to be particularly responsive and may not stall for longer periods of time. A common buffer size for low latency operation is 512 samples, which corresponds to a full buffer every 10.67 ms at a sampling rate of 48 kHz. Professional audio interfaces can operate at even lower buffer sizes and therefore lower latencies as well, however this is not generally recommendable due to the increased risk of frame loss. In the field of audio signal processing, a frame drop is a very critical incident, since discontinuities within the signal can lead to severe errors of subsequently computed feature values. To counteract this risk, the high priority callback invoked by the device driver in the *AudioCaptureUnit* has been reduced to the minimal code necessary. The captured audio frame is directly transferred to a preallocated lock-free FIFO queue, which acts as a sufficiently dimensioned buffer in case of irregularities occurring on the network. The callback returns immediately after the memory-transfer is completed to allow for the next frame to arrive. Despite careful design, on non real-time operating systems like common Windows or Linux distributions, a completely dropout-free operation can never be guaranteed, so a way to handle the occurrence of dropouts has to be defined. This way subsequent signal processing components at least get the opportunity to react to signal errors accordingly. In *Chronos Realtime* this has been realized using the `DropoutFrame` class (section 4.4.2). If a dropout is detected, this special `ControlFrame` is inserted into the data stream at the position where frames are missing. *Data Clients* may interpret this frame and react accordingly.

The available configuration parameters of an *AudioCaptureUnit* module are rather self-explaining. **Device** specifies the audio device to capture from. A list of available devices on the machine may be retrieved by calling the executable with the 'help' command line argument **-h**. If nothing is specified, the default device, as defined by the operating system settings will be used. The portaudio library has been employed to access available audio devices. **SampleFormat** may be either one of the supported datatypes (section 4.4.2). The *Local Description* parameter **Channel** may be used to specify which channels of the audio device to capture. If nothing is defined, consecutive channel numbers starting from 0 are used.

5.2. Audio Monitoring

Quick Reference - AudioMonitor

- **Type:** DataClient
- **Purpose:** Playing back data on an audio device
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:**
  - **[Device]** *(String)*: Name of the output audio device
- **ProcessingParameters:**
  - **[Channel]** *(Int32)*: Channel number on the device to use for playback

An *AudioMonitor* is the counterpart of the *AudioCaptureUnit*. It is a straightforward component, simply playing back any received audio streams on the local audio hardware. Which output device to use may be specified by the *Module Parameter* **Device**. If nothing is specified, the default device, as defined by the operating system setting will be used. Instead of invoking operating system routines directly, the portaudio library has been used to access available audio devices. Therefore, any driver backend supported by portaudio may be used for playback. An *AudioMonitor* is only capable of processing a single incoming stream at once. If an additional stream is established while playback is already active, the old stream will be discarded and the new one will start to play. The number of channels N a stream may contain is only limited by the number of channels supported by the output device. To specify which channel of the stream should be played back on which channel of the audio device, the *Processing Parameter* `Channel` may be used. If nothing is specified, the streamed channels will be played back on the first N channels of the audio device. If `cancelProcessing` is called on the `DataClient` interface of an *AudioMonitor*, playback is halted immediately, discarding potentially filled data buffers. This enables a very responsive control in case of user interaction.

5.3. Simulink Module

Quick Reference - SimulinkModule

- **Type:** DataTransducer
- **Purpose:** Loads and drives a Chronos Simulink DLL
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:**
  - **Model** *(String)*: File path of the shared library to load
- **ProcessingParameters:**
  - **Port** *(String)*: Name of the Simulink model's C++ *Source* to connect this channel to
  - **ModelParameter** *(String)*: Name of the tunable Simulink parameter controlled by this channel
- **LocalDescriptions:**
  - **Port** *(String)*: Name of the Simulink model's C++ *Sink* to connect this channel to

The *SimulinkModule* is used to process incoming data by means of an underlying *Chronos Simulink Dll*. It is an important component to aid in the rapid transformation from a prototyped demonstrator into a real-time system. Which dll to use is specified by the *Module Parameter* **Model**. The model may be exchanged at any point during runtime if the new model is interface-compatible with the previous one. If they are incompatible, the model may be replaced only while there are no streams going out of or into the module. The available number and type of input and output channels is determined by the dll. However, which ones to make available to the application needs to be configured in the *Channel Configuration*. The *Processing Parameter* **Port** is used to map an incoming channel to a specific *C++ Source* within the Simulink model. The *LocalDescription* **Port** is used to map a provided `GlobalId` to the data coming out of a *C++ Sink* of the model. Tunable parameters of a model may be controlled by *Chronos* property streams. To subscribe to such a stream and assign it to a specific model parameter the *ProcessingParameter* **ModelParameter** is used. It specifies the path to a parameter of the model, the name of which may be identified directly in Simulink. An example of how to set up a tunable model parameter is given in section 5.5.

The *SimulinkModule* is capable of driving models with differently clocked ports. Therefore every *C++ Sink* and *C++ Source* may potentially operate on a different sample rate. From this follows that the output rate of the *SimulinkModule* will generally diverge from the input rate as well. Consequently the timestamps of data entering the module will require a resampling step, if the absolute time in relation to the stream is to be preserved. Since a model might introduce an externally unknown delay, only the model itself is capable to accurately carry out the resampling procedure in a safe and accurate way. By convention a mechanism has been established, that enables a model developer to access the timestamp information from within the model. By creating a *C++ Source* and naming it '**TimeIn**', it will automatically be supplied with timestamps from the incoming data stream. Similarly, by creating a *C++ Sink* called '**TimeOut**', new timestamps may be assigned to the outgoing data streams. The model developer is responsible for calculating the appropriate values. If a model does not contain a '**TimeOut**' port, new system clock timestamps are generated automatically.

One restriction of the current *SimulinkModule* implementation is that the loaded model must contain at least one *C++ Source* actively supplied with data. If only *C++ Sinks* are present, no output will be generated. This is due to the model being driven by its input streams instead of running on a clock of its own. In a future version this constraint could be resolved by implementing an actively clocked driver in case no input ports are present.

5.4. Ring Buffer

Quick Reference - RingBuffer

- **Type:** DataTransducer
- **Purpose:** Buffers a configurable amount of past data on disk
- **Platform:** Win32, Win64
- **ModuleParameters:**
  - **[SavePath]** *(String)*: Directory the files should be stored in
  - **[Duration]** *(Int32)*: Minimum number of frames to buffer
- **ProcessingParameters:**
  - **[Filename]** *(String)*: Name of the file the channel should be stored in
- **RequestParameters:**
  - **StartTime** *(Int64)*: Time stamp to start streaming from (if **Offset** is 0)
  - **[Tolerance]** *(Float64)*: Specifies how much a located time stamp may deviate from **StartTime**
  - **[Offset]** *(Int32)*: Frame offset to start streaming from relative to **StartTime**
  - **FrameCount** *(Int32)*: Number of frames to stream

