Lecture Notes for the Course

Object Oriented Software Development for Control Engineering and Signal Processing Applications

Dr.–Ing. Heiko Hengen

Lehrstuhl für Regelungstechnik und Signaltheorie

Fachbereich Elektrotechnik und Informationstechnik
Universität Kaiserslautern

Kaiserslautern, January 2001

Contents

Chapter 1

Introduction

1.1 Structure of problems considered in this course

A very common setting in a pure control system engineering problem is a plant with a number of terminals (analog I/O). Depending on the speed of the actual plant, one has to select the interface in between plant and IPC. The general informational setup of a basic control problem is given in Fig. 1.1.

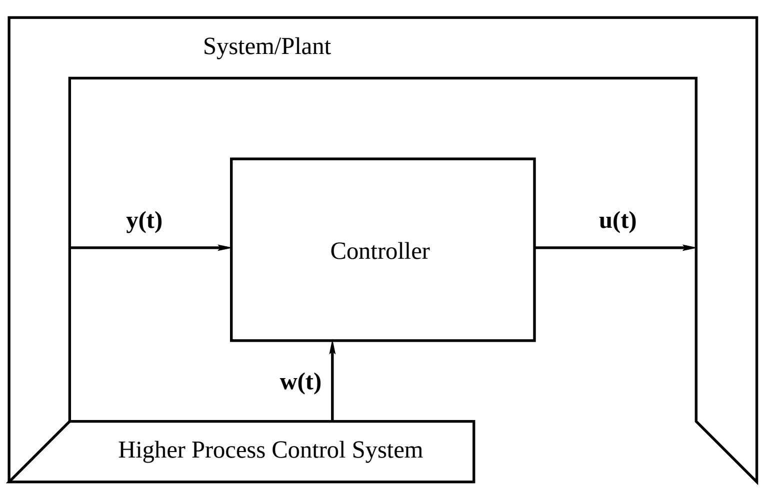


Figure 1.1: Informational Structure of a control system problem

In case the overall process control system is already included in the CPU on which the controller is run, then the system can be represented as given in Fig. 1.2
It is also necessary to classify the problem hierarchically according to the structures found in the environment of the process. In general, process management can be represented as a layer model.
The given Model shows that it is necessary to take different levels of information into account. Devices for direct process control most often have to be designed for serving process information to higher level control devices. (This is beyond the scope of this course and will not be dealt with in detail).

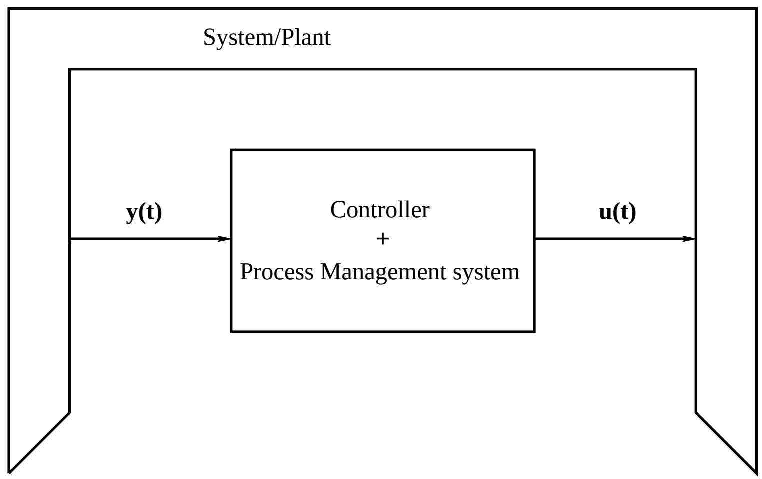


Figure 1.2: Informational Structure of a control system problem with included process management system

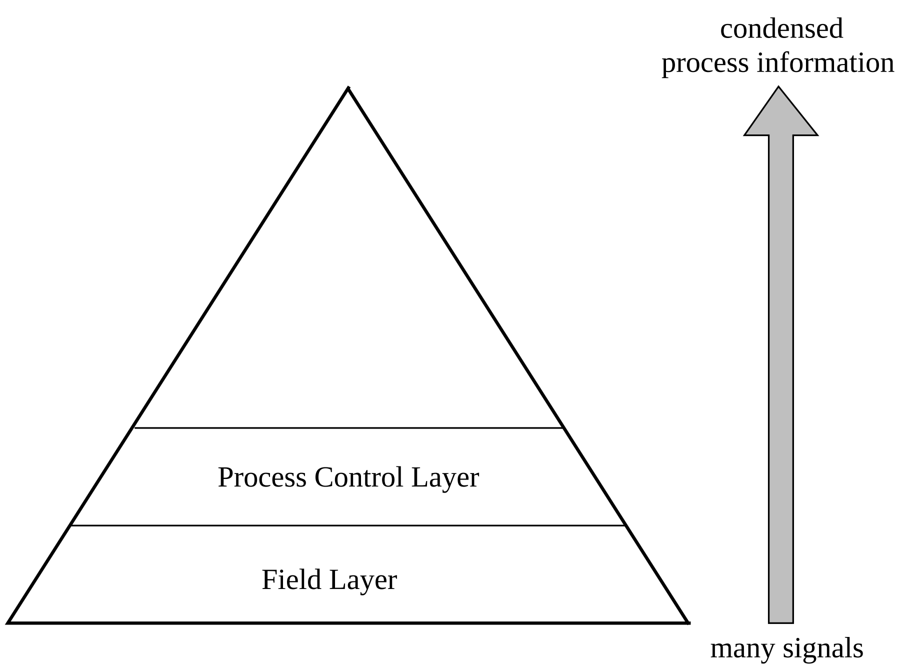


Figure 1.3: An informal layer model for automization

In this course, the focus is set on control-specific design including

- Rapid prototyping issues
- user interfacing
- a concise concept of building up software packages according to modern standards

- real time issues
- avoiding interpretative structures using object oriented techniques

1.2 Aim of the course

The aim of this course is to teach concepts used in the development of software systems used for control applications. The prejudice that engineering and programming do not belong together most often leads to problems in the implementation stage. Neglecting programming issues is, at least in our days as bad as starting any technical project without prior planning - e.g. designing a controller without using any mathematics.

1.3 Basic Knowledge required

For successful understanding and application of the concepts given in this course, it is advantageous to

- be familiar with a programming language
- have basic control engineering capabilities
- have a basic understanding of systems and modeling of systems

Chapter 2

An overview of programming languages, techniques and operating systems

2.1 Programming Languages

The following course will not focus on a special language, although the examples will be done using object oriented C++; there are of course many other possibilities, such as object oriented Pascal (Delphi); the given concepts can be even realized using any assembly language. C++ has been selected as it provides a good compromise between speed and programming effort: in many systems computational power is not the limiting parameter whereas time of development is very costly. This course gives a structural approach.

2.2 Operating System

Today, a big variety of operating systems can be found. Not all of them are meant for industrial automation although they are sometimes used for automation issues.

The question of which operating system one has to use cannot be decided a priori easily. The reason for this is that people use what they know and also use systems for which company-support is granted. If you choose an operating system, choose it according to the requirements of the application. Take also into consideration, that old operating systems are of no use if an open and extensible system structure has to be built up.

Concerning realtime requirements, one has to be aware of the lack of realtime capabilities in many widely used office-application operating systems. If the application has to be realized on such a platform, it is necessary to ensure realtime capabilities using either realtime extensions (which may not grant hard realtime) or to add an additional hardware layer (known as "embedded system").

The two competitors which offer a graphical user interface and are up to date according to their architecture are Windows and Linux.
Although Linux is not yet widely used as far as industrial applications are concerned, it becomes more and more common and also has good realtime extensions. Another advantage is its stability and the number of free compilers and extensions which are most often distributed in the form of source code.

Dos Applications using multitasking environments such as RTKernel are also a good choice if the necessary hardware-drivers and libraries are available.

The best choice is however to use multiplatform toolkits which allow the developer to cvhange the target-system without any adaptions concerning the program code.
In this course, we choose Linux/C++ and use QT as an example for a multiplatform visualization and User-Interface design toolkit. For realtime issues, we choose RTLinux.

The design techniques can be easily transferred to other common platforms or toolkits as well.

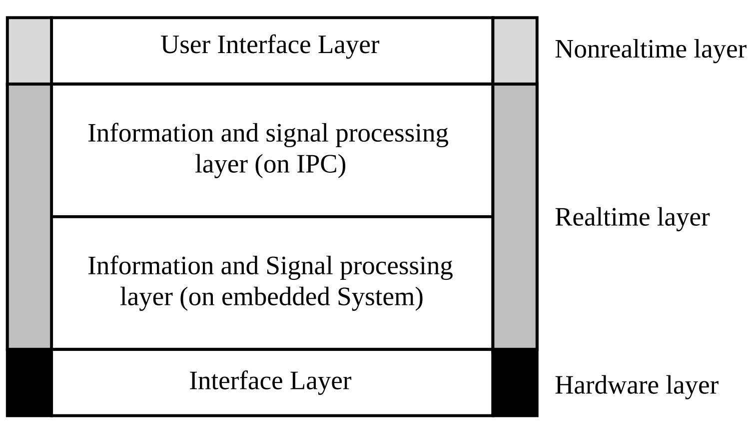


Figure 2.1: User interface-interaction with realtime layers

2.3 Platforms

The choice of a platform for realization of e.g. a control algorithm is also an important question, especially with regard to reliability issues and failure probabilistic approaches.
In most cases, PLC- programs on PLC-systems are preferred to Industrial PC

(IPC) with embedded cards or even as standalone control devices.
In fact, PLC-systems are a good choice as far as they cover the requirements of modern control systems.
At least for rapid prototyping applications, the IPC is a good choice. Concerning flexibility and the possibility to test modern control algorithms, the IPC plays an important role.

2.4 User Interface Issues

A user interface is also part of the concept of a control engineering or signal processing software application just as well as a pressure gauge is part of an industrial valve. Building a user interface is not a highly scientific task but one has to take into account that without an appropriate user-interface-layer, a system for rapid prototyping and industrial control would neither be accepted nor be easy to handle.

For building up user-interfaces, many user-interface kits (class or function libraries respectively) are available. Most of todays user interfaces are graphical. The user-interface libraries can be described according to their underlying structural concept and their abilities as follows:

1. Object oriented (e.g. QT, GTK)
2. Flow-oriented (e.g. Motif)
3. Platform oriented or available for different Platforms (e.g. Linux and Windows)

Chapter 3

Techniques of modeling software systems

3.1 Dimensions of system modeling

In general, two types[1] of system models can be distinguished:

1. Behaviouristic models, giving the specified or actual system-behaviour at the inputs/outputs of the system

2. structural models, these models specify the structure of a system

3.2 Flow diagrams and Petri Nets

The relation between Flow diagrams and Petri Nets will be elaborated using a simple example; the task is specified as follows: "If the ambient light intensity reduces under a certain level, then switch on a bulb, if the intensity of the ambient light is above a certain level, switch off the light. Assume, that I_2 sends out a signal if light intensity $> \alpha$ and I_1 sends out a signal if light intensity $< \beta$.

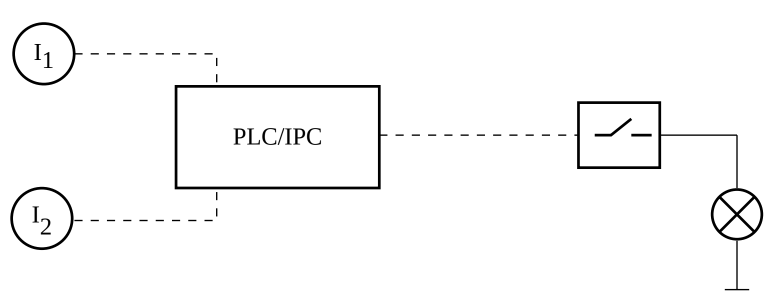


Figure 3.1: Setup with light intensity sensors and bulb

[1]Litz [9] for example distinguishes three type of models in a more general scope of system modeling

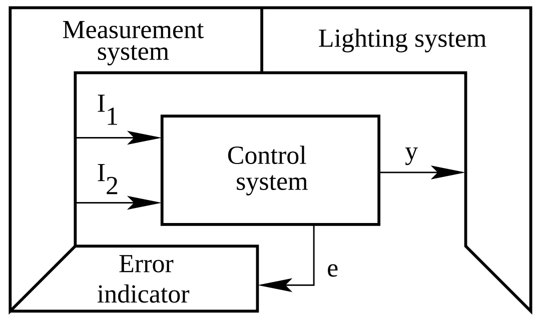


Figure 3.2: Informational structure of the setup given in Fig. 3.1

The given behavioural specification has to be formalized to obtain an algorithm. To ensure good extensibility, the algorithm has to be transparent.

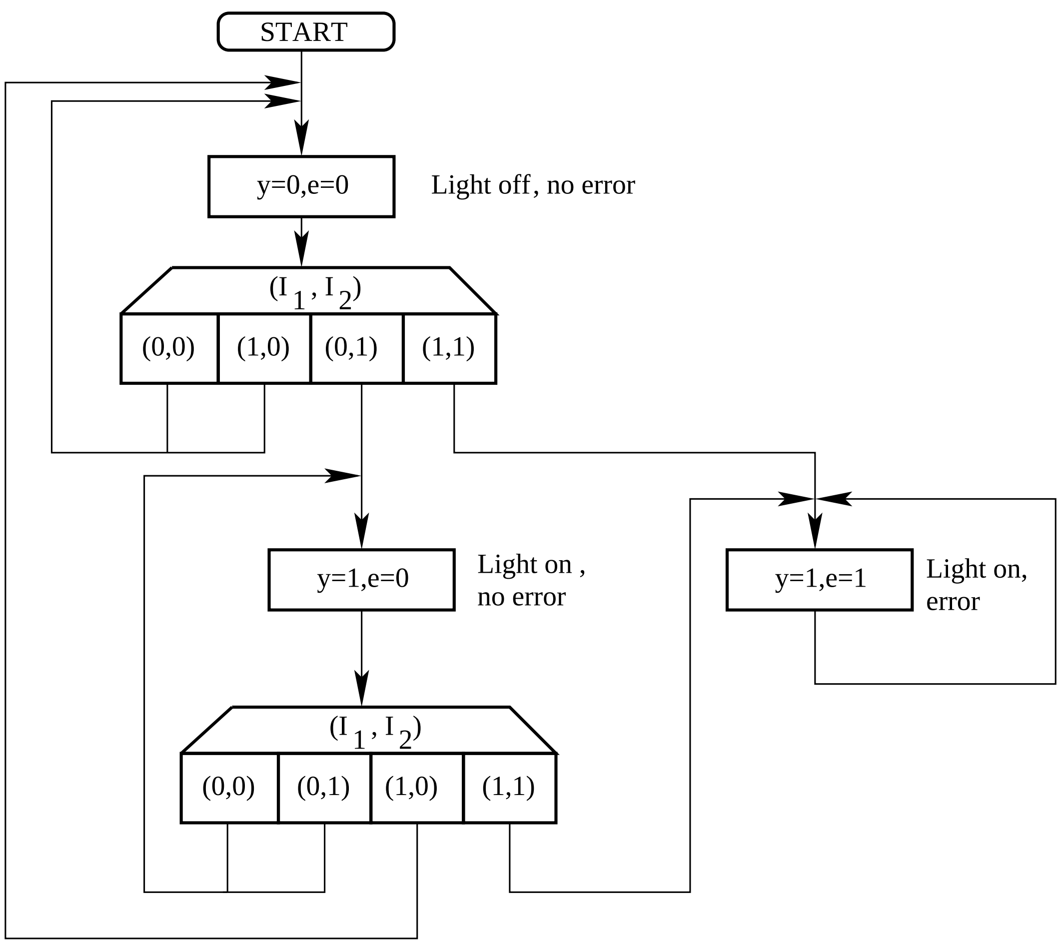


Figure 3.3: Setup with light intensity sensors and bulb

The algorithm given in Fig. 3.3 could be either used to write a program or to design a finite state machine of Moore-type. Alternatively one could use the fol-

lowing Petri net[2] to make a model of the system's desired behaviour:
From basic informatics, one knows the flow diagram for program routines. The flow diagram gives an overview of causality and - in general - shows under which conditions (system inputs) which parts of the program are executed. In order to predetermine timings in certain situations, flow diagrams can be used.

Although flow diagrams are widely used, we will use a special method of modeling processes in process control - the so called Petri nets. The advantage of Petri nets over flow diagrams is that they make concurrencies obvious and thus are a more transparent way of behavioural process analysis and controller synthesis than flow diagrams.

However, one dimension along which it is necessary to analyse or synthesize a program (an algorithm) is time.
The second dimension which one has to take care of is the structure of the designed program.

Talking about the structure includes questions like reusability of code, methods for synthesis, validation, etc. A program which is operational and fulfills its tasks within a timeframe might as well be built up in a very unstructured way. The reverse is also possible, - e.g. that a program is well structured with reusable code but not fulfilling realtime requirements.

The optimal solution is of course a program which suits both the requirements: suitable timing and structural clarity as well as reusability of code.

Petri nets are well suited to show concurrencies in processes. Take for example a drilling process. Consider the following requirements for a part of a drilling process: switch on drill engine, then lower the drill quickly. In the vicinity of the metal, slow down the axial movement of the drill, drill into the metal, after the hole is complete, move the drill upwards in axial direction and switch off the drill engine.

In the interpreted Petri net of figure 3.5, we notice that outputs towards the process are connected with the places and inputs to the (logic) controller are used to determine the switching of the transitions (this is called "control oriented interpretation" according to Litz [7]).

Obviously the states of the Petri net are given by all the tokens in the different places of the net.

It is also possible to set up models for events and series of events using Petri nets.