The *RingBuffer* component is used to buffer a configurable amount of incoming data frames relative to the most recent ones. It enables an application to access data not only in real-time but to also access data from the past. The maximum amount of data to be buffered has to be specified beforehand using the **Duration** parameter and cannot be changed dynamically once the buffers have been created. Internally the buffers have been implemented using memory mapped files that are regularly flushed to disk. This makes the data permanent as long as it does not fall out of the specified duration range, even in case of a power-out. Once the specified duration is exceeded, the oldest data frames will be overwritten with the most recent ones. Every single channel is saved to a separate file, making it simple to back-up data directly from the file system if needed. Attention has been paid to support very large buffer sizes >4 GB, even on 32-bit Windows. To make the *RingBuffer* store a channel it is sufficient to simply establish a new stream. If a particular file name is desired for this channel it may be specified via the *Processing Parameter* **Filename**, otherwise a suitable name is chosen automatically. Data is stored in combination with its associated time stamps. Therefore it is possible for a request to specify an absolute time range to stream data from. There are several *RequestParameters* available for a detailed description of the desired time range. Generally the **StartTime** parameter is used to specify which time point to start streaming from. However, this time point may be modified by the **Offset** parameter, which enables a user to start the stream from a specifiable amount of frames earlier (negative offset) or later (positive offset) than the given start time. Since a requester will not necessarily know the exact time stamps stored in the ring buffer, it is possible to endow the start time with a certain tolerance. The **Tolerance** parameter specifies by how much time in seconds the reference start point may deviate from the one specified as **StartTime**. Finally the **FrameCount** parameter is used to define the number of frames to be streamed.

Requests may be dispatched very quickly, since the *RingBuffer* is capable of reading and writing concurrently and it internally maintains an index, accelerating lookup of timestamps significantly. An important constraint of the employed method is that the timestamps of an incoming stream must always be increasing. If this constraint is not met, the *Ringbuffer* might fail in locating a timestamp requested by the **StartTime** parameter.

5.5. Controller GUI

Quick Reference - ControllerGUI

- **Type:** DataProvider
- **Purpose:** Graphical User Interface for control of property streams
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:** -
- **LocalDescriptions:**
  - **[Widget]** *(String)*: Widget type to use for the property channel
  - **[MinValue]** *(Float64)*: Minimum value of the property
  - **[MaxValue]** *(Float64)*: Maximum value of the property
  - **[DefaultValue]** *(Float64)*: Default value of the property

The *ControllerGUI* is an interactive *Data Provider*, offering a dynamically configurable graphical user interface. It allows a user to control tunable parameters of one or several *Data Clients* during runtime. The number and type of controllable properties is specified by the respective *Channel Configuration*. As an example consider the simulink model presented in section 3.1.1, figure 3.4. When compiled using the *Chronos Dll Target* it may be loaded into an instance of a *SimulinkModule*. All tunable parameters

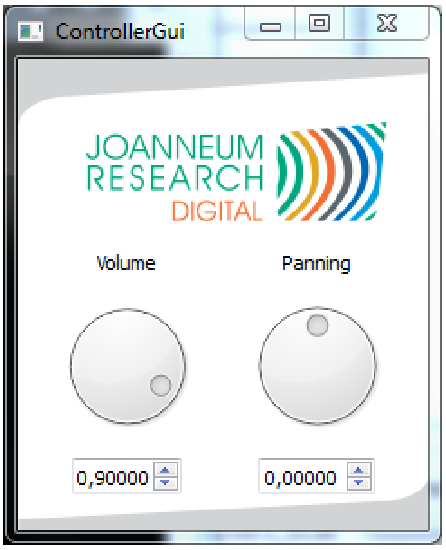


Figure 5.1.: Screenshot of a *ControllerGUI* Window

of the model are then modifiable during runtime via property frames. A configured *ControllerGUI* for the *GainPan16* model may look like the one shown in figure 5.1. In this case two properties have been specified in the *Channel Configuration*. Listing 5.1 shows what the configuration of one property channel looks like, in this concrete example. Apart from default (initial), minimum and maximum allowable value for the property a widget type to be displayed in the GUI may be specified. Currently only two types of widgets are supported:

- **Dial**
- and **Slider**.

Both of them are additionally equipped with a spinbox, where the desired property value may be entered directly. As soon as one of the elements is modified, a `PropertyFrame` is sent to all *Data Clients* who subscribed to the stream.

Listing 5.1 Example snippet of a *ControllerGUI Channel Configuration*

```
...
<Provide>
  <Channel>
    <GlobalID>UserVolume</GlobalID>
    <DataType>CONTROL</DataType>
    <LocalDescription>
      <Parameter name="DataType" xsi:type="String">FLOAT64</Parameter>
      <Parameter name="Widget" xsi:type="String">Dial</Parameter>
      <Parameter name="MinValue" xsi:type="Float64">0</Parameter>
      <Parameter name="MaxValue" xsi:type="Float64">1</Parameter>
      <Parameter name="DefaultValue" xsi:type="Float64">0.9</Parameter>
    </LocalDescription>
  </Channel>
...
```

Clients subscribe to a property stream in much the same way as to any other data stream. Using a *Processing Parameter*, a property may be mapped to some internal parameter to be tuned. The exact mapping scheme is specific to the client's implementation. An example for the *SimulinkModule* component is given in listing 5.2. The property stream **UserVolume** is requested and assigned to the model's **Gain** using the *Processing Parameter* **ModelParameter**. Note that the same stream may be requested several times and assigned to multiple different parameters if desired.

Listing 5.2 Example snippet of the *SimulinkModule Channel Configuration* corresponding to listing 5.1

```
...
<Request>
  <Channel>
    <GlobalID>UserVolume</GlobalID>
    <DataType>CONTROL</DataType>
    <ProcessingParameters>
      <Parameter name="ModelParameter" xsi:type="String">GainPan16/pGain/Gain</
          Parameter>
    </ProcessingParameters>
  </Channel>
...
```

5.6. File Reader

Quick Reference - FileReader

- **Type:** DataProvider
- **Purpose:** Reading data from files of various formats
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:** -
- **LocalDescriptions:**
  - **FilePath** *(String)*:
- **RequestParameters:**
  - **[ChannelPath]** *(String)*: Specifies which channel of the file to read
  - **[StartPos]** *(Int64)*: Sample position to start reading from
  - **[RtSimulation]** *(Int32)*: If set, the stream will be sent at a rate according to its timestamps instead of as fast as possible
  - **[OnEof]** *(String)*: Specifies how to proceed after reaching the end of file

The *FileReader* is designed to read files of various formats and to provide their content to other *Chronos* modules. Currently supported file formats are wav, hdf5, csv and binary. The format is determined by filename extension. If no extension is present, binary is chosen as default setting. The *FileReader* has been laid out so as to enable a straightforward addition of recognized file formats. This is achieved by simply subclassing the internal `StreamFileReader` interface and implementing the desired file format within this scheme. Once registered the new `StreamFileReader` is than automatically instantiated by the internal *Factory Method* [2] when requested. Every file that should be made available to the application needs to be provided with a `GlobalId` using a *Channel Configuration*. The *Local Description* **FilePath** is used to specify where in the filesystem the file is located. When the file stream is requested, several *Request Parameters* may be specified. If a file contains multiple channels **ChannelPath** may be used to specify which one of them should be transmitted. This parameter is file format specific, as the various formats tend to organize their internal data structure differently. For wave files e.g. simple integer numbers starting from `0` may be used to identify a channel. The exact naming scheme should always be documented in the respective `StreamFileReaders` header file. Formats that do not support multiple channels simply ignore this parameter. If data should not be streamed right from

[2]see e.g. [ES10] for details on this software design pattern

the beginning of a file but start at some offset, **StartPos** may be used to specify this position in samples.

Since the process of reading from a file does not adhere to a natural clock (all data is available right away) a stream requested from the *FileReader* module will generally be transmitted at maximum speed. However, in some situations (particularly for tests or demonstrators) it is useful to simulate a real-time stream as if it was originating directly from a sensor instead of a file. If this behavior is desired, the parameter **RtSimulation** should be set to 1.

Finally **OnEof** specifies how the *FileReader* should behave when reaching the end of a file. Available options are:

- **Zeros** - Fills the last frame with zeros until the specified frame size is reached
- **Stop** - Stops immediately when EOF is reached (the last frame may contain less samples than the specified frame size)
- **Repeat** - When EOF is reached the stream goes into a loop starting from the position specified in **StartPos**

5.7. File Writer

Quick Reference - FileWriter

- **Type:** DataClient
- **Purpose:** Writing data to files in a specified format
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:**
  - **[FileFormat]** *(String)*: Format the writer should save files in (defaults to binary files)
- **ChannelParameters:** -
- **ProcessingParameters:**
  - **FilePath** *(String)*: Path of the file the channel should be stored in
  - **ChannelPath** *(String)*: Specifies the channel that should be written to within the file

The *FileWriter* is designed to write received *Chronos Streams* into files of various formats. Currently supported file formats are wav, hdf5, csv and binary. The *FileWriter* has been laid out so as to enable a straightforward addition of writable file formats. By subclassing the internal `StreamFileWriter` interface and implementing the respective class a new format may be supported. Which format to use for writing is specified by the *ModuleParameter* **FileFormat**. Each `StreamFileWriter` registers itself using a descriptive format identifier (usually the respective file extension), which may then be used in the configuration to specify which format to use. The name of the file a stream should be written to is passed by the *ProcessingParameter* **Filename**. If several channels specify the same file, a multi-channel file is created. Which *Chronos Channel* should be written to which channel within the file may be specified using **ChannelPath**. As was the case with the *FileReader* the format of **ChannelPath** is specific to the respective `StreamFileWriter`. For wave files e.g. simple integer numbers starting from `0` may be used to identify a channel. When implementing a new format, the exact naming scheme should always be documented in the respective `StreamFileWriters` header file.

5.8. Signal Generator

Quick Reference - SignalGenerator

- **Type:** DataTransducer
- **Purpose:** Providing generated signals
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:** -
- **ProcessingParameters:**
  - **OscillatorGain** *(String)*: Name of the Oscillator the gain of which should be controlled by this channel
  - **OscillatorFrequency** *(String)*: Name of the Oscillator the frequency of which should controlled by this channel
- **LocalDescriptions:**
  - **[Osc(%i)Type]** *(String)*: Oscillator type
  - **[Osc(%i)Gain]** *(Float64)*: Oscillator amplitude
  - **[Osc(%i)Frequency]** *(Float64)*: Oscillator frequency
- **RequestParameters:**
  - **[RtSimulation]** *(Int32)*: If set, the stream will be sent at a rate according to its sampling rate instead of as fast as possible

The *SignalGenerator* has been specifically implemented for the execution of performance tests (section 4.4.3) and measurements. It is capable of providing signals comprised of a superposition of configurable digital oscillators and noise. One instance of a *SignalGenerator* may only synthesize a single signal at a time. However, the same channel may be subscribed to multiple times within the same request, if multi-channel streaming is desired. Furthermore it is possible for multiple clients to concurrently receive the same waveform at differing sampling rates and sample data types, as every client is served by a separate internal generator object. The waveform generated by the module is determined by its *Channel Configuration*. More specifically, the `LocalDescription` of the provided channel is used to define number, type and initial settings of oscillators to use. The **name** attributes must consist of a prefix (**Osc**) and an integer **(%i)** to identify an Oscillator, and a postfix (**Type**, **Gain** or **Frequency**) to identify a specific property. Any 32-bit integer may be chosen as an identifier, however it is sensible to enumerate starting at 0 or 1. If any property with a unique identifier is specified, an according oscillator will be created. None of the properties are mandatory, if omitted, default values will be used. Listing 5.3 shows an example of what a `LocalDescription` of a *SignalGenerator* containing two oscillators may look like. Note that since no **Gain** has been specified for **Osc2**, a default value will be assumed.

Listing 5.3 Example snippet of a *SignalGenerator Provide Channel Configuration*

```
...
 <Provide>
   <Channel>
     <GlobalID>Synth</GlobalID>
     <DataType>AUDIO</DataType>
     <LocalDescription>
       <Parameter name="Osc1Type" xsi:type="String">Square</Parameter>
       <Parameter name="Osc1Frequency" xsi:type="Float64">200.0</Parameter>
       <Parameter name="Osc1Gain" xsi:type="Float64">0.5</Parameter>
       <Parameter name="Osc2Type" xsi:type="String">Sin</Parameter>
       <Parameter name="Osc2Frequency" xsi:type="Float64">280.7</Parameter>
     </LocalDescription>
   </Channel>
 </Provide>
...
```

Allowed values for the **Type** property are:

- **Sin** - For sinusoidal waveforms
- **Triangle** - For triangular waveforms
- **Square** - For rectangular waveforms
- **Saw** - For sawtooth waveforms
- **Noise** - For white noise

A qualitative overview of how these waveforms look like is given in figure 5.2. Each of the periodic signals is plotted over four periods with an amplitude of 1. The noisy waveform example consists of 3600 data points drawn from a uniform distribution over an interval of -1 to +1.

Note that for the *SignalGenerator* the maximum representable dynamic range is confined to the interval of -1 to +1. This implies that the multiplicative **Gain** factor should similarly not exceed these values if clipping is to be avoided. The same rule applies to the sum of the gains, if multiple oscillators are employed. However, if the synthesized signal exceeds permissible values it is automatically limited. While for musical synthesis the resulting formation of overtones may be a desirable property, for measurement purposes it is generally to be avoided.

Albeit unambiguously a *Data Provider*, the *SignalGenerator* additionally implements a `DataClient` interface to make oscillator parameters dynamically controllable via property streams. The *Processing Parameters* **OscillatorGain** and **OscillatorFrequency** are used to map a requested property channel to the respective oscillator parameter. The initial **Gain** and **Frequency** settings specified by the *Local Descriptions* may therefore be overridden during runtime. Listing 5.4 shows an example of how a property channel named **Osc1Freq** may be mapped to control the frequency of oscillator **Osc1** in a *SignalGenerator* module.