[2]Petri net see Appendix A

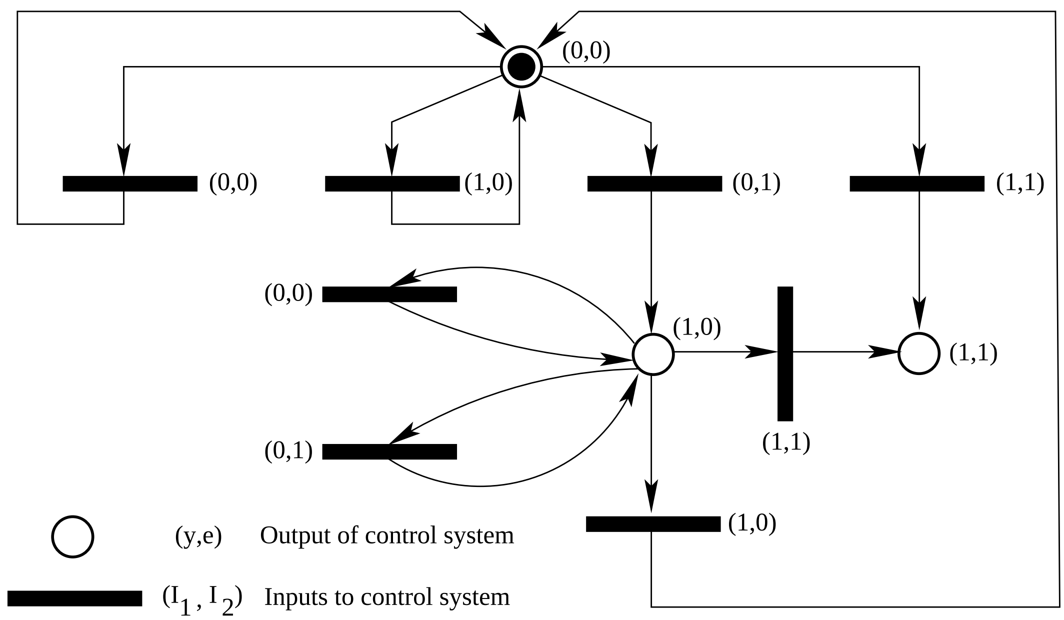


Figure 3.4: A behaviouristic Petri Net for visualization of the control algorithm

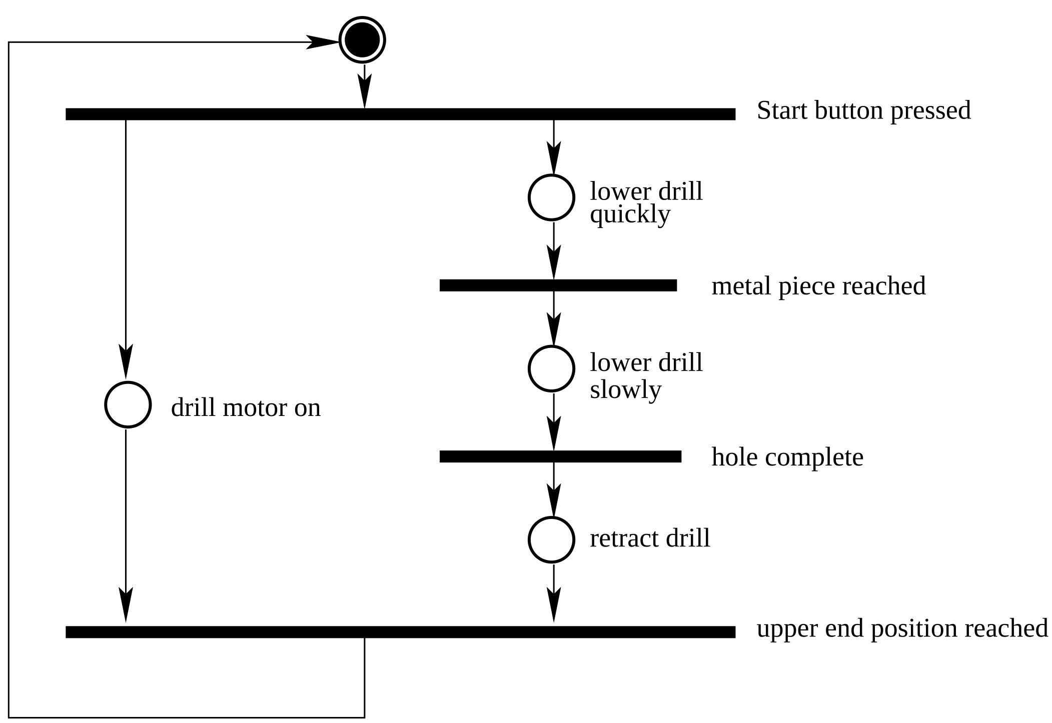


Figure 3.5: A Petri net for a part of a drilling process (control oriented interpretation)

This is especially useful if behaviouristic models of communicating entities are set up.

As far as Petri nets are concerned, one frequently encounters different kinds in literature (see Wendt [17] or Schnieder [14]). The reason for this diversity is the different interpretations of the elements of the net.

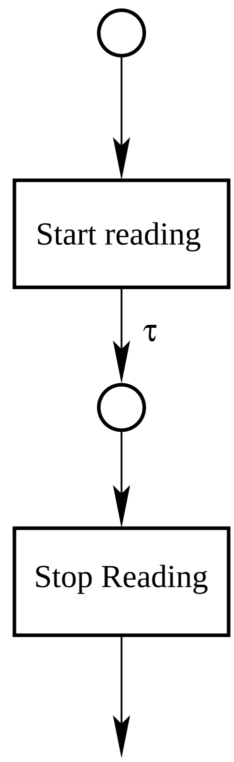


Figure 3.6: A Petri net can be considered as a means of visualization of flows, in this case the time consumption of one step in the flow is described by the τ at the post arc of the transition. Each step contains a beginning and an ending transition and the arcs in between represent the time consumption of the step.

One often encounters Petri nets as a modern type of flow diagram. To become familiar with this modern and transparent type of tool for system analysis and design, we will also use this means during the course[3]. A Petri net interpreted as a flow diagram is shown in Fig. 3.6.

3.3 Object Oriented Development

Till now, we were discussing the modeling of algorithmic behaviouristic aspects of software systems. A Petri net for example does not give any hints about the overall structure of a software system, it tells us about how certain algorithms

[3]See Litz [7], Wendt [17] and Schnieder [14] for a good overview of the application of Petri nets as a means of system modeling in process control, digital system and with further extensions also for analog system modeling

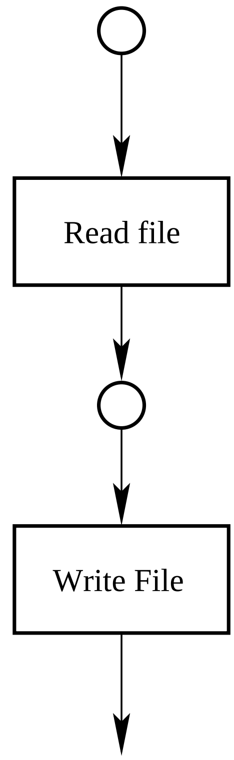


Figure 3.7: A Petri net with another interpretation: every transition represents the execution of a function (procedure) in a process

work. In a flow-oriented software , the flow-oriented modeling tools (e.g. flow diagrams or Petri nets) contains all the necessary information of how an algorithm works and how it can be implemented by conversion of the flow diagram into code.
The object oriented approach contains in addition to flows a structural component - but before we investigate the necessary tools, such as entity block-diagrams, we first take a closer look at object orientation itself.

Object orientation is a frequently used term in modern software development - one problem however is that is most often misinterpreted; object orientation is a concept of how to build up a software system in an efficient and transparent way.

The design concept is to form classes of similar (real-world or software) objects or processes and their handlers - entities that focus on the handling of a certain process or object.

3.3.1 Classes

A class is different from a set because in a set, you will find only selected Elements of a unique type; e.g. consider the set

$$\mathbb{M} = \{1, 2, 3, \ldots, 10\} \quad \mathbb{M} \subset \mathbb{N} \tag{3.1}$$

with $\mathbb{N}$ the set of all natural numbers.

The set consits of elements which are sharply defined; e.g. if you think of trees in a forest, then there is not a unique tree, some of the trees are of a different shape but have certain features in common.

The class of trees contains

$$C_T = \{oaktree, booktree, maple, \ldots, \} \tag{3.2}$$

in the sense that a fuzziness comes into play. As per definition a class is non-sharp, we can introduce hierarchies of specification:

non-sharp	Class of forests	all elements which occur in a forest
	Class of trees	all kinds of trees
	Class of oaks	all kinds of oaks
	⋮	⋮
sharp	Class of red oaks with 25 leaves	all oaks with 25 leaves

The extreme case of a class is again a set (if all features of the elements are fully specified).

Forming classes makes it possible for us to find a complexity structure which subdivides a problem into multiple layers of abstraction and specification.

3.3.2 Object orientation

In the previous section we specified a certain class of objects, objects of one class can however be handled in a similar way and so it is sensible to introducte an entity that has the duty to handle a certain class of objects.

Example: You are designing a system with a number of i/o-boards of the same type that can be removed or inserted into the device.

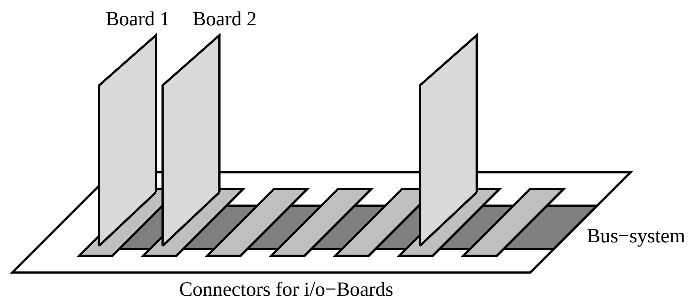


Figure 3.8: I/O board devices

If you have to read or write data from the boards' in- and outputs and you would write a flow-oriented code, you first had to check for every board whether it is at its place and then handle the board's i/o which results in lots of if and case commands and makes even a well structured program code intransparent.

It was much more efficient (for the system designer) if she/he had to integrate only one entity which can be copied and then after the setup each entity handles one i/o-board instead of many if-branches in a flow-oriented code. In fact, we need a higher level entity which sets up the entities, one for every installed board.

In this context, we did not speak about the term class. If we consider the same example a bit extended, we can find out why thinking in different abstraction levels is very useful in the context of object orientation.

Example: Assume the same hardware setup as before with the change that 3 types of i/o-boards can be inserted. First, we should think aout what these boards have in common.
If they have e.g. common identification or communication features, it is not necessary to construct a new entity for every board-handler. We could just extend a basic entity that gives a minimum functionality which is needed for the more specified entities.

Now we already have two abstraction levels:

1. The generic i/o-board handler
2. The specific i/o-board handlers for board types 1,2,3

The selected approach already gives a hint, of which elements an entity must consist and which featues it can have:

1. It should encapsulate a certain functionality, the same as our conventional flow-oriented software has had.
2. It must be possible to inherit the features of a certain type of entity

3. The entity must contain pieces of flow-oriented code as well as data-fields for variables

To extend this map of features let us again go back to the previous example.

Example: Now we allow in addition to the three different i/o-boards also the insertion of boards which are not i/o-boards. Thus we need the possibility to replace certain features in our Entity (so called overriding).

The entities as mentioned previously, are created during run-time of the program; they are built up according to a certain plan which is copied or instantiated in memory.

To fulfill our needs, an entity has to contain:

- membership functions
- membership variables

3.3.3 Object oriented Entity-Block Diagrams

If software synthesis is performed according to an object oriented approach, we also have to use models for such software-systems. The model for object oriented structures is an Entity-Block diagram. An entity consists of storage (blocks of memory) which is normally organized in member variables (members of the entity) Further an entity contains a certain set of so called member functions which operate on the memory or on a certain real world object which is handled by a specific entity.
The predescribed entities exchange information via places which symbolize (in the general case) memory.

How the data is transferred in reality is of no interest - through the places we could either have real data-flow or they could symbolize message passing or even pointer based memory access with a pointer to a certain block of memory in another entity.

Entities are coupled using arcs and places. The arcs describe the data-flow direction in case of unidirectional flow or can also symbolize bidirectional flow (arcs without arrows).
With an object oriented entity-diagram, one can visualize the structure of e.g. a technical plant or a receipe as well as a software module.

To synthesize software structures (this is the second dimension of design of a program), we have to put up some additional rules:

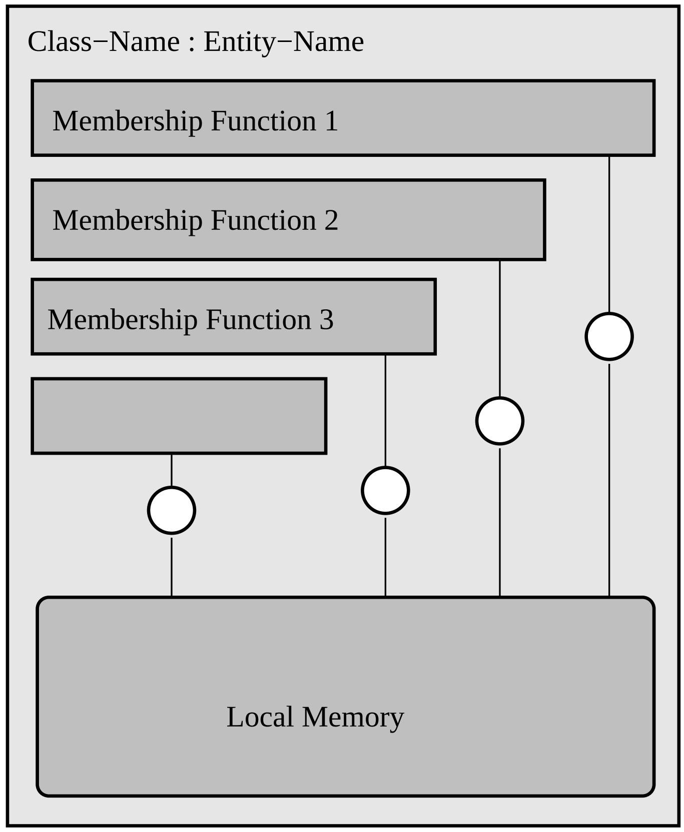


Figure 3.9: An object oriented Entity-Block Diagram

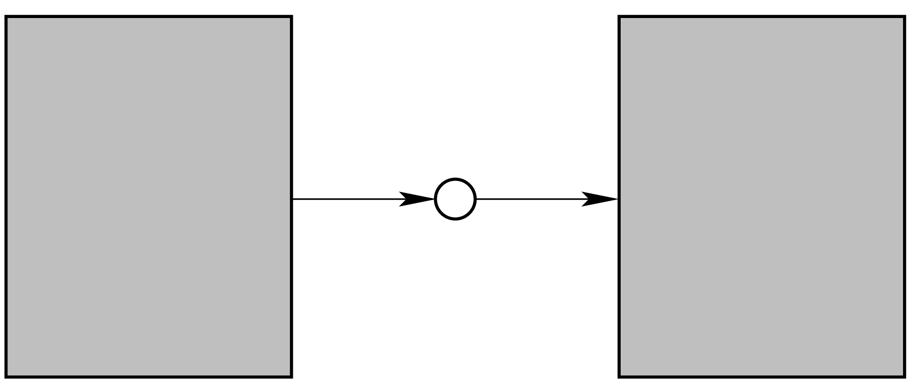


Figure 3.10: Exchanging information

- One entity handles a specific object (real-world or software)
- the entity is connected to another entity using arcs and places as described before
- Hardware components are modeled as entities as well

In addition to the structural approach described, the entity diagrams can be used

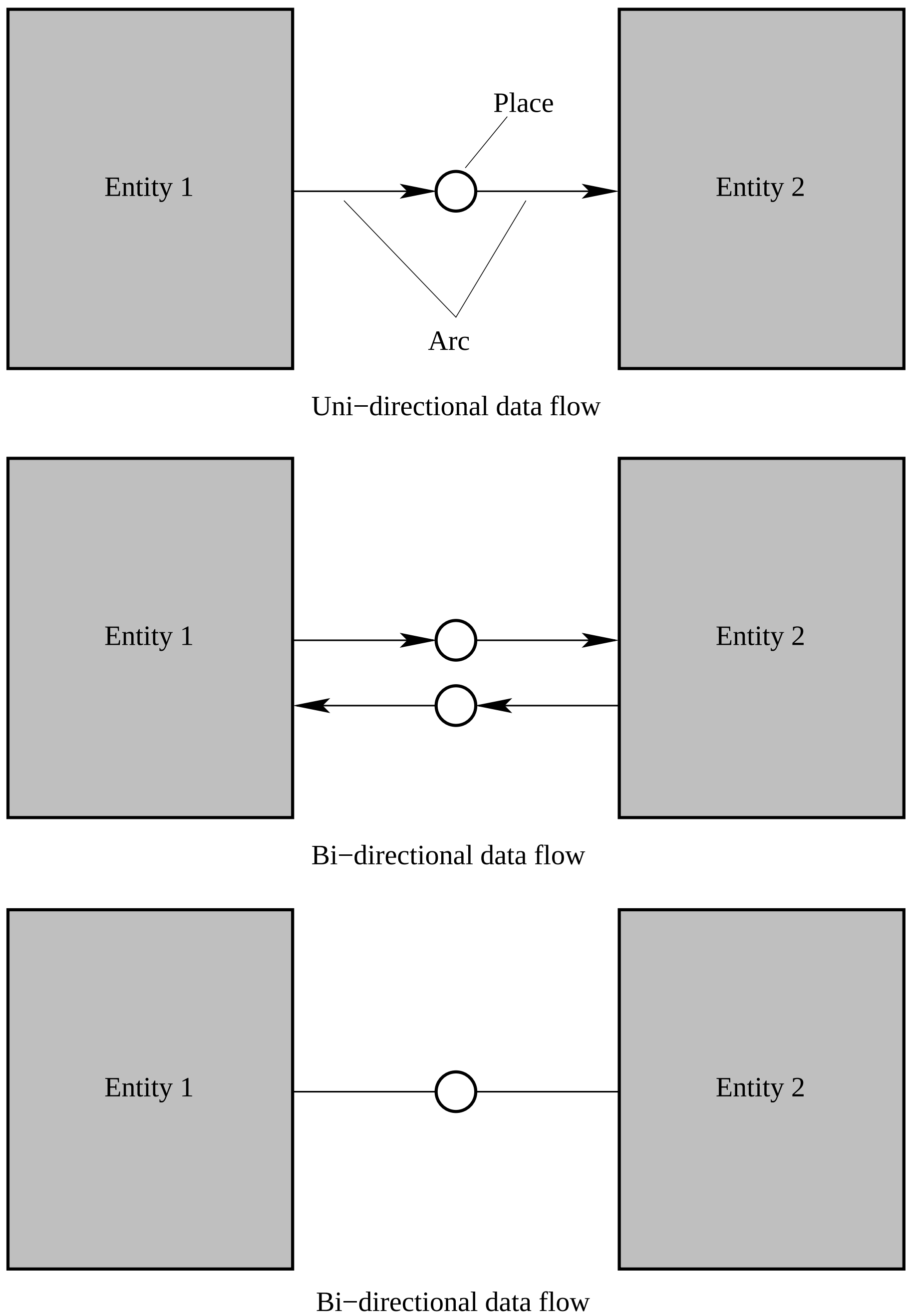


Figure 3.11: Directional Data Exchange between entities

as a tool for determining the complexity level of an entity. This we need for our hierarchical approach to inheritance concepts.

The major difference between a Petri net and an entity block diagram is that the Petri net visualizes actions, reactions and situations and not structures.

Using a combination of Petri nets and entity block-diagrams we can effectively model all kinds of complicated software structures.

3.3.4 Relation of C++-classes to entity block-diagrams

A C++-class is generally speaking an extended type definition which contains member functions as well as member-variables; this is exactly how we have defined the construction of an entity.
The difference between an entity and a class is that a class only gives the plan for how an entity is built up. The class thus defines the structure of an entity.

To realize an entity, a class is instanciated, i.e. an entity is built up according to the definition of the class.

Example: We define the class i/o-Board:

```
class ioboard
{
  //Construction and Destruction of the class
  ioboard();
  ~ioboard();

  //Initialization of Board
  void initbrd();

  //Member-Variables
  int m_type;           //Type of io-board
  int m_buflen;         //Length of buffer

}
```

This class is the plan of how to build up the entity i/o-board. To correctly build up an entity and to initialize the entity appropriately, we need two special functions which are invoked directly after instanciating the class and during deletion of the entity: contructor and destructor respectively.

Using the given concepts leads to a structured approach to software design. One aspect is however that the resulting code has to be ordered concerning execution. The entity is like a musical instrument - the whole functionality is implemented into the entity but it has to be used.

Directing commands, actions and reactions to the block structure of a software consisting of different entities is the task of a higher level system : the so called meta system. The need for a meta system will be illustrated in the following example.

Example: Assume the following setup (e.g. from the user requirement document); A part of a processing plant, a tank with input and output valves and a mixer, has to be controlled according to the following specification: "First open valve V_1 and fill up the tank (without level-indicator, for a time of $\tau_f = 5$s, then stir the contents of the tank for 50s and finally remove the tank's contents by opening valve V_2.