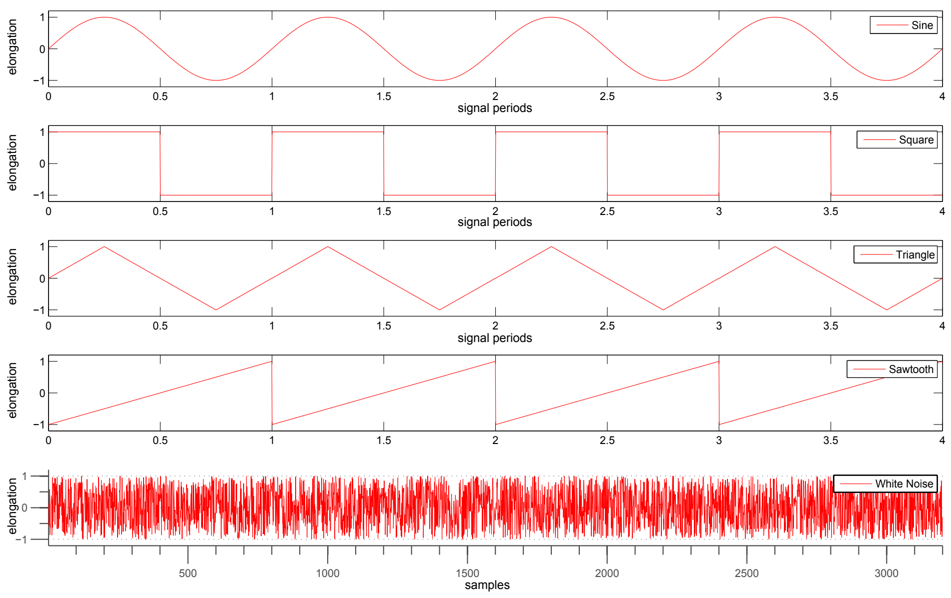


Figure 5.2.: Time-domain Representation of available Waveform Types

Listing 5.4 Example snippet of a *SignalGenerator Request Channel Configuration*

```
...
    <Request>
    <Channel>
      <GlobalID>Osc1Freq</GlobalID>
      <DataType>CONTROL</DataType>
      <ProcessingParameters>
        <Parameter name="OscillatorFrequency" xsi:type="String">Osc1</Parameter>
      </ProcessingParameters>
    </Channel>
  </Request>
...
```

Similarly to the *FileReader* module (section 5.6), it may be desirable to simulate real-time behavior instead of streaming at maximum speed. This is achieved by setting the global `RequestParameter` **RtSimulation** to 1.

5.9. TCP Capturing

Quick Reference - TcpCaptureUnit

- **Type:** DataProvider
- **Purpose:** Capturing data from a TCP port
- **Platform:** Linux, Win32, Win64
- **ModuleParameters:**
  - **Host** *(String)*: Name or IP address of the host to connect to
  - **Port** *(Int32)*: Port number to connect to
- **LocalDescriptions:**
  - **Bytes** *(Int32)*: Channel width in bytes

The *TcpCaptureUnit* has been implemented to support receipt of raw TCP data streams from an external source. This kind of communication is common in graphical or high-level programming languages to enable a simple exchange of data between two hosts (e.g. LabVIEW [13g] or PureData [13i] support sending and receiving data via TCP). Since the *TcpCaptureUnit* acts as a client, the TCP server's address and port number have to be specified using respective *ModuleParameters*. Due to raw streams not following a general format specification, the *Provide* and *Request ChannelConfiguration* are utilized to describe the employed data layout.

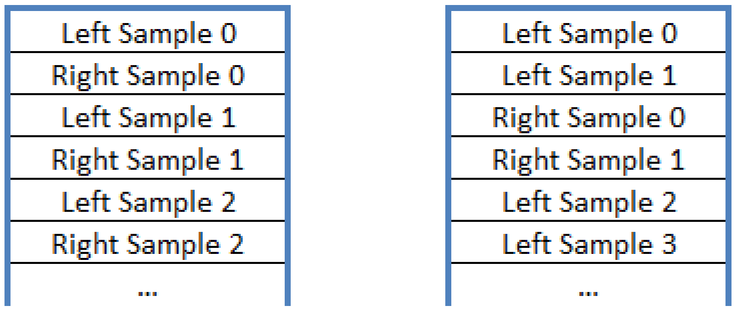


Figure 5.3.: Two Examples of how a Stereo TCP Stream might be arranged

Consider the examples given in figure 5.3 for how a 2-channel audio signal might be streamed. The example on the left is a simple interleaving scheme, whereas the one on the right illustrates frame-based interleaving with a frame size of 2 (In principle sample interleaving is just a special case with a frame size of 1). Depending on the employed sample bit depth, each sample may consist of several bytes. In the *Channel Configuration*, provided channels have to be specified in the same order as they are arranged within the stream and the `LocalDescription` **Bytes** has to be set to *number of bytes per sample* times *frame size*. Listing 5.5 illustrates how a *TcpCaptureUnit* would be set up for receiving a stream of 16-bit audio data conforming to the format depicted on the right side of figure 5.3.

Listing 5.5 Example snippet of a *TcpCaptureUnit Channel Configuration*

```
...
 <Provide>
   <Channel>
     <GlobalID>Left</GlobalID>
     <DataType>AUDIO</DataType>
     <LocalDescription>
       <Parameter name="Bytes" xsi:type="Int32">4</Parameter>
     </LocalDescription>
   </Channel>
   <Channel>
     <GlobalID>Right</GlobalID>
     <DataType>AUDIO</DataType>
     <LocalDescription>
       <Parameter name="Bytes" xsi:type="Int32">4</Parameter>
     </LocalDescription>
   </Channel>
 </Provide>
...
```

Note that the data is not interpreted by the *TcpCaptureUnit* itself. A requester therefore must provide the correct `SampleFormat` in its `TimeSeriesSpecifics` to make the client capable of correct interpretation. Another restriction is that the byte order of the received data has to either conform to the endianness generally employed by *Chronos Realtime* streams, or must be dealt with externally. Extending the concept to support a more elegant treatment of these issues is subject to future work.

6. Chronos Realtime Applications

6.1. Custom Application Development and Configuration

This section is meant to summarize the main procedural steps on how to assemble a custom new application using *Chronos Realtime*. The first questions that need to be answered concern the applicability of existing core components. Can the application be assembled purely from the *Chronos Core Data Modules* described in chapter 5? If the answer is no - Is it expedient to enhance an existing core module with the desired features (e.g. it is simple to add support for new file formats to the *FileReader* or *FileWriter* core modules), or is it necessary to create an altogether new module? If implementation of a custom module is inevitable, a convenient programming language has to be chosen. Currently it is mostly recommendable to use C++ over any other languages supported by ICE, due to the available core classes encapsulating a lot of internal procedures and providing many recurring elements for *Data Module* implementation. In some cases it might be necessary to resort to another programming language due to external factors. It was a major concern to keep interfaces clear and simple to accommodate for this situation, however a certain amount of extra work is to be expected. The interfaces and their expected behavior are documented in section 4.3. Whenever possible, a module should be made configurable exclusively via *Channel-* and *Module Configurations*, so a suitable design has to be devised for this purpose.

Once all required *Data Modules* have been selected or implemented, appropriate configuration files need to be created to define the precise task of each module and how they are supposed to interact with each other (more details and examples may be found in sections 4.2.3, 4.2.4 and chapter 5). Finally an IceGrid configuration file is used to define how the application is to be distributed among different machines (for more details refer to [13e], an example configuration is given in the appendix, listing A.2).

6.2. Real-world Applications

To provide the reader with more insight into the framework's capabilities and application design, this section will review a couple of real-world applications and their corresponding architecture. All of them have been realized using *Chronos Realtime* or an early precursor thereof.

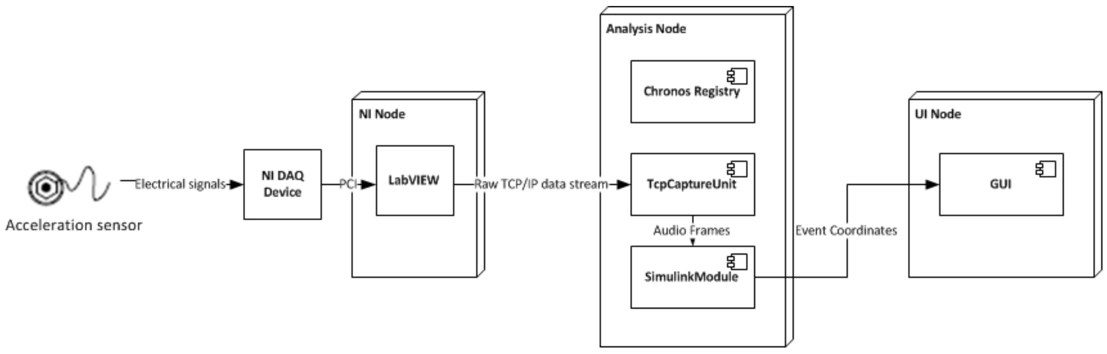


Figure 6.1.: System Diagram of the Tangible Acoustic Interface Application

6.2.1. Tangible Acoustic Interface

This section describes the implementation of a real-time tangible acoustic user interface demonstration system. The setup consists of a tri-axial acceleration sensor mounted on a rigid, flat surface (e.g. a floor or tabletop) with the aim of detecting impacts and estimating their location. In a real system this information might be used to trigger specific actions, depending on the characteristics and location of the impact's sound, in this demonstrator however, the estimated coordinates are simply displayed on a graphical user interface whenever an impact is identified. The theory behind the applied signal analysis is described in [Dob+14] and is not part of this work. Here it is of interest how the developed algorithms have been integrated into a real-time system using *Chronos Realtime* with only little additional implementation overhead. Figure 6.1 illustrates the system diagram of one particular configuration of the demonstrator where three hosts have been used to handle the task. As is generally the case, different components of a *Chronos Realtime* application may be spread over as little or many nodes as reasonable.

Each 3-dimensional box in the figure represents a separate host (or node). Shapes inside the 3-dimensional boxes represent software components running on the respective machine while shapes on the outside represent external hardware devices. *Chronos* modules are specifically designated by the component symbol in the upper right corner. Arrows between shapes denote interfaces. Communication channels with the *Registry* are not explicitly displayed, since every *Data Module* is inevitably interacting with this component.

A tri-axial acceleration sensor was connected to a National Instruments data acquisition device as common for laboratory measurements. To read sampled data from the device, a small "*virtual instrument*" has been developed in National Instruments' LabVIEW software [13g], acquiring frames of audio data using the standard graphical programming tools shipped with the application (the term *virtual instrument* or *vi* was devised by the manufacturer for a program written in LabVIEW). During the prototyping phase this *vi* simply wrote acquired data frames into files, which could then be used for algorithm development and off-line testing. To integrate the capturing process into *Chronos Realtime*, the file sinks have been replaced by a TCP/IP writer in the otherwise unchanged *vi*. Once a raw stream is set up to be transmitted via TCP, it can be transformed into a *Chronos Stream* via an instance of a *TcpCaptureUnit* (see section 5.9). Despite the communication taking place over TCP it is legitimate to run both, LabVIEW and the *TcpCaptureUnit* on the same host if desired. In the setup presented here however, *NI Node* is exclusively executing the described LabVIEW *vi*, therefore requiring no *Chronos Realtime* installation. The lion's share of required computations is handled by the *Analysis Node*. During the prototyping phase, a Simulink model was developed, reading input data from a file and writing results to the Matlab workspace for further inspection. The same model was then taken for application in the real-time demonstrator, except with the file source replaced by a *C++ Source*, the workspace sink replaced by a *C++ Sink* and the model compiled into a shared library (as detailed in section 3.1). The generated plugin was loaded into a *SimulinkModule* (section 5.3) and supplied with sensor data from the *TcpCaptureUnit*. Finally a custom JAVA GUI has been written, implementing a *Data Client* to visualize the estimated coordinates on a mobile device or notebook for demonstration purposes. The stream from the *SimulinkModule* to the GUI module was a pure *Control* stream carrying instances of `PropertyFrame` (section 4.4.2).

6.2.2. Interactive Beamforming on a Smartphone

Another rather small-scale application that has been implemented using a very early version of *Chronos Realtime* is the tech-demo presented in [Ret+11]. It involves a Microsoft Kinect, a Windows workstation and an Android smart phone. Kinect is the name of an extension Module for Microsoft's XBOX 360 platform and has been released in November 2010. It integrates several sensors such as an RGB and depth camera as well as a microphone line-array, consisting of four non-uniformly spaced capsules, which can be used as an acoustic beamformer. Beamforming is a spatial filtering technique, combining the output signals of multiple specifically arranged microphones to achieve a desired directivity pattern. The procedure of changing the direction of the main lobe of such a pattern is referred to as beam steering and constitutes the gist of the application at hand. The field of view of the Kinect's camera is transmitted to and displayed on the cellphone. Using the phone's touch screen, the beamformer of the Kinect may be steered to a desired angle within the field of view by tipping on the corresponding position. The output signal of the beamformer is played back on the phone's audio interface, emphasizing the sounds originating from the selected angle and attenuating environmental noise.

Figure 6.2 shows a diagram of the employed system. The Kinect sensor bar is equipped with an USB interface exclusively, through which it is connected to a standard PC (the *Kinect Node*). Microsoft provides a special software development kit enabling a user to access the various sensors present on the device. Apart from the raw signals a simple

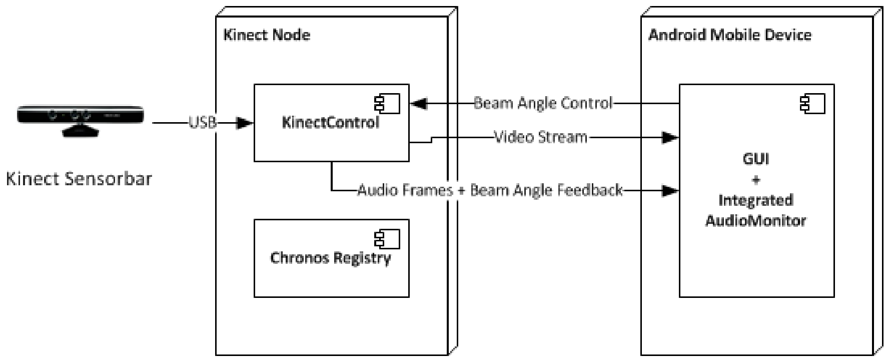


Figure 6.2.: System Diagram of the Beamforming Application

beamforming algorithm is also provided off-the-shelf. This comprehensive set of tools has been integrated into a custom *Chronos Realtime* component called *KinectControl*, offering both a `DataProvider` and `DataClient` interface. It implements the necessary routines to capture audio and video signals from the camera and to control parameters of the integrated beamformer. Naturally it has to be located on the same PC the sensor bar is connected to. The other essential part of this system is a JAVA app developed for the android operating system. Just as the other module it was developed specifically for this task and integrates both `DataProvider` and *DataProvider* objects to enable transmission and reception of *Chronos* streams. It serves as a graphical user interface, displaying a live capture of the Kinect's RGB camera and highlighting the angle the beamformer is currently steered to (figure 6.3 shows a photograph of a cellphone running the GUI). The stream is configured so that it is established by the *Registry* as soon as *KinectControl* and *GUI* are running and stays active until one of them shuts down. In addition there is one control stream transmitting the desired beam angles, as chosen by the user, to the *KinectControl* module. Whenever the user touches the screen, a red point lights up on the display and a `PropertyFrame` carrying the coordinates is transmitted. The user interface app also receives the stream of audio data and plays it back on the device. However, the received audio stream is not a pure data stream, but is interleaved with control frames, updating the steering direction if it had been modified. The GUI highlights the according region on the camera image only after this feedback frame was received. This way the displayed steering angle and the sound playback are always in sync, despite potential delays due to buffering.