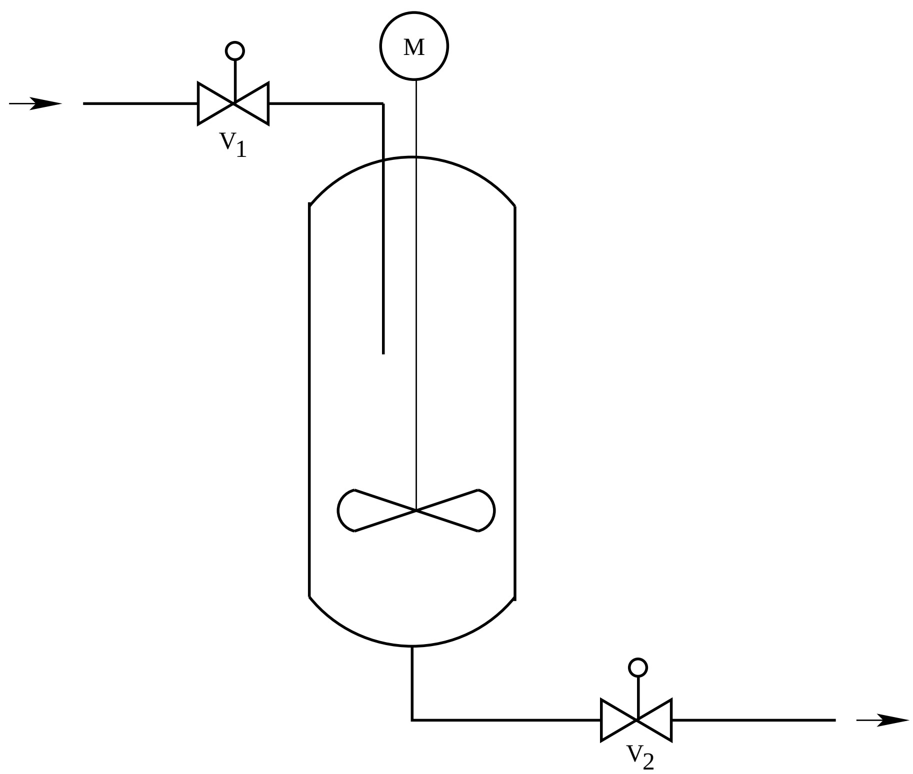


Figure 3.12: A chemical reactor

For every physical object which can be controlled (logic control) we introduce an entity:
With the given entities, we cannot realize the given task unless we use their membership-functions.
To investigate this problem further, we will now take a closer look onto the flow-oriented control algorithm.

The flow-oriented structure now has to be built up from the functionality in the entities. The functions which realize the actions which are written next to the transitions in Fig. 3.14, can be found sorted according to membership to the given entities (see Fig. 3.13). This is the object-oriented concept - the member functions which have to be invoked belong to a certain entity, to the entity which takes care of the underlying physical object (m_V1 takes care of valve 1, m_V2 of

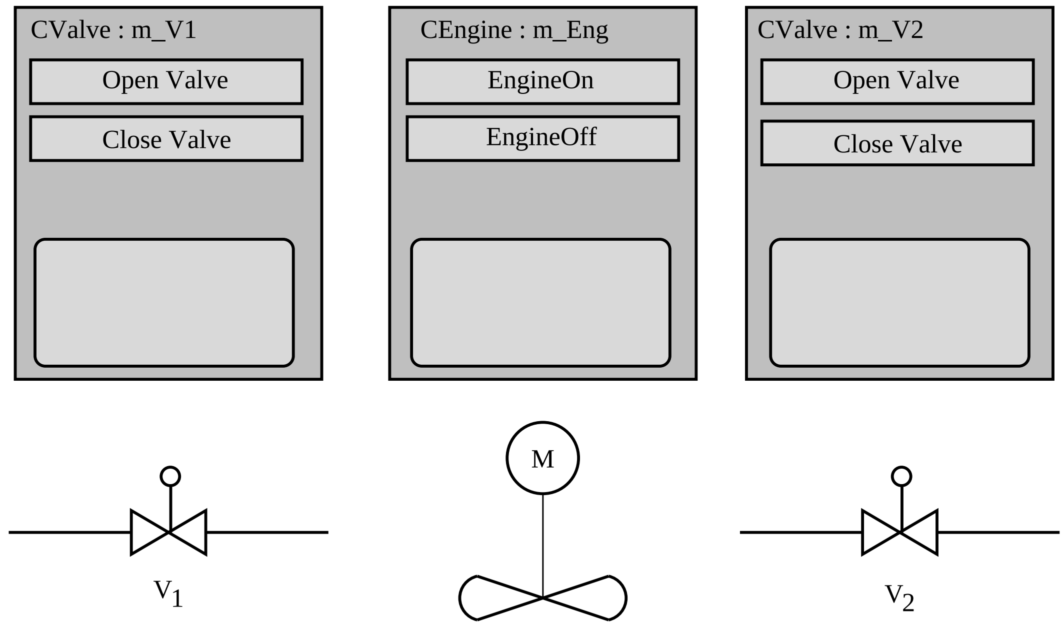


Figure 3.13: Entities for handling the reactor's components

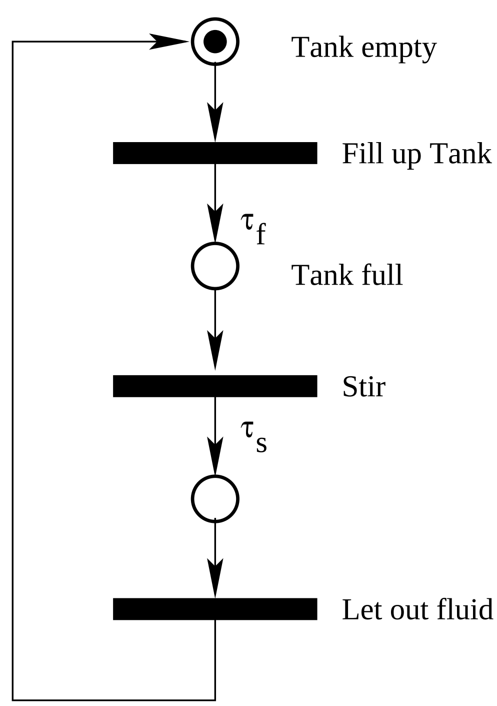


Figure 3.14: A petrinet for the tank process (process oriented interpretation according to Litz [7]

valve 2 and finally m_Eng of the stiring motor).

From this, it becomes clear that we need a higher level system which triggers the membership functions and uses the functionality provided by the entities.

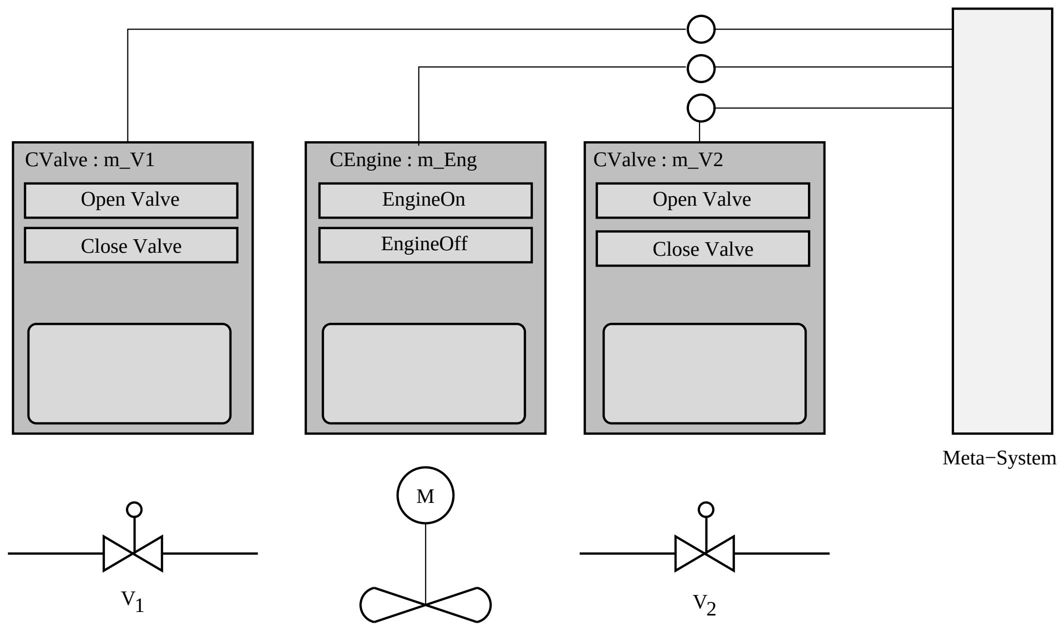


Figure 3.15: The tank system logic control and the meta-system

Through execution of the member functions in the given entities, the meta-system creates the flow-oriented structure depicted by the Petri net in Fig. 3.14. To investigate this further, we look at one member-function of the engine-entity (see Fig. 3.16)
The meta system calls the member functions of the entities to fulfill the process-requirements. Thus it “expands” every entity along the time-axis.

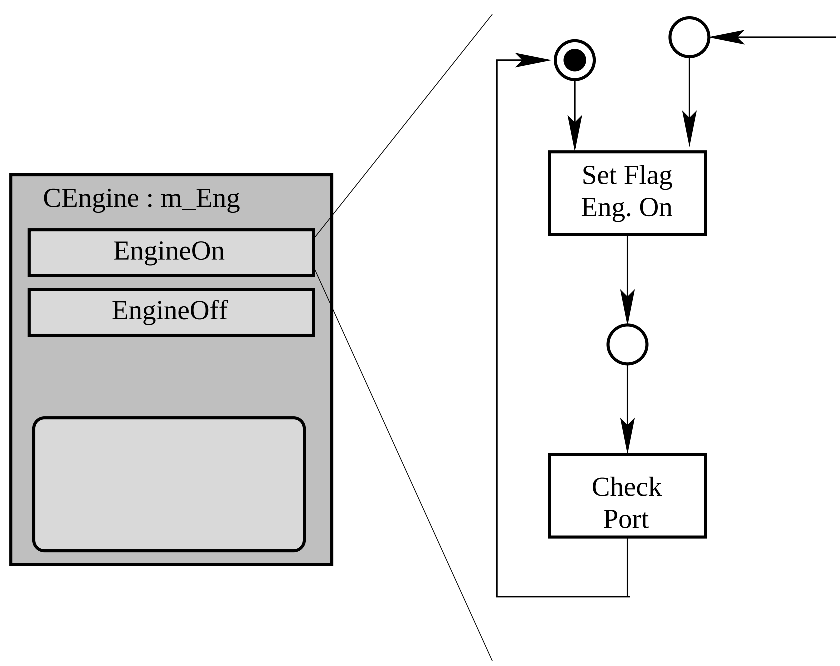


Figure 3.16: One Entity of the tank-system and its flow-oriented components

3.3.5 Building up a Metasystem

A Metasystem is the system which encapsulates smaller systems and exerts administrative influence on its subsystems. The Metasystem takes care of the structure which is built up using the blocks we have discussed in the section above. There must be one component in a complex system which sets up the blocks, generates blocks from plans and fills them with certain sets of parameters.

If we look at it in an administrative manner, one can setup a hierachy: an object oriented software consists of:

1. A *Meta System*
2. Multiple plans for the subsystems (classes)
3. Entities which have been constructed from these plans (classes which have been instantiated on the stack or in the heap)

We said that the meta system coordinates the structural variance in the whole system. In fact, a meta system sets up data-streams to and from the entities and coordinates creation and destruction of these entities.

In general, every block which takes control of other blocks or creates/destroys them is also a meta-system with respect to that block.

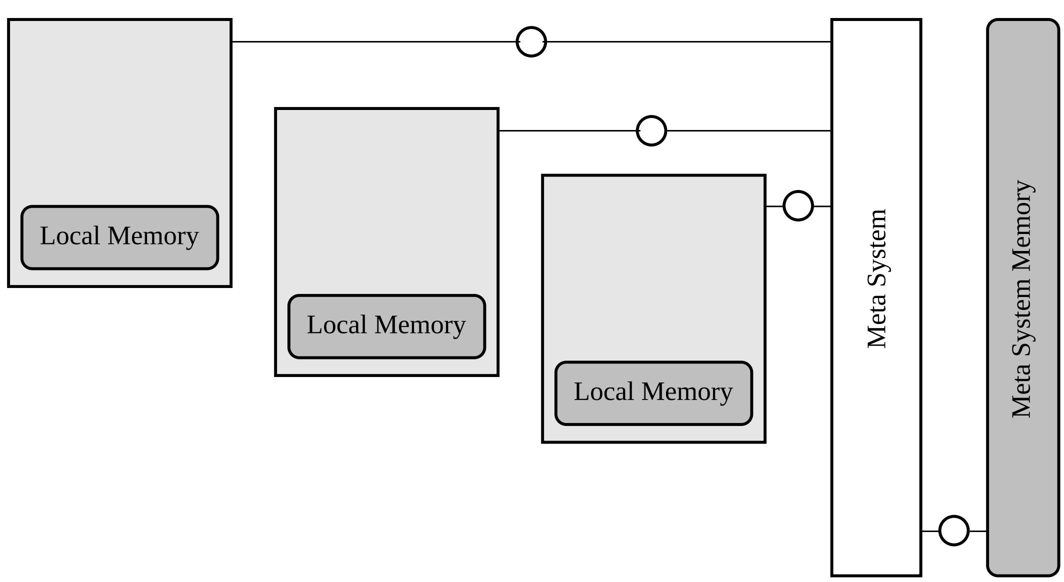


Figure 3.17: A Meta System

The Meta-System also routes events and orders to the blocks. In fact, for our purposes, we do not stick exactly to the concepts given in [17], that all information flow is routed by the meta-system, but that the meta-system sets up the blocks such that they can capture data themselves.

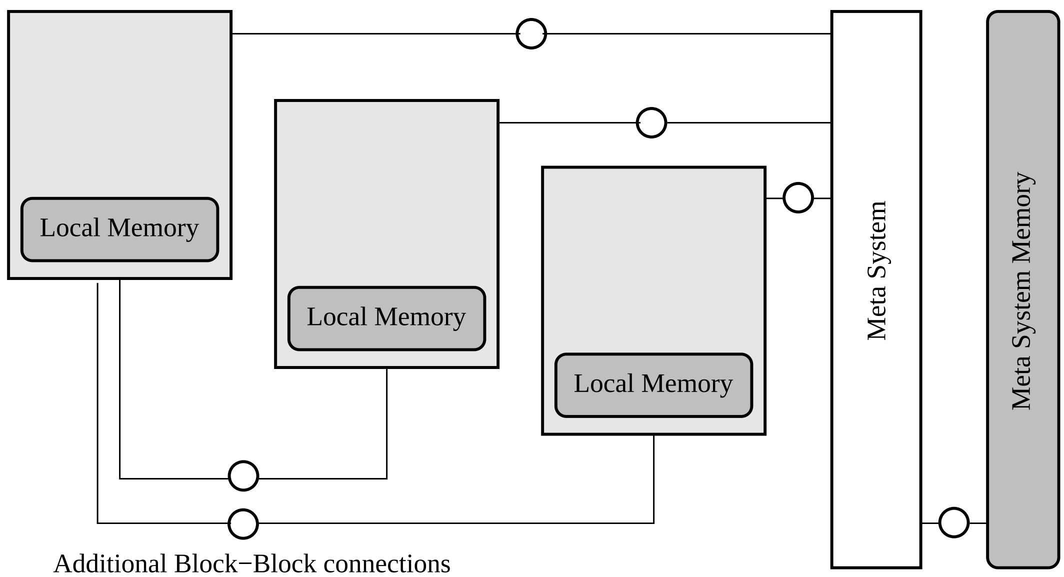


Figure 3.18: A Meta System with information flow between entities

If one considers embedded systems, we could even have two metasystems which communicate using shared memory
From the concepts we discussed till now, we see that the whole design-process can be expressed as a multiresolution problem and there is no unique meta system or unique solution. However, our effort must be concentrated on producing a well operable and well defined structure.

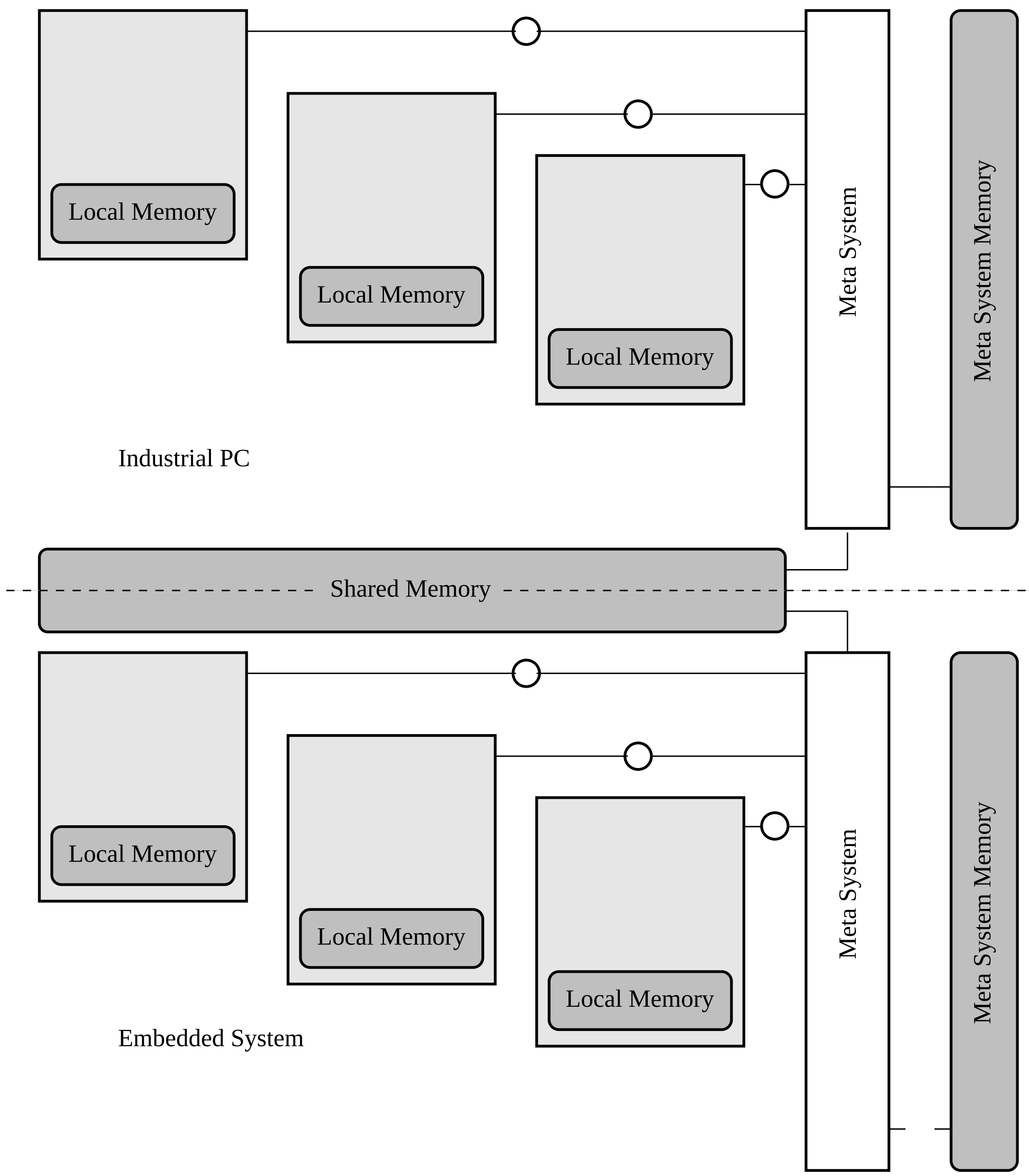


Figure 3.19: Two Meta-Systems in an embedded system

3.3.6 Inheritance and Overriding

From our i/o-Board example, we drew the conclusion, that inheritance is a major feature required for an entity if abstraction or specification level comes into play.

Example: i/o-board handler (continued)

We first build up an entity of which the task is to handle i/o for a specified board type.

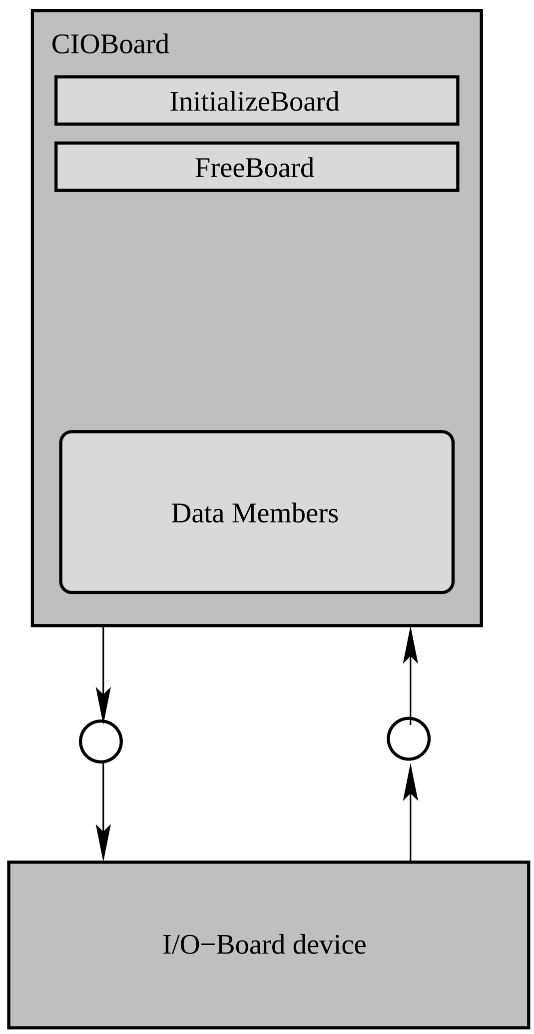


Figure 3.20: The basic i/o-board handling entity

For an extended version of our board, we have to include more functionality. The most convenient way to do so is to inherit the functionality of data and code members (member functions and member variables) from the CIOBoard entity as depicted in Fig. 3.20. This we perform by defining a new class called CExtIOBoard (Class for extended i/o-board functionality) and add all the necessary functionality. The predescribed inheritance is equivalent to assuming the structure given in Fig. 3.21

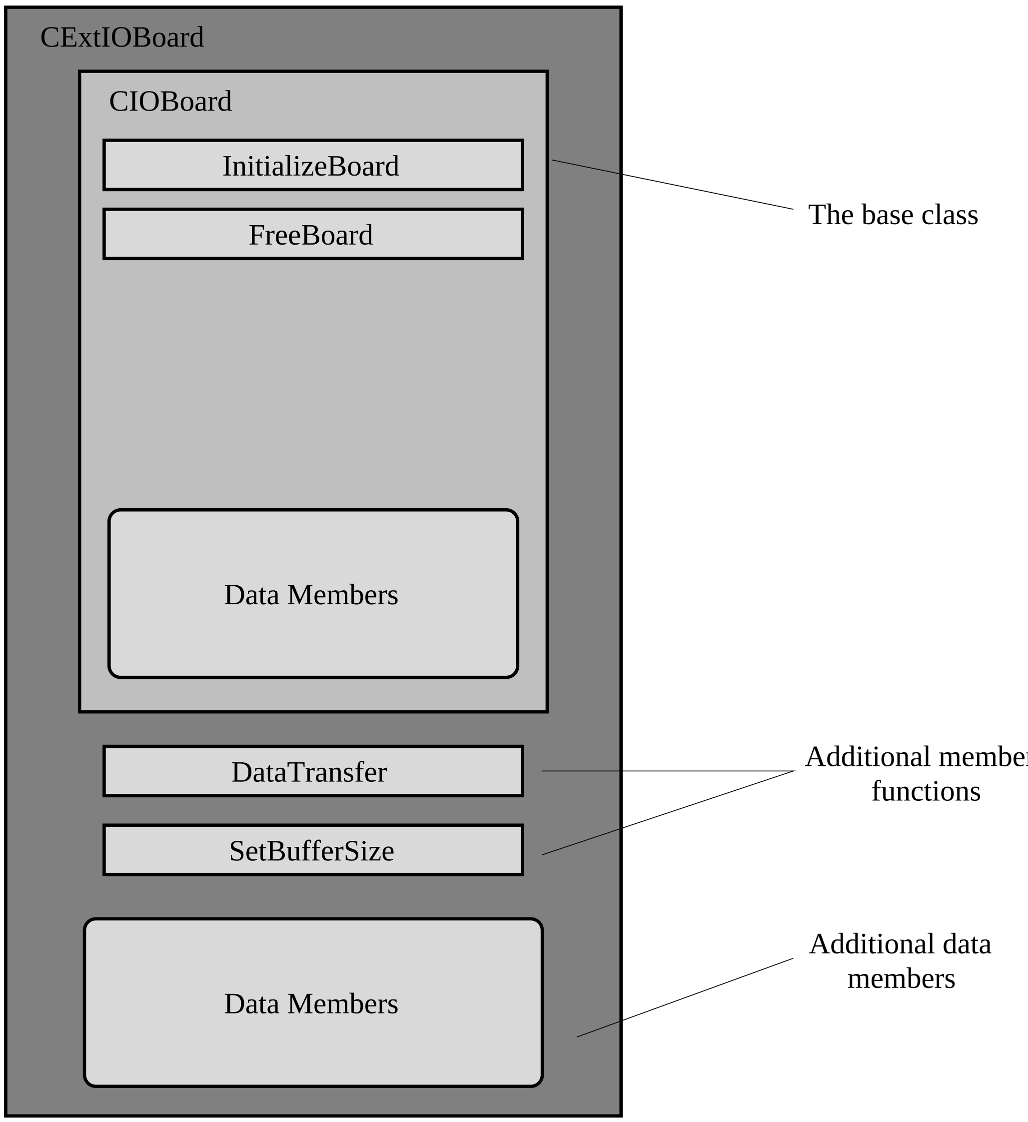


Figure 3.21: The extended entity with more functionality

If we want to determine the whole or at least most of the structure of the classes derived from the base class - in our example we would define a generic board handling entity - then the problem is, that in a more specific derived class, some or even all of the inherited functions have to be replaced or extended.

To overcome this problem, we define so called overridables. Overridables are virtual functions which have a body and are declared as a normal function. Their redeclaration in the derived classes leads to a non-execution of the base classe's implementation of the function but of the one declared in the derived class. Still, their code is present and the overridables can be called by explicitly calling the

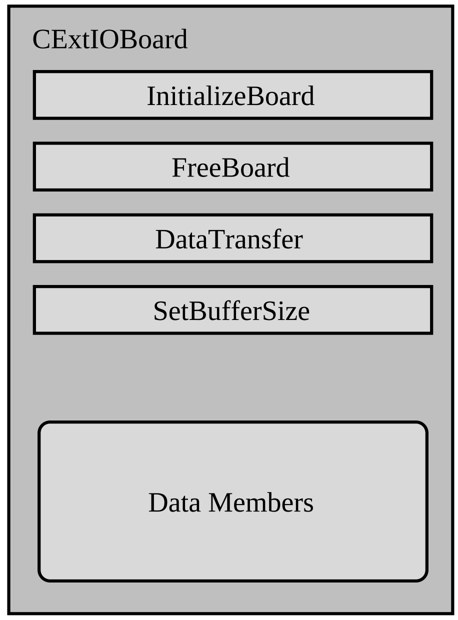


Figure 3.22: Outside view of the entities capabilities

base-class.

Example: For a special i/o-board, we need special code for initialization
Now we want to conserve the code which is present in the base class for initialization of the board. First we want the overriding function to be called and further the base class' function as well. This is depicted in Fig. 3.25 The mechanisms overriding and inheritance are especially important for rapid prototyping applications and object oriented replacement for classical interpreters.

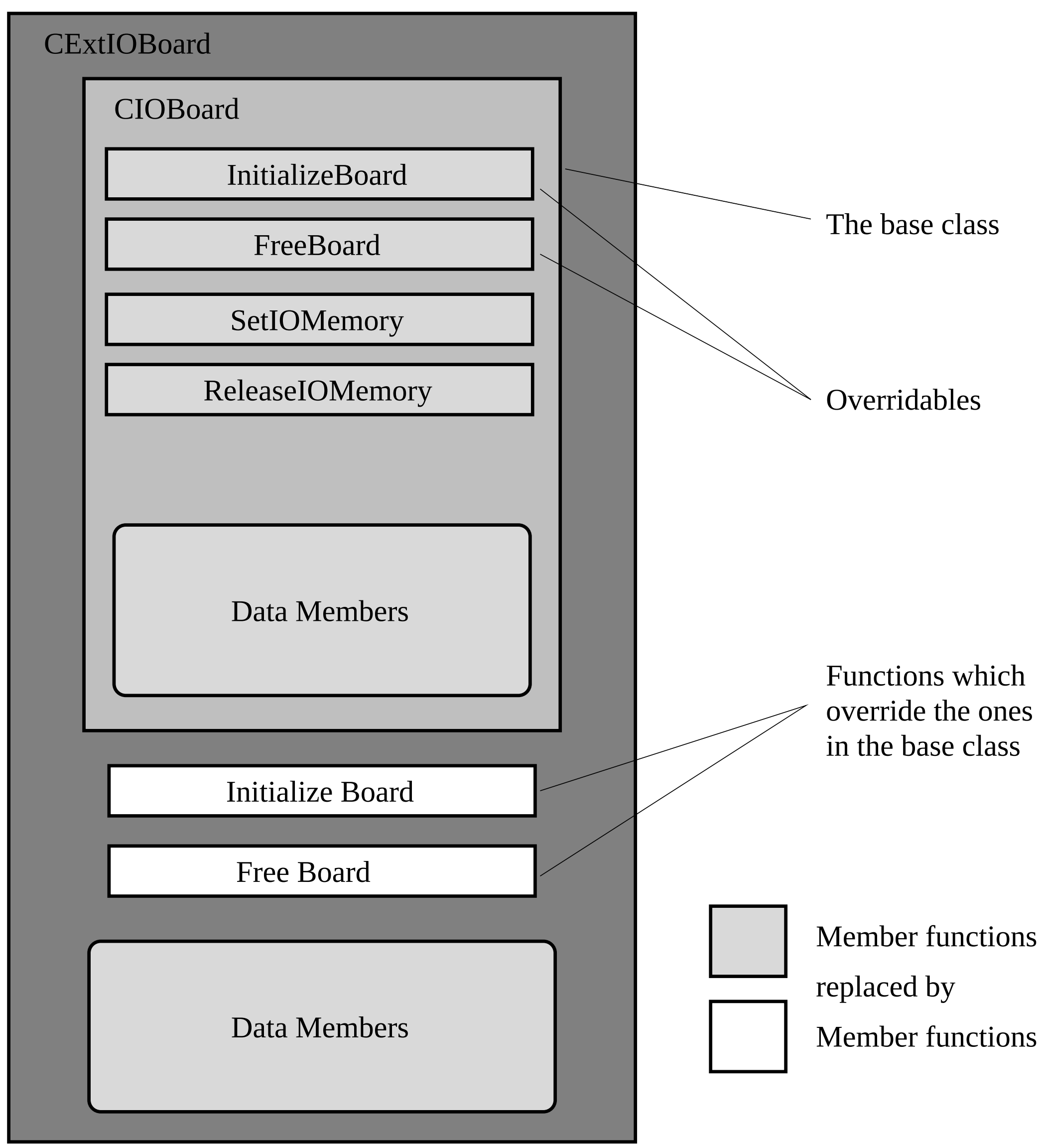


Figure 3.23: Simple overriding

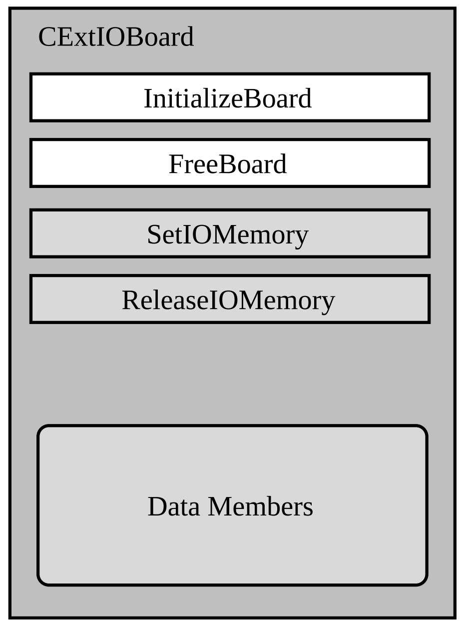


Figure 3.24: A “terminal”-look at the entity with overridden functionality

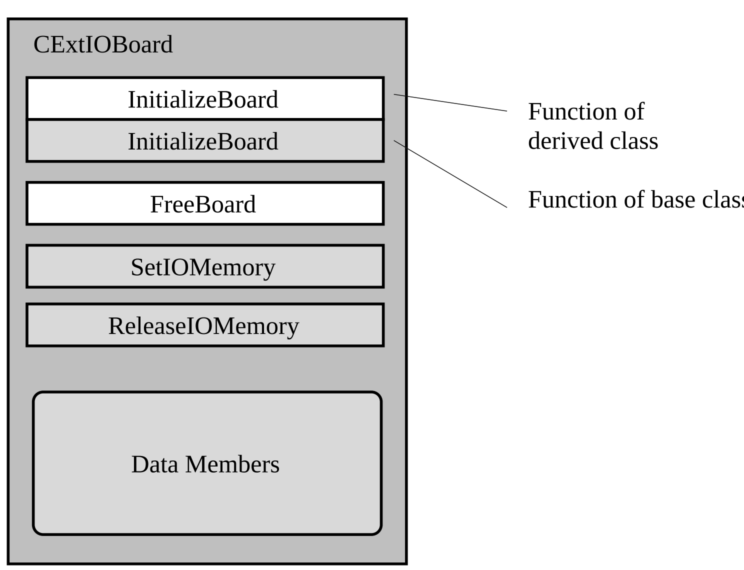


Figure 3.25: Overriding with explicit call to the base class

3.3.7 Encapsulation

We have designed entities according to a certain principle: one entity handles one physical (real-world) or software object.
The term encapsulation describes how certain functionalities are embedded in the entity. It also has a second meaning - encapsulated data-members in entities are normally passed by a member-function to any requesting unit/entity.

Example: The entity of type CBank handles money - the accepting and handing out actions are performed by member functions.

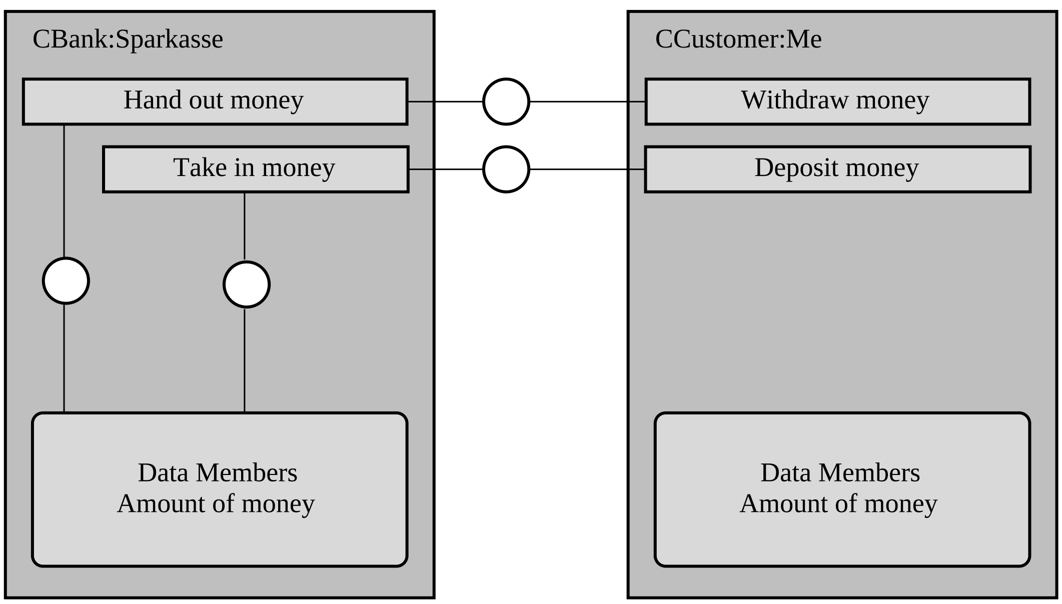


Figure 3.26: Encapsulation of money-handling in the Bank entity

Why is this necessary? One reason is of course that a complex entity has to take care of all its data members. Assume the case in the example; if a bank accepts money, the customer's accounts have to be updated. In addition to this the entire financial tracking system has to be well maintained.
Thus it is not enough if the customer were to keep his own account (provided the customer can be trusted). The same question can also occur in software systems. Although a strong encapsulation takes a lot of computational effort, it can be used at least in some specific cases.

To strike a compromise between speed and safety, we are setting up the following rules:

- Data-members may be read from everywhere
- Data-members are not directly written except for some special cases

- Each operation changing the contents of the entitiy's memory block has to be a member of the data-containing entity

This approach is not 100% compatible with the public, private and protected concept in C++.

3.3.8 Too much object-orientation

Object-orientation is as discussed a versatile tool for software development. Of course, it also has its drawbacks and disadvantages.
The grouping of data and code, the instantiation procedures etc. take time and make a program slower, if used frequently.
As in most technical problems, the developer has to make a trade-off between structured development and speed of execution. In modern systems, the need of computational power is in most cases satisfied and in a well structured application, we will not find much difference in execution time in object oriented and flow oriented software systems.
However we still have a question to answer "how much object-orientation is too much and how much is all right?" A descriptive example will help us understand the problem better.

Example: A dynamic matrix class is needed for an in image processing application. We will try to develop the code according to our object-oriented principles.

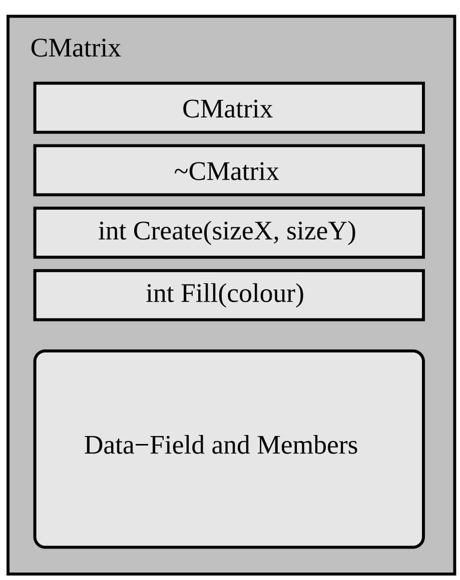


Figure 3.27: Object oriented approach to matrix storage class

The Create member function allocates a field of memory which we allow to be accessed by other entities.

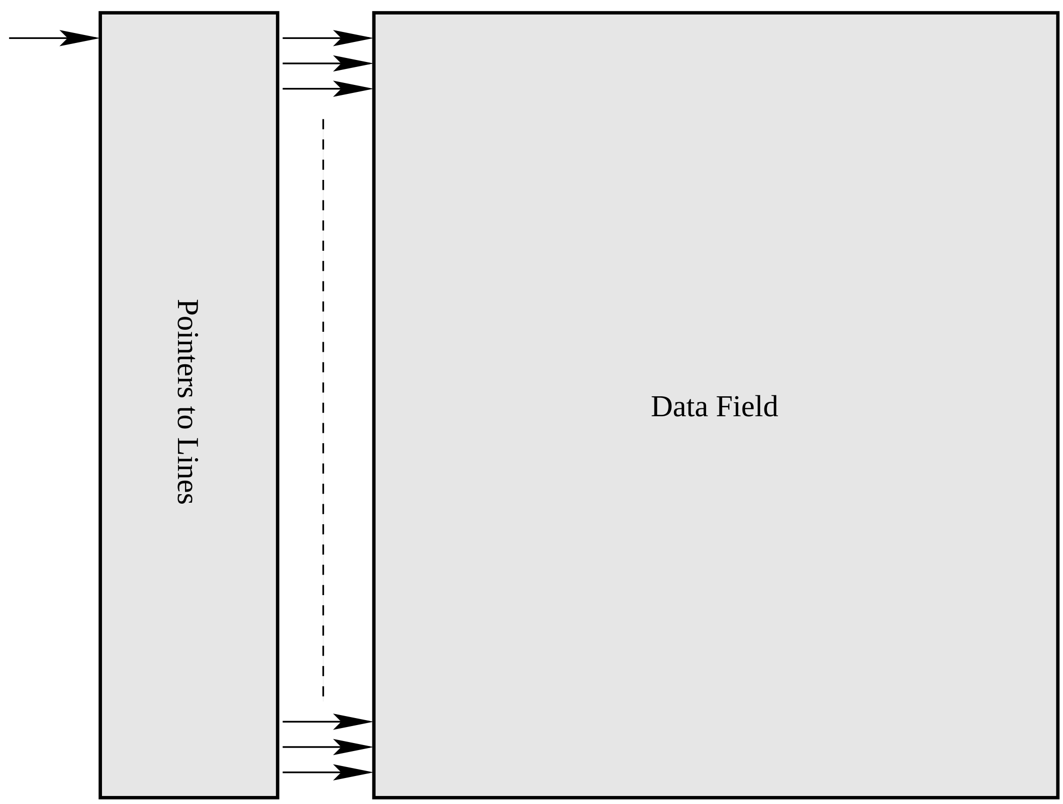


Figure 3.28: The matrix's data-field

Hereby we guarantee a well structured entity for matrix operations which is easier to handle compared to flow-oriented code. So far this approach helps us. The overhead for class-initialization and cleanup consumes very little time in comparison to all other operations and will thus neither be detected nor be remarked even using benchmarking tools.