Although all involved modules are very application specific and therefore require exhaustive implementation efforts, the main benefit of using *Chronos Realtime* here is the predefined layout of communication necessary between the distributed components. Another benefit is the ease of extensibility, e.g. by integration of a custom beamforming algorithm implemented in Simulink.

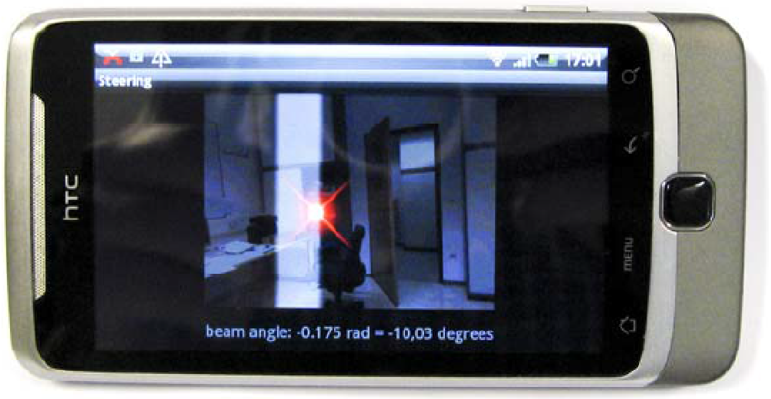


Figure 6.3.: Mobile Phone running the Beamformer Tech-demo

6.2.3. Acoustic Tunnel Monitoring - AKUT

Acoustic Tunnel Monitoring is a term that has been coined by [SG05]. The idea is to equip road tunnels with microphones, in addition to already prevailing video systems, to further enhance the capabilities of automated traffic monitoring systems. Primary goal of safety-measures in tunnels is the prevention of critical incidents that are a threat to human life, the environment and tunnel operating facilities. As a secondary goal, the impacts of critical incidents that have already happened should be kept as low as possible. This is achieved by getting the people involved in such an incident and other endangered tunnel users to safety as quickly as possible. A major aspect in this context is that emergency services, such as fire-brigades, rescue teams and police are alarmed quickly and efficiently. The reaction time of emergency services should be kept as short as possible in order to minimize injuries, property damage and damage to the environment [Ret+10]. As such, the response time of the operator personnel is essential. Due to the large number of cameras assigned to a single operator, it is not possible to monitor all of them concurrently, but instead different camera streams are switched through sequentially. Therefore a number of automated emergency detection systems have been deployed, all of which suffer from a rather slow reaction time (reaching from several seconds to minutes). Almost all dangerous incidents are accompanied by acoustic noise, which may be detected rather quickly (in the range of a few hundred milliseconds to a few seconds) by an adequately designed pattern recognition algorithm. At Joanneum Research a number of algorithms have been developed specifically for this task [GRR11]. However, it is a long way from a functional off-line prototype to a large-scale real-time system integrated into a SCADA infrastructure. This situation has been the starting point of this thesis, as the worlds first pilot system was to be implemented in the "Kirchdorf-Tunnel" in Styria, Austria [Ret+10]. The software originally deployed is still in use to this day and formed the basis of *Chronos Realtime*, which has been generalized in its design to allow for a broader applicability. Its duties cover capturing of several dozens of audio channels,

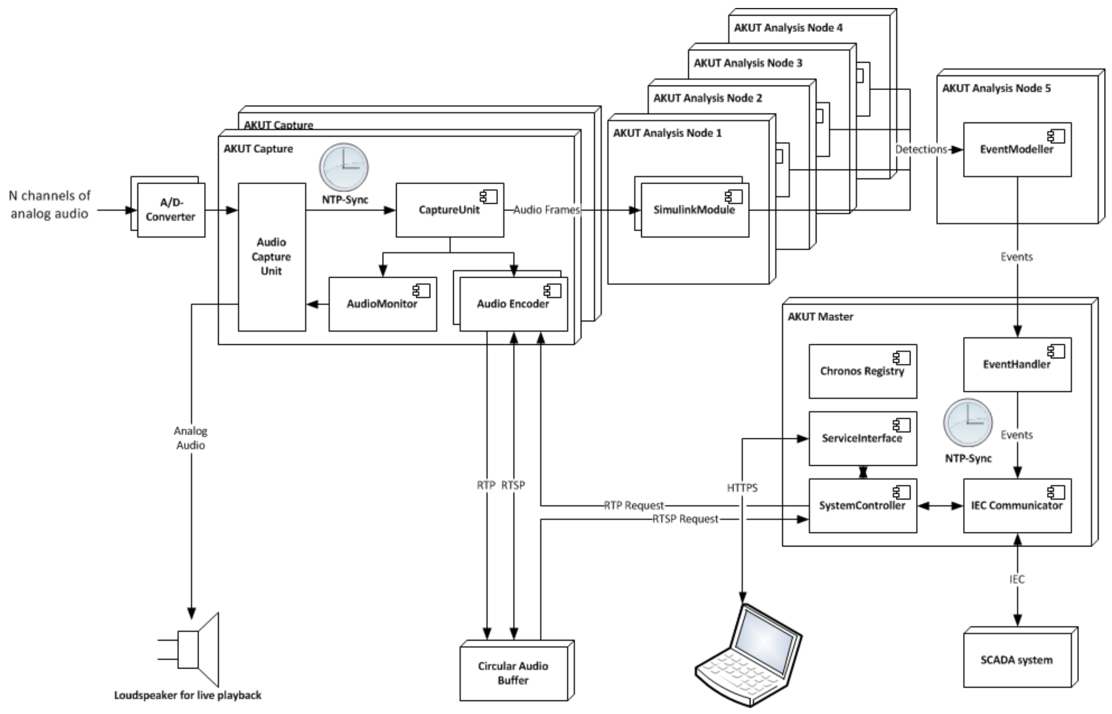


Figure 6.4.: System Diagram of a typical AKUT Application

preprocessing, classification, event-detection, -relay and -storage, 24-hour buffering of uncompressed audio data, live and event playback and controllability via an IEC-60870 SCADA protocol [CRW04]. The principle system diagram of this appliance is shown in figure 6.4.