The extreme case - the purists' case - is still correct according to our design strategy: we could set up an entity for every pixel (as a software object) and encapsulate the floating point value in the pixel. To initialize one floating point pixel of approximately 8 byte, we would have to initialize a whole entity (which takes maybe 1 kByte of memory). Even worse than the memory consumption is the overhead (if only calling the encapsulating data-handling functions and constructors/destructors). By the way, this is one reason why modern text processors create huge files which take lots of megabytes even for very small text of some pages: every letter in a line of text is set up as an entity containing all specifications such as style, colours and other properties.

3.3.9 Rapid Prototyping and Object oriented replacement for Interpreters

Rapid Prototyping is in our days a frequently used term - rapid prototyping allows engineers to have their solutions ready fast because development and programming-effort is minimal and a development-environment with an overall-design is available.

To develop the structures for a rapid prototyping system, we have to find out how interpreters can be replaced by object oriented structures.

The role of an interpreter is - roughly speaking - to read command-lines and translate them into function calls, passing the recognized (resp. interpreted) parameters to this function.
One problem related to this task is that a parser program has to be set up which reads the commands from either the keyboard or any other input source, such as file-input.
This is one reason which makes an interpreter slow in comparison with compiled code.

For many projects which are related to rapid prototyping, a somehow interpretative structure is absolutely necessary if one wants to avoid building compilers for special applications.

The classical interpreter reads one command, such as

```
PRINT a
```

and decomposes it into argument and command part. The command is then simply translated into a function call with the given parameter. Assume the interpreter is built in C, then it would make a call to

```
void print(char* a)
{
 //Printing code
 ...
 //Finished!
}
```

In addition to that, the interpreter checks whether the given parameters are ok or if e.g. the number of parameters is exceeded, etc. We can avoid the interpretation mechanism and obtain nearly the speed of a compiled program if we introduce an object oriented structure.
In order to obtain a suitable structure, we first think of what to include in an operator:

- the executable command (a standard function)
- the parameters
- a possibility to parametrize the given function

If we take a closer look, this structure resembles closely to a standard entity which we have already examined.:

- member functions which also include the executable command
- the necessary parameters (member-variables)
- a possibility to parametrize the operator: either through serialization or through a graphical user input

Example: a classical interpreted program for reading user-input and printing the same input

```
INPUT A
PRINT A
INPUT B
PRINT B
```

This is according to our first investigation translated into:

Interpreter Action	Program action
read command + params decompose command found INPUT command	input(a);
read command + params decompose command found PRINT command	print(a);
read command + params decompose command found INPUT command	input(b);
read command + params decompose command found PRINT command	print(b);

Concerning the program actions, we consider again functions written in C with the following prototypes:

```
void input(char* text);
void print(char* text);
```

The same code will now be built up according to the object-oriented approach and to the principles introduced above. For every command, we set up an entity

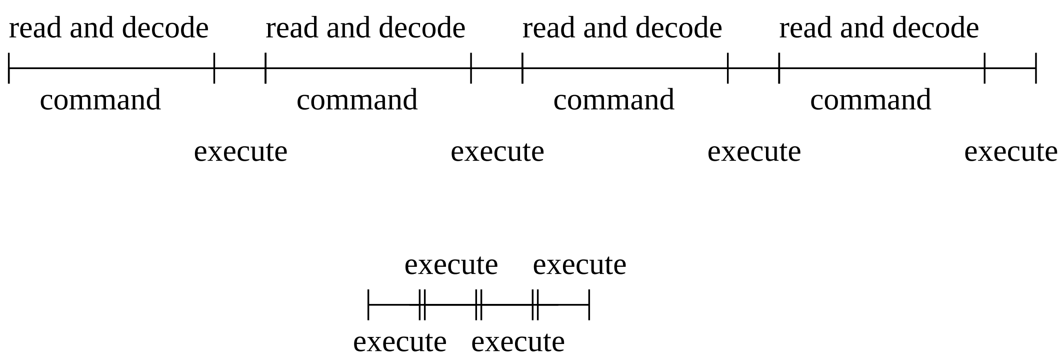


Figure 3.29: Top: timing in the interpreted case, bottom: timing using an object oriented replacement, the small gaps in between the execute-parts depict the time consumption by the overhead processing (pointer- and list-operations)

providing the required facilities.

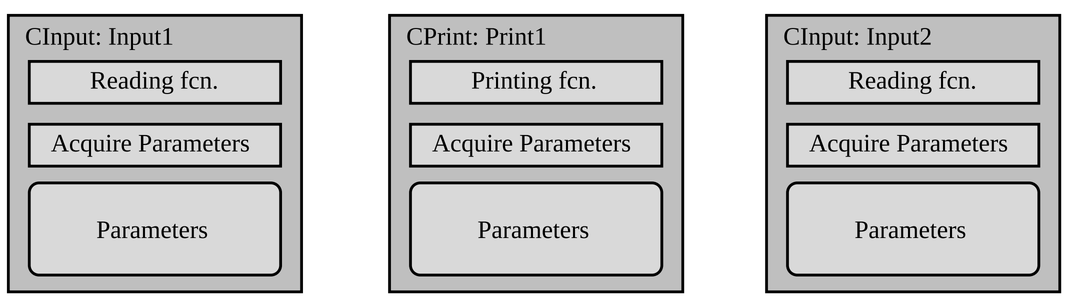


Figure 3.30: Object oriented representation of the command structure

These blocks must now be connected in an appropriate manner. As the blocks need data from other blocks, it is necessary to provide simple access. Which data is needed by one block is set up using the parametrizing function of the block. The parameter could simply be a pointer to the storage area of any other block which contains e.g. the user's input.

The appropriate meta-system for these operator-blocks is the object oriented list. Every node of the list contains one of the operators and besides the list functionality a possibility to execute the member functions. of the blocks in the list. The blocks themselves have to “know” about the list to obtain their required data from another block in the list. And the blocks themselves contain lists for data-output.

In this chapter, we have developed the necessary tools to decompose or compose a system the object oriented way. Now we have to think about another important aspect of object-orientation: taking advantage from inheritance and building up abstraction levels - so called class hierarchies.

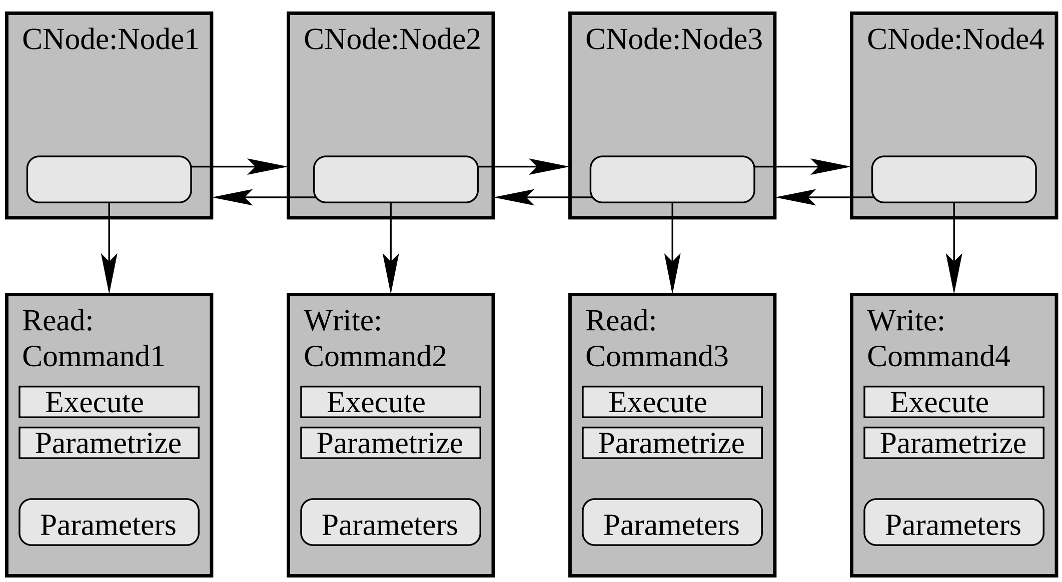


Figure 3.31: An object oriented list which interprets commands

Chapter 4

Hierarchical Class Design

Hierarchical class design is an important topic in object oriented modeling. By using a good design concept, the building process of a complex application can be kept transparent.
A hierarchical class design requires thinking in terms of abstraction levels: the basic operators must form a most unspecified functionality but a well defined structure. It has to form a generic base class.

Example: Given is a block diagram. All the different types of blocks must be specified originating from one base class.

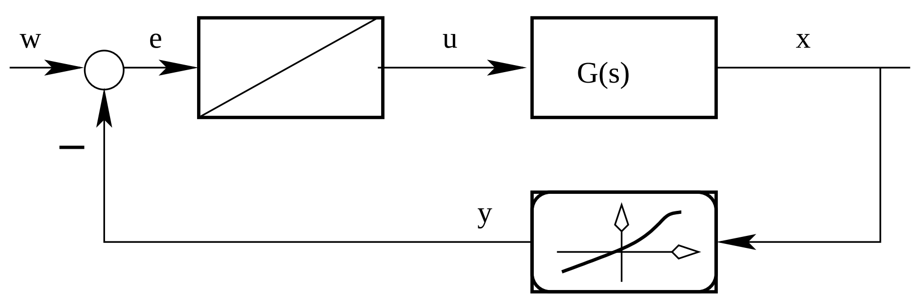


Figure 4.1: A block diagram for a control systems setup

4.1 Building a generic basis class

To build up a hierarchical class design for the problem given above, we have to first find the most general design for a block. A block has inputs, outputs, a certain functionality and as it is realized on a computer system, some initialization and deinitialization operations (such as allocating or freeing memory etc.) have to be performed.

Example: From the given Block diagram, one has to build the generic basis class

Figure 4.2: One Block as a grey box for all blocks

The unspecified block has to contain:

- p Input connections
- q Output connections
- An empty functionality
- Cleanup and initialization code

This functionality has to be inherited by every block. If we think in block-structures, the software equivalent to the generic block of the block diagram can be given as shown in Fig. 4.3.

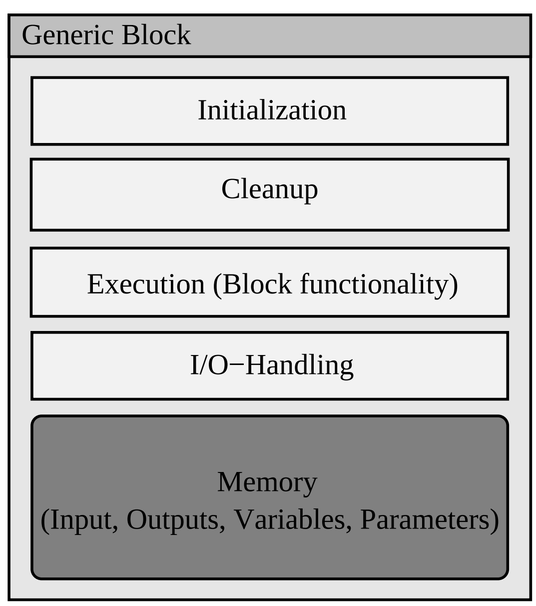


Figure 4.3: The generic block in software

4.2 Building more specified classes

The next step is to extend the generic block's capabilities. For this purpose, we first focus on an integrator block. The integrator block (in case a single

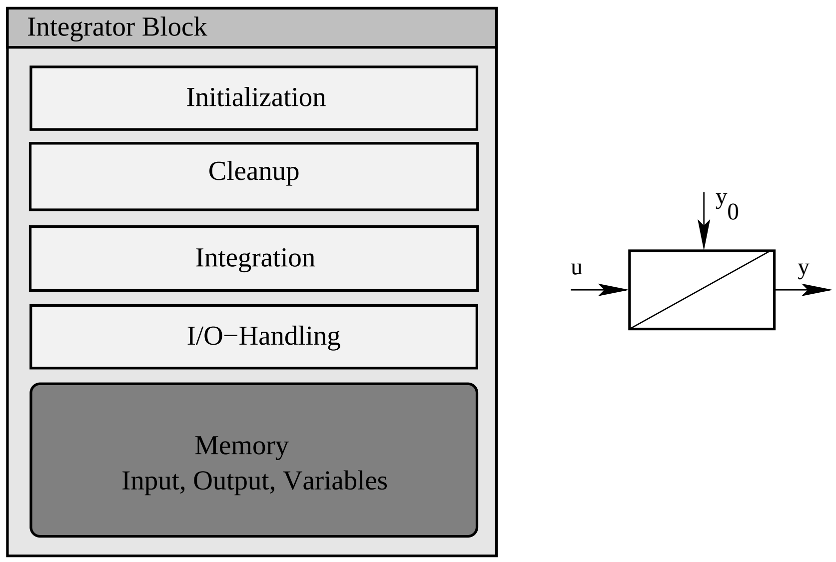


Figure 4.4: The integrator block in software

input/single output system is considered) would be structured as given in Fig. 4.4
the Input/Output restriction to a SISO-system is not satisfactory; for generality reasons, one has to keep the input output structures as variable and transparent as possible. One important fact is, however, that the number of inputs and outputs should be freely configurable.

Before we go into detail concerning the different structures of the operator, we first analyze the two other blocks. For the nonlinearity, one obtains the structure given in Fig. 4.5
Finally, we obtain an informational block structure for the transfer function $G(s)$. This block is given in Fig. 4.6
Now, the question is, if there should be introduced another abstraction level, another layer of less or more specified block types. This reflection leads to the following result:

- All given Blocks differ concerning their needed data and their *basic* functionality

- It can be foreseen, that for different block-types, e.g. nonlinearities, another type of functionality is needed than for e.g. an integrator

If we want to introduce more generality, another Block layer can be included. This new block-layer is a more specified version of the generic block with special functionality for the actual block's duty. Assume, for our problem, we specify the block types *Transfer Functions*, *Integrators*, *Nonlinearities*, *Multipurpose blocks*. From these intermediate and more specific definitions, one can derive the final

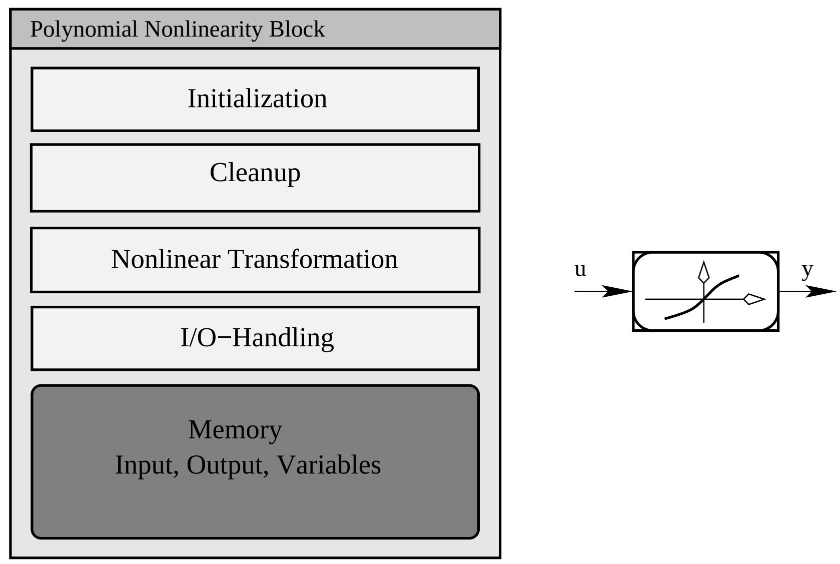


Figure 4.5: The polynomial nonlinearity block in software

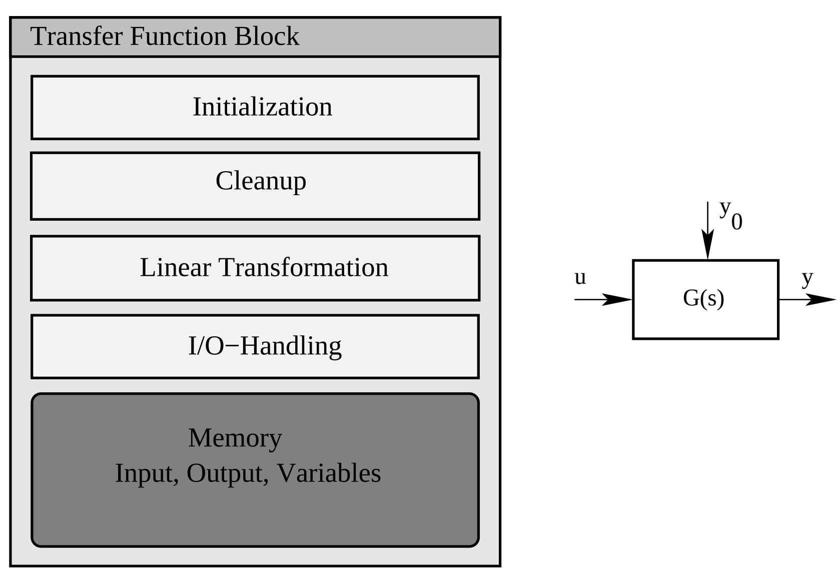


Figure 4.6: The transfer function block in software

block, using the functionality already given in the intermediate layer.

With these layers of abstraction, a real hierarchy can be built up; an example hierarchy is given in Fig. 4.7

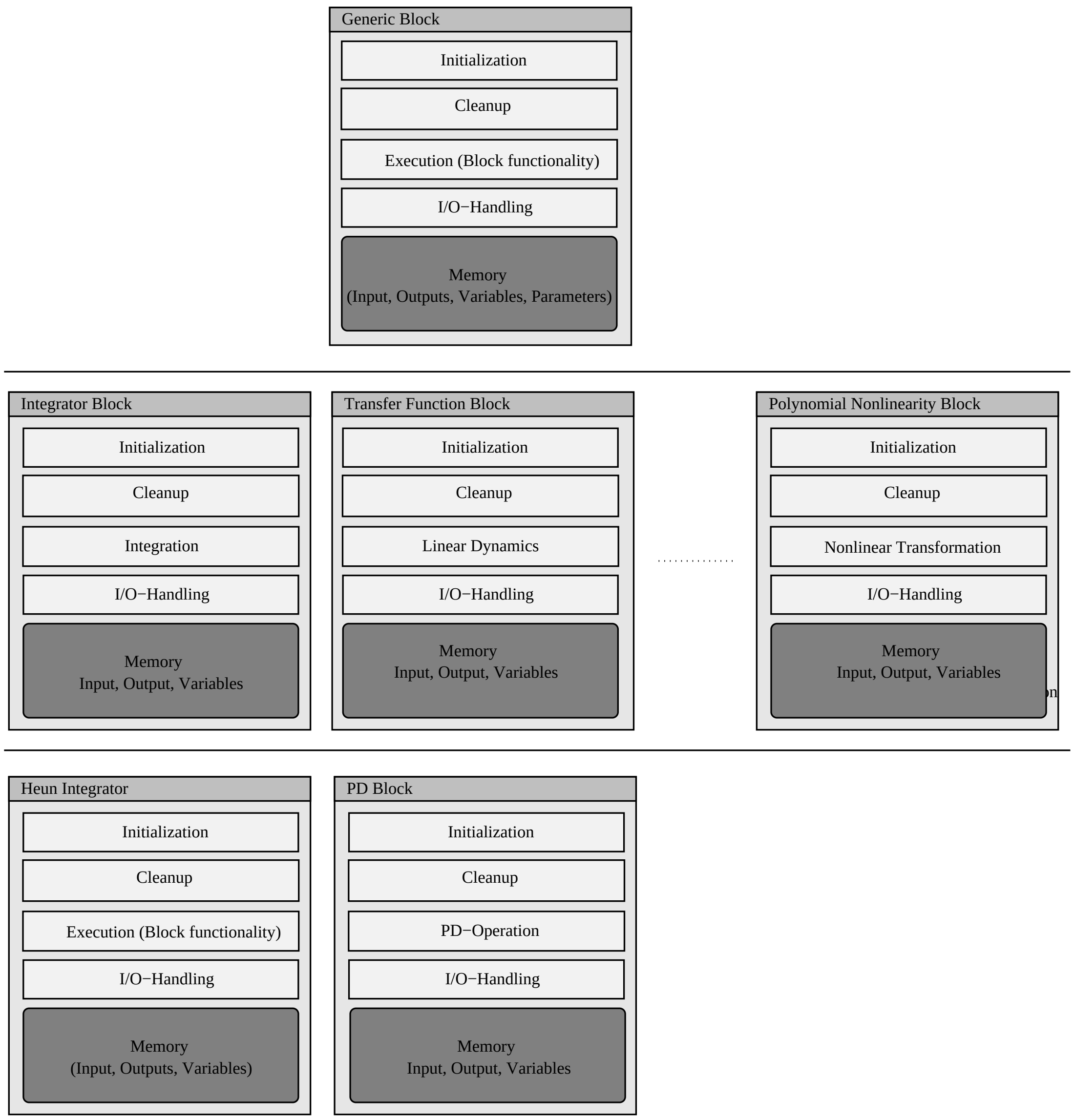


Figure 4.7: The hierarchy of transfer function blocks

4.3 Structure of an operational block with respect to execution time

To evaluate the structure of an operational block, it is necessary to make a model of the execution process. As we saw in the beginning, Petri nets with timed switching added can be used for that purpose.
The flow diagram, given in Fig. 4.8 shows the execution-oriented view of the block which takes us back to the timing question. A very effective timing "visualization" can be performed again using Petri nets with duration extension.

The object oriented view on a system structure does not give any *timing-related*

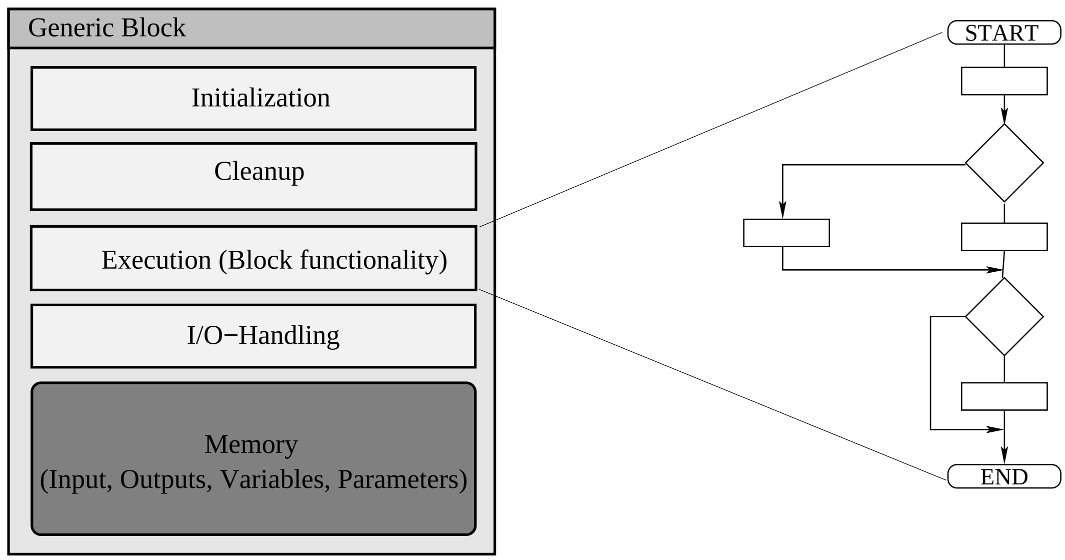


Figure 4.8: The flow diagram of the functional implementation of the operator

information. To show that, we go back to the block diagram example for which we derived the classes for the different blocks. Now the task is different; from the given blocks, which are now present in software, we have to construct a control loop and run the loop.

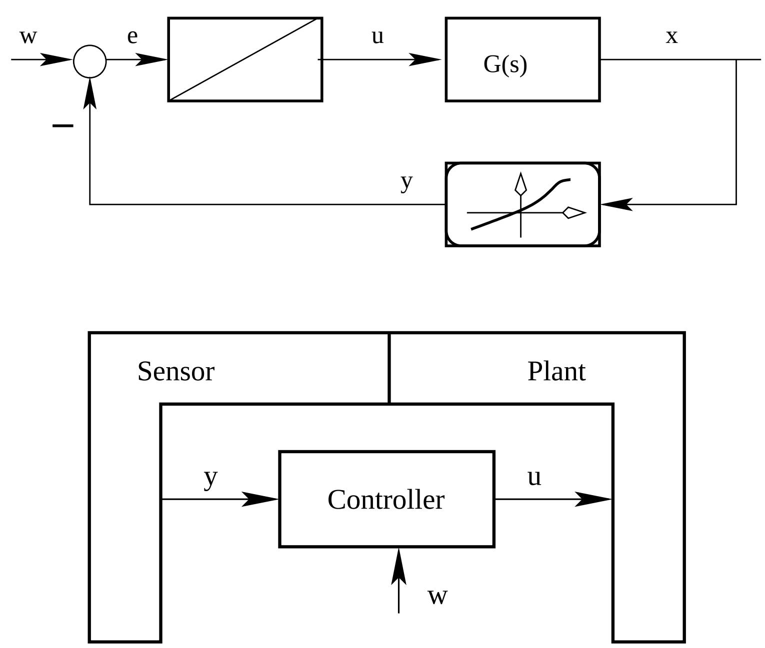


Figure 4.9: A block diagram and the informational setup

If Fig. 4.9, we find the control loop from the above example. We now assume, that we do not have a real hardware, but that the whole loop is constructed using software blocks. For that reason, we are free concerning the execution time of the blocks and can focus on the flow structure which we build up using the given blocks as given in Fig. 4.10

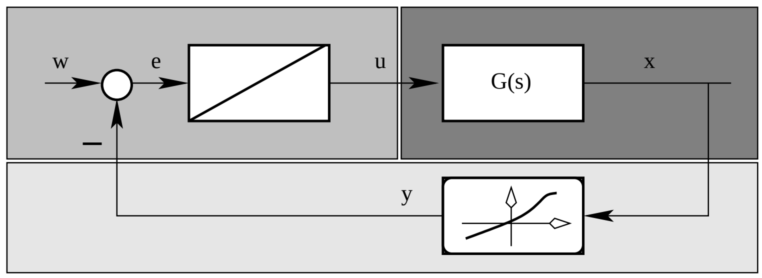


Figure 4.10: The informational view on the block-diagram

For building up the given control loop in software, we need:

- *Controller*

  → A summation block

  → An integrator block

- *Plant*

  → A transfer function block

- *Sensor*

  → A nonlinear function block

The informational setup leads to the software-block-structure given in Fig. 4.11 Now, the setup of the execution has to be made up. As you can see from Fig. 4.11, the structure is modeled but another system is needed which triggers the execution of the execute-functions in the blocks and before that forces initialization or after the execution ends cleanup of the operator blocks resp. their memory. A system which controls a system with structural variance is called a *Metasystem* in the following. The Metasystem has in our case the following tasks:

1. Seting up the given block structure
2. Executing the initialization routines of the blocks
3. Executing the execute-routine of the blocks
4. Cleaning up, using the cleanup routines provided by the blocks

The blocks must, to grant a proper system-theoretic function be executed in such a way, that no algebraic loops occur. In more complex applications, one has to set up an execution scheduler that decides, which block has to be executed. In the present case, execution is easier: following the loop from the left of Fig. 4.11,

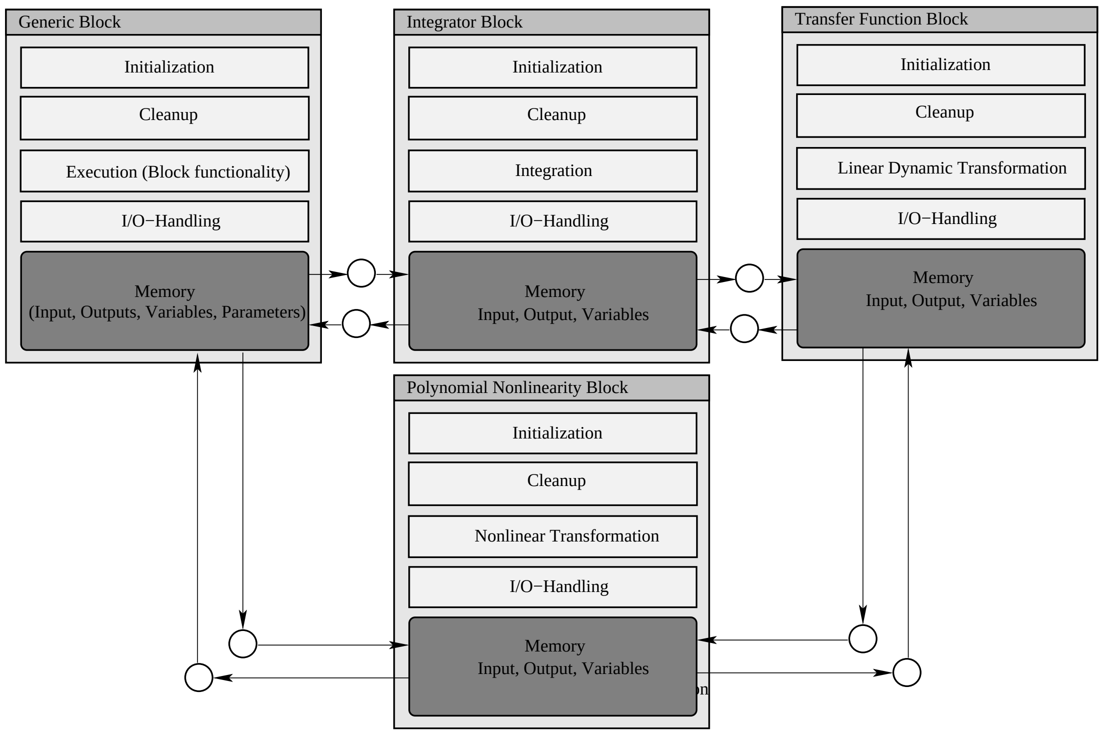


Figure 4.11: The informational view on the block-diagram

one execution function after the other can be executed, rebeginning at the top left of Fig. 4.11. This problem becomes more complicated if real-time systems with connections to hardware are focused.

The example makes clear, that time-oriented modeling and structural setup of an object oriented program are practically independent and in addition to that have to be thought over explicitly.

Now, there are two questions remaining: how is data transport organized between two blocks - or is it necessary to transport information? If we extend our view and take the Metasystem into account, then we obtain a better idea of how execution of the operators and data-handling can be performed.

4.4 Hardware in the loop simulation

Hardware in the loop simulation is a more and more frequently used term in our days. The difference between hardware in the loop simulation and pure simulation is that certain actors or sensors are used as in an industrial setup also in the laboratory experiment.
Hardware in the loop simulation makes the modeling error smaller as only parts of

the whole system have to be approximated using mathematical models. Another advantage is, that parts for which it is difficult to find an appropriate model can be used in their original form and the effort of the developer can be directed towards e.g. the control system design.
In our general software structure, we can easily build up a hardware in the loop system.

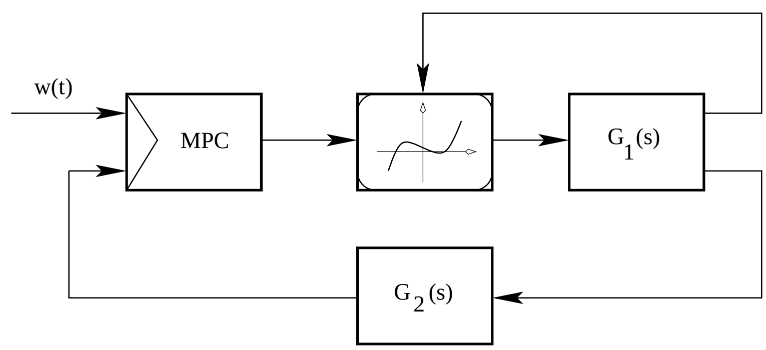


Figure 4.12: A complex nonlinear control systems setup

Figure 4.12 shows a control systems setup which is difficult to handle and furthermore contains an MPC (model predictive controller). Modeling of such a system would require a very high effort. A model which only creates nominal system outputs is not suited to controller design. This problem can be avoided using hardware components, in our case for the plant and measuring system.

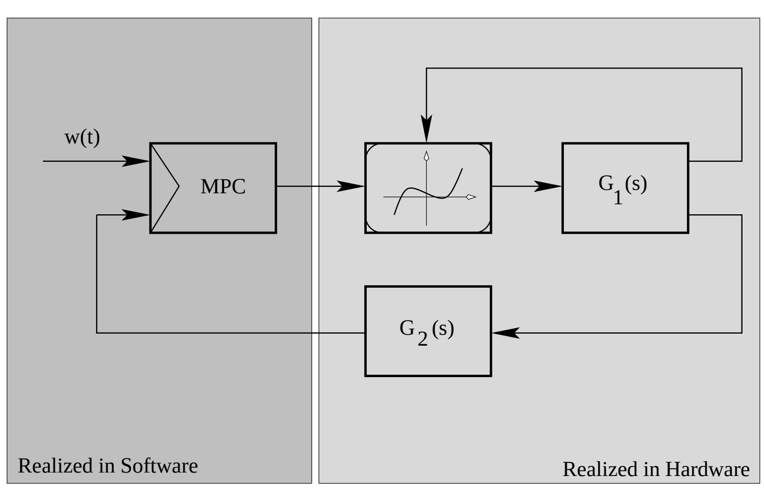


Figure 4.13: Subdivision into simulated parts (software) and hardware parts (hardware)

In Fig. 4.13, an exemplary subdivision into two parts, hardware and software part are shown. Of course, the software has to be run on e.g. an industrial PC and this one has to be coupled with the hardware parts using a suitable interface. This is in general an A/D-D/A converter board (analog to discrete time and vice versa). To handle these parts of the system, we have to introduce special entitys in our informational setup. The necessary extensions are shown in Fig. 4.14.

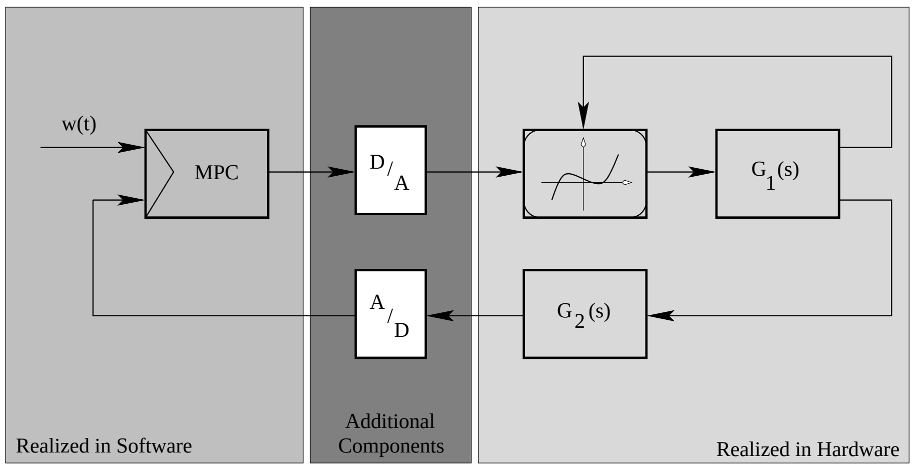


Figure 4.14: Additional components for a hardware in the loop system

From the block diagram given in Fig. 4.14 we have to derive a new informational structure to deal with the connections between hard- and software (see Fig. 4.15).

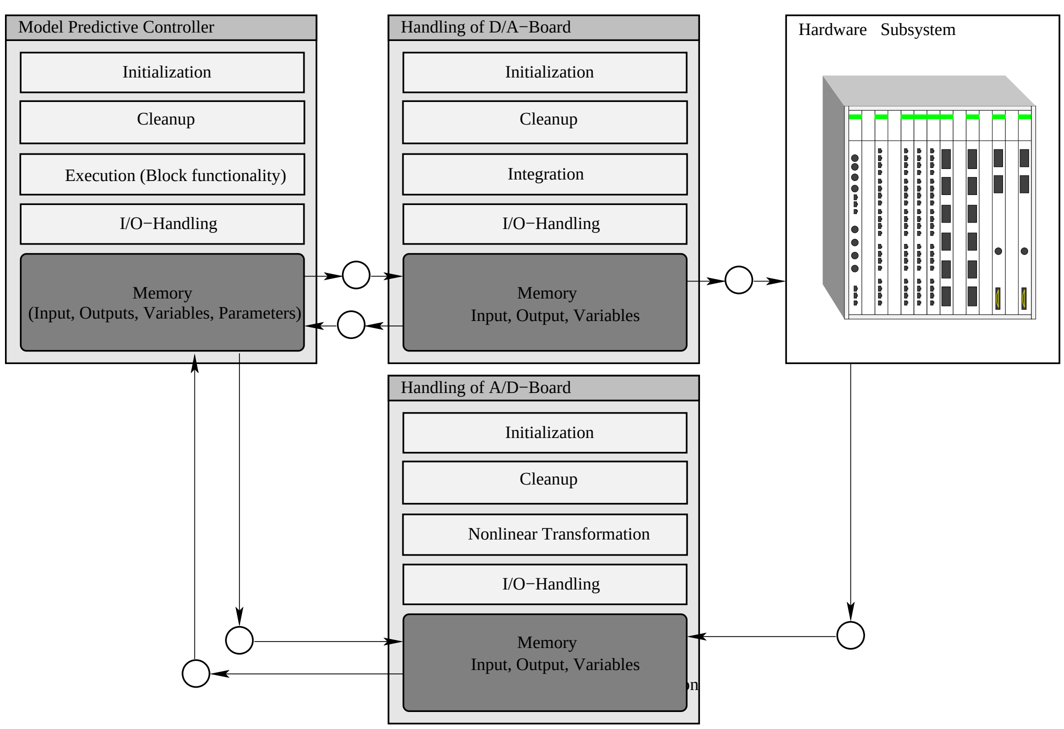


Figure 4.15: The extended informational structure of a hardware in the loop system

4.5 Transfer of structures for different platforms

Informational setups as given in Fig. 4.16 can be transformed for different platforms. In general, one has to set up e.g. a controller's or sensor's microcontroller in order to finish the development and to set up the end product for industrial use. A fully parametrized structure as it is given in 4.16 can be easily converted into a realization another platform by providing the operator functionality and the necessary data fields with the configurations that were set up using the original structure.