The following external interfaces are to be handled by the system:

- N audio input channels
- one or more audio output channels for playback
- service interface for system configuration and maintenance
- SCADA interface for interaction with tunnel operator personnel
- optionally an RTP/RTSP interface for external data storage

In this configuration the audio input consists of N A/D converted audio signals captured by one or more professional audio interfaces (N may range from a few dozen to several hundred microphones, depending on the length of the tunnel). The signals are transformed into *Chronos* streams by an *AudioCaptureUnit*. If one of them is to be played live the stream is forwarded to the *AudioMonitor* component, outputting it on the audio device. Optionally an encoder for external data storage may be present, subscribing all audio channels to be stored. Finally, all captured streams are forwarded to the *SimulinkModules* carrying out signal analysis to detect and classify potentially critical audio events. Since the analysis is computationally expensive, a single workstation is not capable of processing all audio channels in real-time. Therefore the work may be distributed among M machines, each of which processes N/M channels. Since there are interdependencies between the signals of consecutive microphones, the individual outputs of the first analysis stage are merged together by the *EventModeller* in a separate step, forming the final event results of the system. The *EventHandler* is responsible for logging and filtering the raw events and forwarding a definable selection to the *IEC Communicator*, the component responsible for interfacing the SCADA system to display the detections on the alarm screens of the tunnel control center. The operators are able to send a couple of commands (e.g. disabling certain microphones or playback of live streams), which are accepted by the *IEC Communicator* and relayed to the *SystemController*. The *SystemController* is a custom component, encapsulating very application specific control features and system monitoring functionality (e.g. if a channel is to be played back or detection of a particular channel or a whole tunnel tube is to be disabled, the *SystemController* knows how to reconfigure the system accordingly). While the *IEC Communicator* is only allowed to execute a reduced set of commands provided by the *SystemController* (as defined in the requirements specification), the *ServiceInterface* component provides full control access to the system via a web interface. This allows system administrators a simplified way of monitoring and reconfiguring the application during runtime.

7. Conclusions And Future Work

For this thesis, a software framework capable of rapidly transforming data stream processing algorithms from a prototyping environment into a distributed real-time system has been designed and implemented. An illustration on how to integrate a Simulink model into a generic, prefabricated frame application using code generation has been given. Support of other prototyping environments is intended but has not yet been implemented. The main objective of *Chronos Realtime* is to allow for a modular composition of distributed streaming applications using basic building blocks. The devised architecture for this task has been described and core modules as well as their configuration procedure have been introduced. Proposed components have been implemented in C++ using the ICE middleware for network communication. Measurements have been carried out to assure real-time capabilities of the implemented data streaming procedure and to give an estimate of achievable throughput for dimensioning a distributed system. To verify its applicability to diversified scenarios, the framework has been applied to various use-cases, three of which have been presented. All of them were realizable with considerably less effort than an implementation from scratch would have required. Furthermore the efficiency of cross-language, cross-platform applications could be verified. It can therefore be concluded that the devised architecture is capable of generalizability and implementation of new systems using *Chronos Realtime* is feasible.

The framework has been outlined in its first operational version. As with any piece of software, there is a lot of room for improvement and extension. Among the features most eligible for future updates are better monitoring capabilities of active stream states and more control over error mitigation routines. Error handling is a particularly intricate task, since different use-cases often turn out to require different actions in case of particular error scenarios. Most common scenarios would have to be identified and a concept for configuring alternative methods of handling them devised. Application configuration is generally a laborious task and could be simplified by implementation of a graphical user interface tool. This would allow for a more intuitive composition of a distributed application and alleviate a user of the burden to write XML configuration files by hand. Another extension beneficial to the framework would be a better integration of further prototyping environments. The programming language Python is of particular interest, since it enjoys high popularity and features a vast amount of libraries available from all fields of application. Since Python is capable of interfacing with ICE directly, it could potentially be integrated into a *Chronos Realtime* without requiring a meta-compilation step, if processing performance is sufficient. To further enable a more robust runtime operation it would be expedient to add mirroring capabilities to the *Chronos Registry*. In other words, several redundant instances, distributed among different hosts, concurrently reside in operation, all sharing the same state. In case of a breakdown of one machine, an application transparent take-over would take place. A promising way of implementing this feature would be to resort to a database back-end inherently supporting distributed replication and utilizing the automatic end-point deduction already inherent to ICE proxies.

Finally it can be said that there is always room for enhancement of existing core data modules and addition of new ones, as their general benefit becomes apparent. Due to its modular, decoupled architecture, implementation of new modules can not negatively affect the framework's core.

Bibliography

[13a] *Audinate*. 2013. URL: http://www.audinate.com/ (cit. on p. 10).

[13b] *Clam*. 2013. URL: http://clam-project.org/ (cit. on p. 7).

[13c] *CppUnit*. 2013. URL: http://sourceforge.net/projects/cppunit (cit. on p. 15).

[13d] *DataTurbine*. 2013. URL: http://www.dataturbine.org/ (cit. on p. 7).

[13e] *ICE Manual*. 2013. URL: http://doc.zeroc.com/display/Ice/Ice+Manual (cit. on pp. 11, 31, 59, 76).

[13f] *IT standards and organizations*. July 24, 2013. URL: http://whatis.techtarget.com/glossary/IT-Standards-Organizations (cit. on p. 2).

[13g] *LabVIEW*. 2013. URL: http://www.ni.com/labview (cit. on pp. 74, 77).

[13h] *Portaudio*. 2013. URL: http://portaudio.com (cit. on p. 14).

[13i] *Pure Data*. 2013. URL: http://puredata.info (cit. on p. 74).

[13j] *RAVENNA*. 2013. URL: http://ravenna.alcnetworx.com/ (cit. on p. 10).

[13k] *Wikipedia - Simulink*. July 12, 2013. URL: https://en.wikipedia.org/wiki/Simulink (cit. on p. 12).

[13l] *ZeroC*. 2013. URL: http://www.zeroc.com/ (cit. on p. 10).

[14] *CMake - Cross Platform Make*. 2014. URL: http://www.cmake.org (cit. on p. 16).

[AA05] P. Arumı and X. Amatriain. "CLAM, an object oriented framework for audio and music." In: *LAC2005 Proceedings* (2005), p. 43 (cit. on p. 7).

[Ama04] X. Amatriain. "An Object-Oriented Metamodel for Digital Signal Processing." In: (2004) (cit. on p. 7).

[BS06] J. Blanchette and M. Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006 (cit. on p. 15).

[Cou+11] G. Coulouris et al. *Distributed Systems: Concepts and Design, 5th Ed.* Addison-Wesley, 2011 (cit. on p. 9).

[CRW04] G. R. Clarke, D. Reynders, and E. Wright. *Practical Modern SCADA Protocols: DNP3, IEC 60870.5 and Related Systems*. Newnes, 2004 (cit. on p. 81).

[Dob+14] K. Dobbler et al. "Vibroacoustic Monitoring: Techniques for Human Gait Analysis in Smart Homes." In: 6. AAL Kongress. Ed. by R. Wichert and H. Klausing. Berlin: Springer Verlag, 2014. ISBN: 978-3-642-37987-1 (cit. on p. 77).

[ES10] K. Eilebrecht and G. Starke. *Patterns kompakt 3rd. Ed. Entwurfsmuster für effektive Software-Entwicklung*. Spektrum, Akademischer Verlag, 2010 (cit. on p. 68).

[Fou+12] T. Fountain et al. "The open source dataturbine initiative: empowering the scientific community with streaming data middleware." In: *Bulletin of the Ecological Society of America* 93.3 (2012), pp. 242–252 (cit. on p. 7).

[Gau02] G. Gautschi. *Piezoelectric sensorics: force, strain, pressure, acceleration and acoustic emission sensors, materials and amplifiers*. Springer, 2002 (cit. on p. 37).

[GRR11] F. Graf, G. Rattei, and G. Ruhdorfer. "Giving tunnels ears – installation of the first acoustic monitoring system for road tunnels worldwide." In: *Internationaler Tunnelkongress, Hamburg* (May 2011) (cit. on p. 80).

[GSS02] R. P. Garg, I. A. Sharapov, and I. Sharapov. *Techniques for optimizing applications: high performance computing*. Prentice Hall PTR, 2002 (cit. on p. 33).