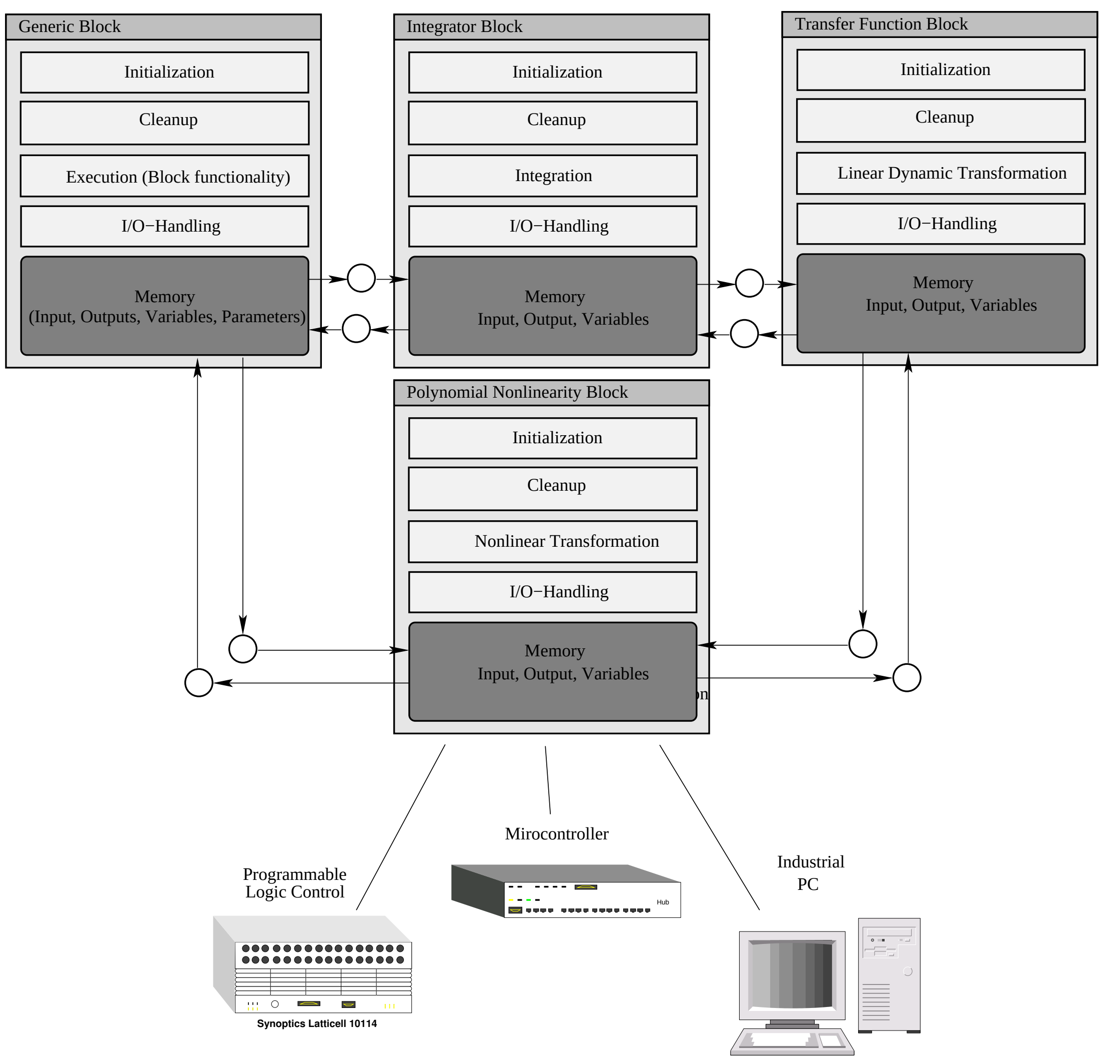


Figure 4.16: Informational setups for different platforms

After parametrization has been performed, the block structure is built up as specified by the developer on the simulation system but using the code for the operations for the target system and then downloaded on the target. This process can be more or less automated.

Chapter 5

Elements of complex systems

Systems which are easy to build up are most often systems with a fixed structure. Fixed structure systems are frequently used for simple tasks - e.g. a digital PID controller need not be built up dynamically. Many controllers however, especially in learning or predictive control systems require an overall dynamic structure.

5.1 Dynamic lists

If we consider dynamic system structures, the terms dynamic memory allocation and dynamic setup are essential. Dynamic data handling can be achieved using dynamic lists. The implementation of such a list will be handled in our block-oriented way of looking at this class of problems.

5.1.1 Construction of a dynamic list

A dynamic list - if it is built in the classical non-object oriented way - is difficult to handle; this is one reason why most programs developed during the 80's were built up in a statical manner. Dynamic data handling can be held very easy if object oriented concepts are used.

After the entity which is in charge of the organization and handling of the dynamic list is once constructed, the developer should not come into contact with anything else than a comfortable API which allows manipulation of the list's entries (such as insertion and deletion).

Then we have the list's nodes The list's nodes encapsulate pointers to the data-objects handled and pointers to the following or preceeding element in the list. this concept allows to store even elements of a different type in such a list (with additional overhead, which is automatically provided by C++). The list can be visualized as given in Fig. 5.1.

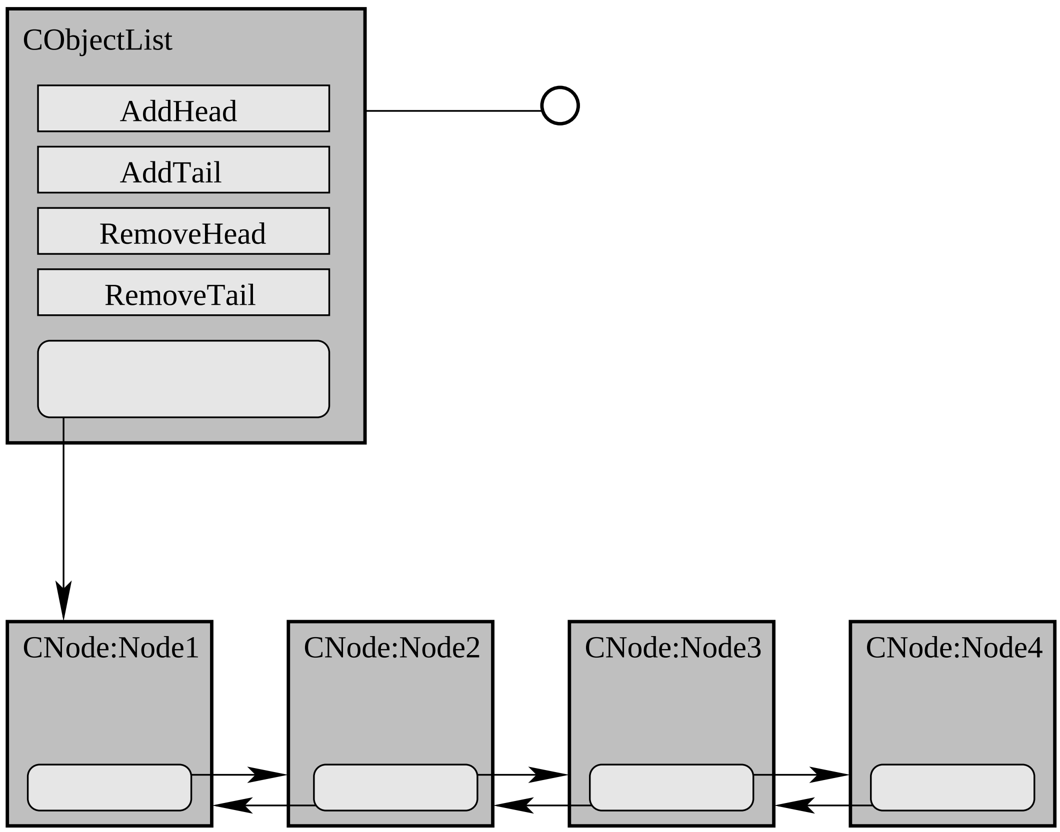


Figure 5.1: The object oriented entity diagram of a linked list with presently 4 nodes

The meta-system for the list's nodes is in this case the object oriented list entity itself which provides the list's API as member functions; these member functions perform all necessary tasks concerning the handling of the list.

Example: A sample class definition for a dynamically linked list and its nodes.

1. The lists node with constructor and destructor and pointers to the preceding object as well as to the following object

```
#ifndef ONEDIMNODE
#define ONEDIMNODE

#include <qobject.h>

class OneDimNode : public QObject
{
public:
        OneDimNode();
        OneDimNode(QObject* Previous,QObject* Next,
                   QObject* NewOne);
```

```
        //This member-function initializes the fields
        //of the OneDimNode-Object
        //with the given pointers

        virtual ~OneDimNode();
        QObject *m_Previous;   //Pointer to preceding node
        QObject *m_Next;       //Pointer to following node
        QObject *m_Actual;     //Pointer to contents of this node
};
#endif
```

2. The class definition for the ObjectList-Entity

```
#ifndef OBJECTLIST
#define OBJECTLIST

#include <qobject.h>

class ObjectList : public QObject
{
public:
        ObjectList();
        virtual ~ObjectList();

public:
        long       m_ListPos;      //Listenposition
        long       m_Count;        //Nummer der Elemente in der Liste
        long       m_old_i;        //Old wanted Position in List
        OneDimNode* m_FirstNode;   //First node of the list
        OneDimNode* m_OldNode;     //Last wanted node
        OneDimNode* m_TailNode;    //Last node of the list

public:
        long GetPosition(QObject* ThatObject);
        //Get Listpos from Pointer
        //If pointer is not valid, that means Object wasn´t found,
        //-1 will be returned
        //the long variable for the first object in the list ist 0

        void InsertBeforePtr(QObject* ThatObject,QObject* Follower);
        //Insert Object before another Object
        //Insert a Pointer (to the new Obejct) before an Object
        //in the list
        //if the Ptr. to the follower is not valid,
        //Object is inserted at the Head of the List

        void InsertBeforePos(QObject* ThatObject,long Position);
```

```
//Insert the Object before the Object according to the
//Position variable
//If the position given is not valid,
//the Object will be inserted at the
//head of the list

void InsertAfterPtr(QObject* ThatObject,QObject* Previous);
//Insert Object after another Object
//Insert a Pointer (to the new Object) before an Object
//in the list
//if the Ptr. to the previous object is not valid,
//the Object will be inserted at the Head of the List

void InsertAfterPos(QObject* ThatObject,long Position);
//Insert the Object after the Object according to the
//Position variable
//If the position given is not valid, the Object will
//be inserted at the
//Tail of the list

void AddTail(QObject* ThatObject);
//Adds the Element given by the pointer to a QObject
//at the End of the given
//List

void AddHead(QObject* ThatObject);
//Adds the Element given by the pointer to a QObject
//at the beginning of the
//given List

void RemoveHead();
//Removes the Head of the list

void RemoveTail();
//Removes the tail of the list

void RemoveAtPtr(QObject* ThatObject);
//Removes an Element according to the given pointer

//Removes the Element at the specified position
void RemoveAtPos(long Position);

//Gets the Tail-Element´s position
long GetTailPos();

QObject* GetTailPtr();
//Retrieves the ptr to the !!user's!! object stored
```

```
        //in the Tail node in m_Actual

        long GetAtPtr(QObject* ThatObject);
        //Retrieves the ptr to the !!user's!! object stored

        QObject* GetHeadPtr();
        //Retrieves the ptr to the !!user's!! object stored
        //in the Head node in m_Actual

        long GetCount();
        //Returns the length of the list

        long GetHeadPos();
        //Returns the Position of the list´s head

        QObject* GetAtPos(long Position);
        //Returns the Pointer to the Object which is
        //determined by position

        OneDimNode* GetHeadNodePtr();
        //Retrieves a pointer to the Head-Node

        OneDimNode* GetTailNodePtr();
        //Retrieves a pointer to the Tail-Node

        OneDimNode* GetAtNodePtr(QObject* ThatObject);
        //Retrieves a pointer to the Node at a given position

};
#endif
```

5.2 Serialization and Reconstruction of blocks

The dynamic data-handling also requires a dynamic concept for data storage and restauration. One problem however with the dynamic structure is that the meta-system cannot perform these operations (called serialization in short) - if these operations had to be performed by the meta-system, it had to “know” about the overall data structures of all entities which have to be serialized. This is of course easy if everything is fixed. From the dynamic point of view, it is better to “delegate” the storing and reconstruction to each entity itself.

To avoid these difficulties, the object-oriented approach provides the necessary tools:
We add the serialization to the entity itself - the object oriented list for example “knows” its length as it operates on its contents and sets up or respectively de-

stroys nodes and has to remember its state.

Example: Take again the example of a dynamic list; now the list has to be saved and also reconstructed. To perform this task, the meta-system had to “know” the following details:

- Type and length of the node-items
- number of nodes

The same holds for reconstruction of the list.

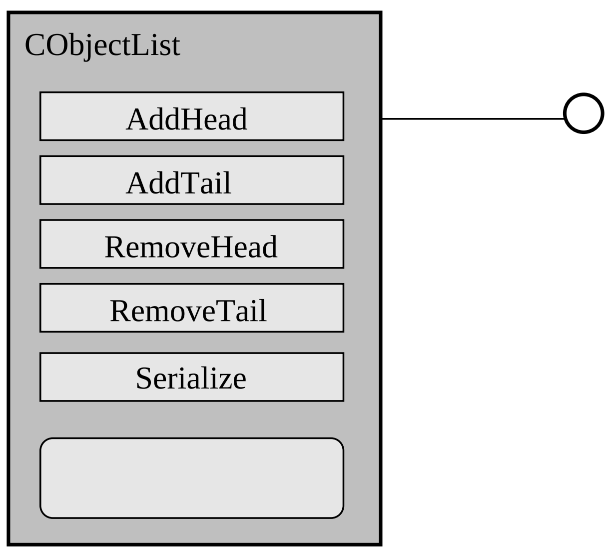


Figure 5.2: Extended list entity with serialization added

The serialization-functionality is taken over by a specific member-function as it is shown in Fig. 5.2.
The serialization-function is build up with storing and loading code. The higher level-entity calls the serializaition function of the lower level entities.

```
void ObjectList::Serialize(Archive *ar)
 CNode* m_Node;
 int i;

 if (ar->isLoading==TRUE)
 {
  //Read length of list
  ar>>m_count;

   for (i=0;i<m_count;i++)
   {
     //Add a node to the list
```

```
     m_Node=new CNode;
     //Call the node's serialize-method
     m_Node->Serialize(ar);
   }
 }
else
{
     m_count>>ar; //Save length of List

     for (i=0;i<m_count;i++)
     {
       m_Node->Serialize(ar);
     }
}
```

The code for serialization of the object stored in the node is then invoked and migth look as follows:

```
void Node::Serialize(Archive *ar)
{
  //Call the Serialize method of the node's contents
  m_Actual->Serialize(ar);
}
```

The node's contents could then have the following serialization method:

```
void Object::Serialize(Archive *ar)
{
  //Call the Serialize method of the node's contents
  if(ar.isLoading==TRUE)
  {
    ar>>m_myvariable;
    ar>>m_mytext;
  }
  else
  {
    ar<<m_myvariable;
    ar<<m_mytext;
  }
}
```

5.2.1 Dynamic Data Handling and Data Exchange

Dynamic data handling and data exchange can be set up using the dynamic list that was introduced in the beginning of this chapter.

Example: Assume a multiple input multiple output signal processing operator block which occurs frequently in control applications.

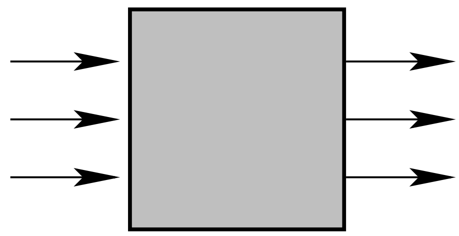


Figure 5.3: A multiple input multiple output (MIMO) signal processing operator

We have already developed software entities which are handling or representing real-world objects. As one of our major aims is the construction of a fully dynamic environment, we now focus on how data in- and outputs of such blocks can be connected dynamically.

First the outputs: if an operator provides multiple sets of information and different operators provide different sets of information, then it is difficult to find a dynamical solution for connection of in- and outputs if no whole underlying dynamical concept is used.

One possibility to handle data in a fully dynamical way is to put the entity's output-data in a dynamical list. The data-set can then be extended by a type-specifier which is checked by the following block for which this output has been selected as an input. Another reason for using a list-structure is that lists and their contents can easily be used to fill up user interface elements. Another advantage is that lists can be easily indiced and thus their entries can be selected by just giving the list position.

Our operator X itself contains a pointer on the list-entity. Thus it can access the list and all of its nodes - the other operator entities. If we now specify the index of another operator in the list, then we can obtain the pointer to the specific operator selected by specifying its index in the list. Further, we can select an input four our operator X just by giving the index of the list entry in the output values list of the operator Y. In that manner, we can connect all the operator blocks dynamically.

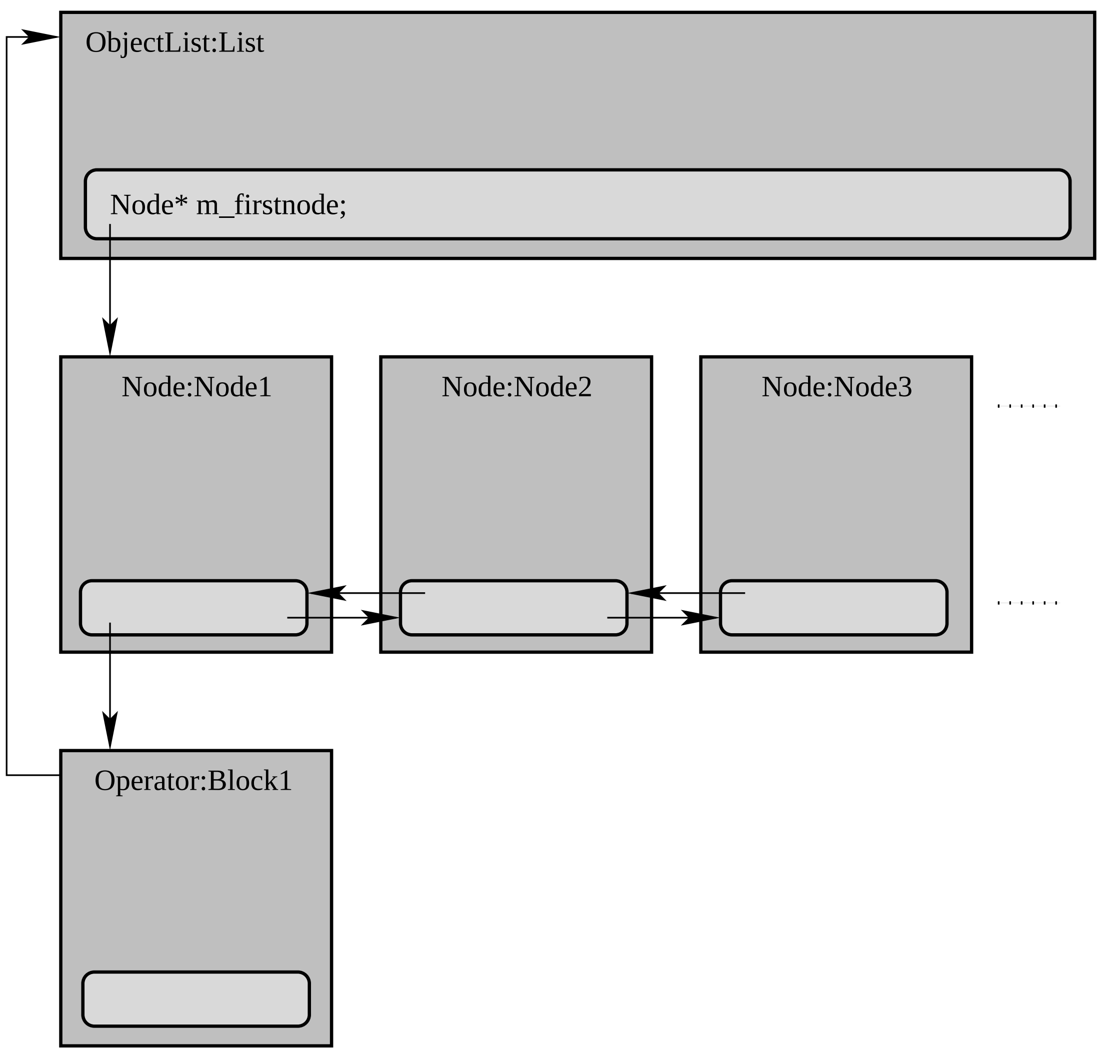


Figure 5.4: A dynamic data exchange concept for object oriented systems

Chapter 6

User Interfaces

The user interface is a very important part of the man machine interface. On one hand, the design of a user interface is a time consuming task, on the other hand, it is definitively a part of an engineering solution.