[Hen04] M. Henning. "A new approach to object-oriented middleware." In: *Internet Computing, IEEE* 8.1 (2004), pp. 66–75 (cit. on p. 11).

[HS01] Y. Hardy and W.-H. Steeb. "Finite State Machines." In: *Classical and Quantum Computing*. Springer, 2001, pp. 229–250 (cit. on p. 33).

[HS11] M. Henning and M. Spruiell. "Choosing Middleware: Why Performance and Scalability do (and do not) Matter." In: (2011) (cit. on p. 13).

[HW03] G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2003 (cit. on p. 45).

[Ret+10] B. Rettenbacher et al. "A pilot system for acoustic tunnel monitoring." In: *Proceedings of 1st EAA EuroRegio Congress of Sound and Vibration* (2010) (cit. on p. 80).

[Ret+11] B. Rettenbacher et al. "Interactive Sound Source Localization using Natural User Interfaces and Mobile Devices." In: *Forum Medientechnik - Next Generation, New Ideas*. Ed. by A. Frotschnig and H. Raffaseder. 2011 (cit. on p. 78).

[SG05] M. Steiner and F. Graf. "Akustisches Tunnelmonitoring." In: 3. *Internationaler Fachkongress - Verkehr und Sicherheit in Straßentunneln, Hamburg*. 2005 (cit. on p. 80).

[Tan02] A. Tanenbaum. *Computer networks 4th ed.* Pearson Education), 2002 (cit. on p. 9).

[Tan09] A. Tanenbaum. *Modern operating systems 3rd ed.* Pearson Education, Inc., 2009 (cit. on p. 3).

[The99] Inc. The MathWorks. *Real-Time Workshop. For Use with SIMULINK*. Jan. 1999 (cit. on p. 18).

[VKZ05] M. Völter, M. Kircher, and U. Zdun. *Remoting Patterns. Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley and Sons Ltd, 2005 (cit. on p. 10).

[Xen99] C. Xenophontos. *A Beginner's Guide to MATLAB*. Technical Report. Clarkson University, 1999 (cit. on p. 38).

Appendix

Appendix A.

Configuration Examples

A.1. Simulink Module Channel Configuration Example

Listing A.1 Complete Channel Configuration of a *Simulink Module Data Transducer* used for feature extraction

```
<ChannelConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <IndexDefinition name="chIdx">0:15</IndexDefinition>
  <IndexDefinition name="mfccIdx">1:12</IndexDefinition>
  <Request>
    <ForEach index="chIdx">
      <Channel>
        <GlobalID>Test_C[chIdx]</GlobalID>
        <DataType>AUDIO</DataType>
        <ChannelSpecifics type="TimeSeries">
          <SampleRate>44100</SampleRate>
          <SampleFormat>INT16</SampleFormat>
          <FrameSize>4096</FrameSize>
          <StreamFormat>PCM</StreamFormat>
        </ChannelSpecifics>
        <RequestParameters>
          <Parameter name="ChannelPath" xsi:type="String">0</Parameter>
        </RequestParameters>
        <ProcessingParameters>
          <Parameter name="Port" xsi:type="String">AudioIn</Parameter>
        </ProcessingParameters>
      </Channel>
      <Channel>
        <GlobalID>Test_C[chIdx]</GlobalID>
        <DataType>AUDIO</DataType>
        <ChannelSpecifics type="TimeSeries">
          <SampleRate>44100</SampleRate>
          <SampleFormat>INT16</SampleFormat>
          <FrameSize>4096</FrameSize>
          <StreamFormat>PCM</StreamFormat>
        </ChannelSpecifics>
        <RequestParameters>
          <Parameter name="ChannelPath" xsi:type="String">1</Parameter>
        </RequestParameters>
        <ProcessingParameters>
          <Parameter name="Port" xsi:type="String">AudioIn</Parameter>
        </ProcessingParameters>
      </Channel>
    </ForEach>
    <RequestParameters>
      <Parameter name="RtSimulation" xsi:type="Int32">1</Parameter>
    </RequestParameters>
  </Request>
  <Provide>
    <ForEach index="chIdx">
      <ForEach index="mfccIdx">
        <Channel>
          <GlobalID>MFCC[mfccIdx]_C[chIdx]</GlobalID>
          <DataType>CUSTOM</DataType>
          <LocalDescription>
            <Parameter name="Port" xsi:type="String">Mfcc[mfccIdx]</Parameter>
          </LocalDescription>
        </Channel>
      </ForEach>
      <Channel>
        <GlobalID>ZCR_C[chIdx]</GlobalID>
        <DataType>CUSTOM</DataType>
        <LocalDescription>
          <Parameter name="Port" xsi:type="String">Zcr</Parameter>
        </LocalDescription>
      </Channel>
    </ForEach>
  </Provide>
</ChannelConfig>
```

A.2. IceGrid Application Example

Listing A.2 IceGrid configuration for GainPan16 application

```
<!—Chronos Realtime Test Application: GainPan—>
<icegrid>
    <application name="GainPan">
        <node name="TestMaster">
            <server id="CXR" activation="always" exe="C:\Users\fim\Projekte\svn\
                Chronos\Realtime\trunk\cpp\bin\Release\CXRegistry.exe" pwd="C:\Users\
                fim\Projekte\svn\Chronos\Realtime\trunk\tests\gainpan\resources">
        <option>—c</option>
        <option>C:\Users\fim\Projekte\svn\Chronos\Realtime\trunk\tests\gainpan\config
            </option>
                <properties>
                    <property name="Ice.ThreadPool.Client.SizeMax" value="5"/>
                    <property name="CXRAdapter.ThreadPool.Size" value="3"/>
                    <property name="CXRAdapter.ThreadPool.SizeMax" value="15"/>
                </properties>
                <adapter name="CXRAdapter" endpoints="tcp" id="CXRAdapter">
                    <object identity="CXR" type="::chronosX::CXRegistry"/>
                </adapter>
            </server>
      <server id="CU" activation="always" exe="C:\Users\fim\Projekte\svn\Chronos\
          Realtime\trunk\cpp\bin\Release\AudioCaptureUnit.exe" pwd="C:\Users\fim\
          Projekte\svn\Chronos\Realtime\trunk\tests\gainpan\resources">
                <property name="CU.ThreadPool.Size" value="3"/>
      </server>
      <server id="SLM" activation="always" exe="C:\Users\fim\Projekte\svn\Chronos\
          Realtime\trunk\cpp\bin\Release\SimulinkModule.exe" pwd="C:\Users\fim\
          Projekte\svn\Chronos\Realtime\trunk\tests\gainpan\resources">
                <property name="CU.ThreadPool.Size" value="2"/>
      </server>
      <server id="AM" activation="always" exe="C:\Users\fim\Projekte\svn\Chronos\
          Realtime\trunk\cpp\bin\Release\AudioMonitor.exe" pwd="C:\Users\fim\Projekte
          \svn\Chronos\Realtime\trunk\tests\gainpan\resources">
                <property name="CU.ThreadPool.Size" value="2"/>
      </server>
      <server id="GUI" activation="always" exe="C:\Users\fim\Projekte\svn\Chronos\
          Realtime\trunk\cpp\bin\Release\ControllerGui.exe" pwd="C:\Users\fim\
          Projekte\svn\Chronos\Realtime\trunk\tests\gainpan\resources">
      </server>
    </node>
    </application>
</icegrid>
```