A user interface has to fulfill the following specifications:

- It has to be conformal to the norms given by the end user
- A good user interface is easy to use
- Process visualization is an important topic in building user interfaces
- User input has to be either forwarded to other entities of the system or to be preprocessed or displayed

Considering todays user-interfaces, the user-requirements raise with respect to time. That is one of the main reasons, why graphical user interfaces became popular. One problem which is encountered in the context of graphical user interfaces is that most operating systems that offer such an interface are not built for process control or signal processing issues and thus are not real time capable. Possibilities to overcome this lack will be dealt later in this book. However the need of a good visualization and graphical user-interaction is there. In the object oriented way of system design, one can easily decouple the user interface from other underlying system layers, also from realtime layers. That is the reason why user interfacing is dealt separately from other topics here.

6.1 The Basis of a user interface

If we are talking about User-Interfaces, we usually think of todays graphical operating systems and the software available. In fact, text oriented user interfaces vanish more and more. However, all the predescribed types of user interfaces have in common that they realize a man machine interface and thus have an important role in system design.

All user interfaces have in common, that they accept specific inputs from e.g. Keyboard, Mouse, etc. and evaluate those in order to yield the execution of a certain algorithm.

Talking about user-interfaces, the following important expressions occur:

- *Input devices* and their handling
- *Message queues* and *messages*
- Event or *message handling*

The input devices are the sources of user input for the user interface.

Where in simplistic systems, one can easily handle inputs and need not share any devices, in more sophisticated systems, the concept of messages comes into play. Messages are generated by the underlying layer of the operating system.

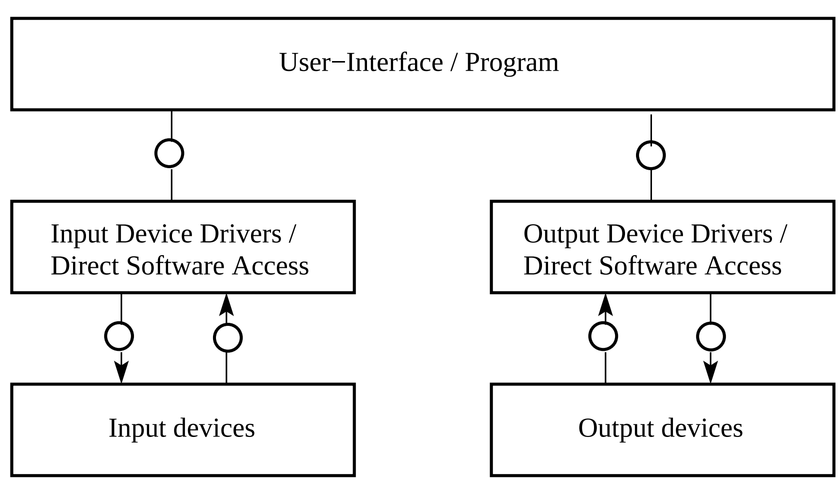


Figure 6.1: The simplistic and classical approach to a user interface

Thinking in terms of e.g. X11 (X-Windows on Linux/Unix-Machines) or Windows NT, we have to extend our view from the hardware-architecture to a multi-user multi tasking system. The resources (devices, i/o, file-systems) have to be shared amongst multiple processes (programs).
These systems usually provide drivers for the hardware and an API (applications programmers' interface) for a standardized i/o-interface.
This helps the programmer a lot, because he finds a hardware-independent interface and the program works on different machines not regarding the type of hardware used in the underlying architecture.

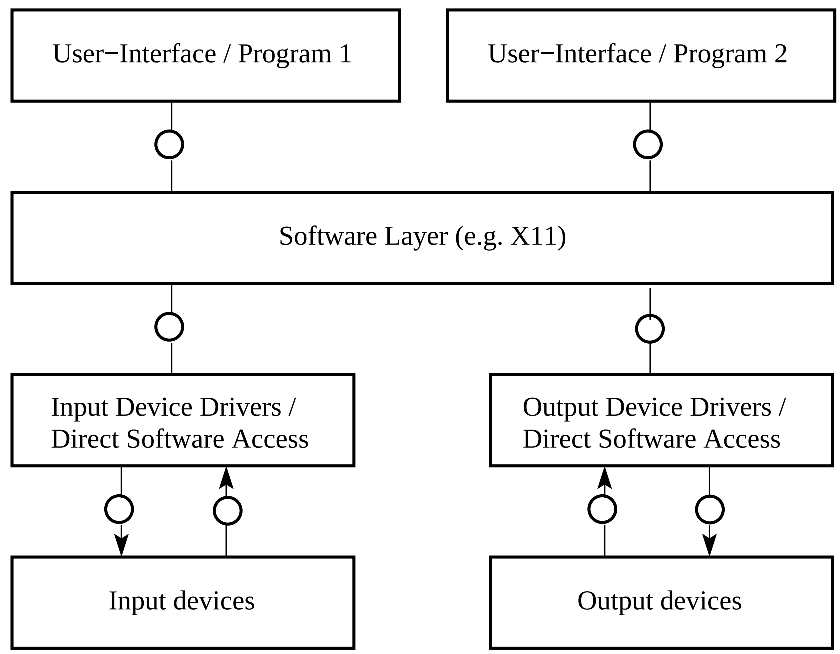


Figure 6.2: The simplistic and classical approach to a user interface

6.2 Classical programming

The classical way of programming a user interface is to introduce a function which asks for keyboard input, mouse input and other events.

6.2.1 A text-based interface

Text based interfaces are easy to implement. In general one obtains the structure given in Figure 6.3.
The simple concept given in Figure 6.3 works only if all actions are executed in a small time window. If more computational effort is expected, user input should be processed asynchronously.

Asynchronous checking can be performed if two tasks, one for calculus and one for user issues are introduced.
The structure given in 6.4 is the realization of the predescribed two task method; the communication can be realized using a fifo buffer, into which one task writes the actual requirement.

6.2.2 A graphical user interface

A standard graphical user interface can be built up in the same way as the predescribed text-oriented one. Concerning graphical output, this is not much more effort than the text-based version. One problem however is to implement

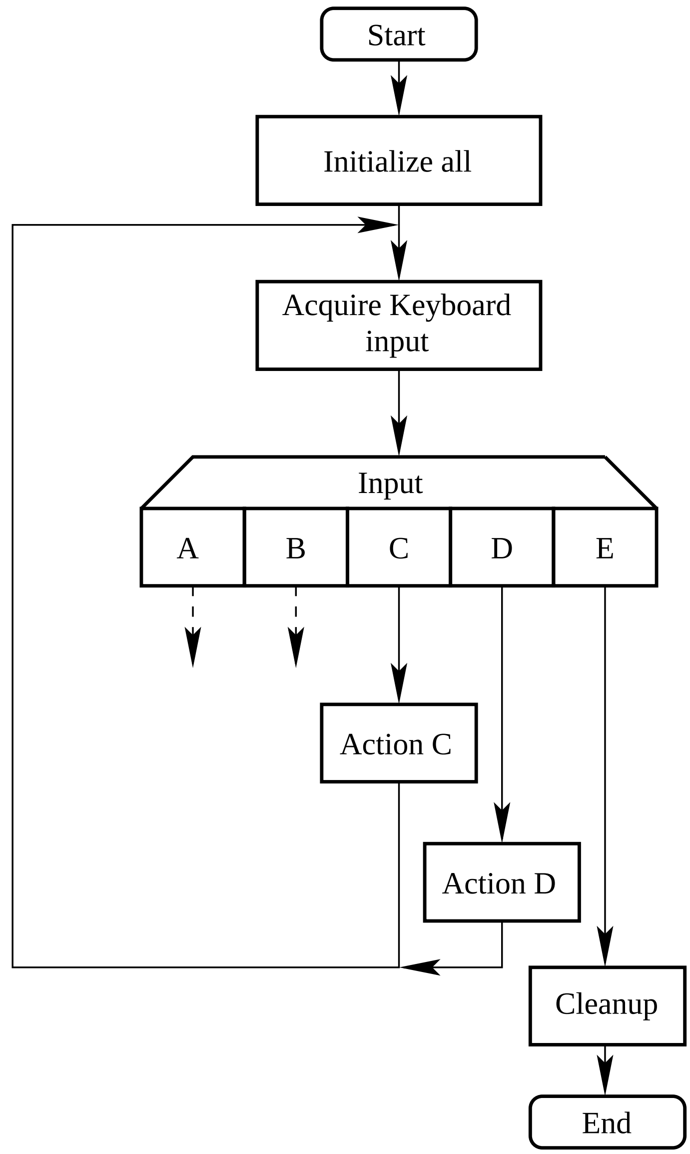


Figure 6.3: A classical way of user interface handling

graphical user input - in fact, it is a very time consuming task to build up user-interaction in a flow-oriented environment.

First, all Buttons and all user-changeable items are put up. The rectangular areas which have to be checked for e.g. mouse or touchscreen input have to be inserted in a list.

In the main loop of the software, mouse/touchscreen position has to be acquired and through the list, one has to find out, whether a button has been pressed or other fields have been altered.

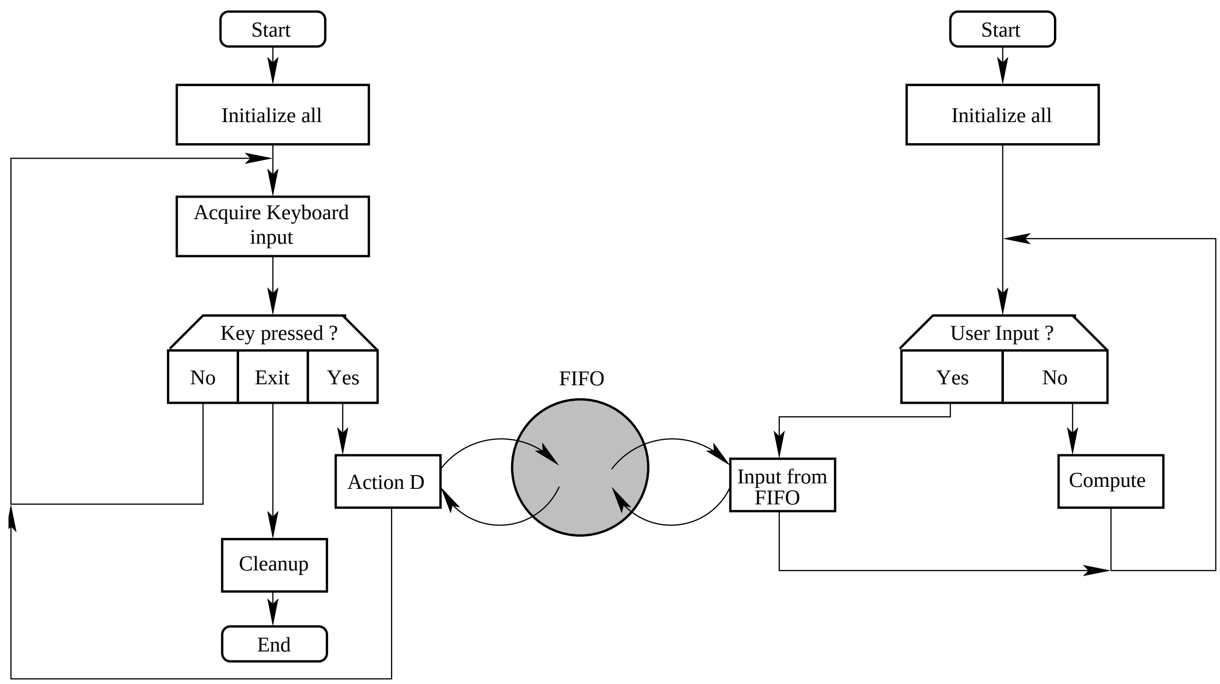


Figure 6.4: A user interface with two tasks and FIFO for message passing

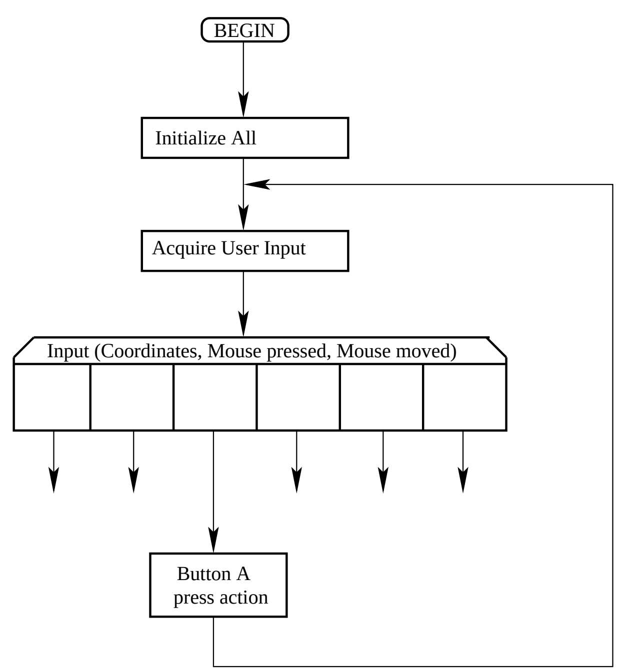


Figure 6.5: A simple graphical user interface (flow-oriented technique)

Then a huge case-command is executed in which we call the necessary functions to act accordingly to the user's input. (In former Windows for Workgroups programs, this was called the winmain-function)

One problem remains; that is the complicated structure of the user interface - it has to be tested, which element has the focus (is active) at a point of time and then the keyboard/mouse-actions have to be routed to that element.

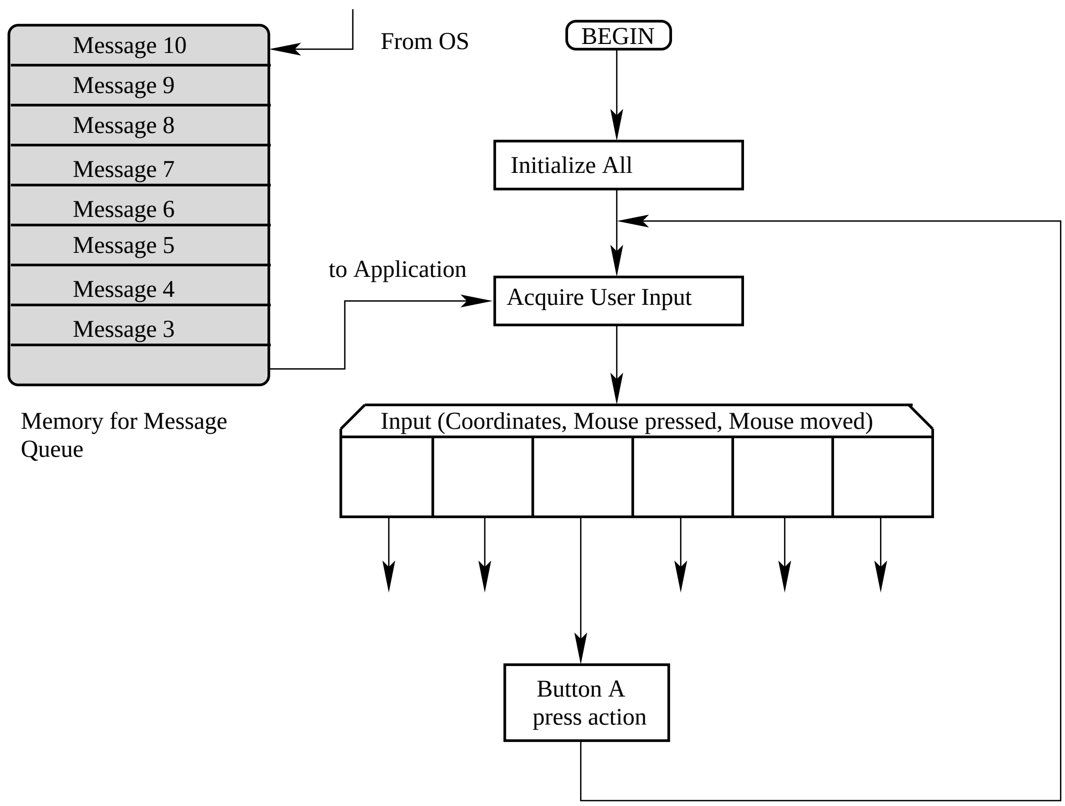


Figure 6.6: A simple graphical user interface with message queuing

That is the reason, why the concept of messages was introduced: all events which were generated by the user are first fed to a message queue and then read by the part of the program which is dealing with this particular element. In that way, even user interfaces with a high depth can be handled.

6.3 The object oriented user interface concept

For building up user interfaces, object oriented techniques have prooved that they are very efficient. The explanation for that fact is, that every user-interface element is an object which has to be handled. A button for example has certain facilities that are the same for all possible buttons.
Every button takes the same messages (e.g. mouse-inputs) as another button and thus has the same message handling routines. Again this is a case for our

object-oriented entity concept.

6.3.1 The main application entity - message handling

Talking about object oriented systems, we must not forget, that at least the "top-most" of the entities has to be flow-oriented.
This is compatible with what we said concerning setting up entities.
In general, the overall-meta system is the operating system. Dependent on the operating system, a certain startup-code is executed. This startup-code then installs the entities of which comprises the designed software system and in addition to that "connects" their information flow paths.

Example: Standard application paradigm

Figure 6.7: The standard message handling entity

The member function which handles the data streams from and to the operating system is the so called message loop "HandleMsg"-Function. This function polls the system for messages and passes them on to other entities of the system (see Chapter 6 for the standard-application and user-interface concepts).

An application-entity which handles the messages obtained from the operating system is contained in most of today's standard-application concept.

Example: A user-interface with 2 buttons

Generally speaking, there is no unique way of designing an interface (concerning its underlying structure). One approach has become very common although and is implemented as a standard-approach: the so called application-document-view paradigm.

The idea is again to separate different entities according to their "duty" in the whole construction; the paradigm that is described in the following has been first involved with a very specific aim: text-processing. Nevertheless the approach is useful as we will find out.

6.3.2 Setting up the entities

We need a concept for message queuing and initialization of the entities. This is the first step that leads us to a working application. The following program code is the main function of a small application using the qt-toolkit.

```
int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  a.setFont(QFont("helvetica", 12));

  Simpleapp *simpleapp=new Simpleapp();
  a.setMainWidget(simpleapp);

  simpleapp->setCaption("Document 1");
  simpleapp->show();

  return a.exec();
}
```

The application entity is created through instanciation of QApplication with Simpleapp as a main window. The Message-loop which is then handled by QApplication is started by calling a.exec(); The application's main window's constructor is called when Simpleapp is instanciated. It initializes the other entities: view and document.

```
Simpleapp::Simpleapp()
{
  setCaption("Simpleapp " VERSION);

  initDoc();
  initView();
}
```

where initDoc(); and initView(); are member functions of the Simpleapp entity as one can see from the following class definition:

```
class Simpleapp : public QMainWindow
{
  Q_OBJECT

  public:
    /** construtor */
    Simpleapp();
    /** destructor */
    ~Simpleapp();

    /** setup the document*/
    void initDoc();
    /** setup the mainview*/
    void initView();

  private:

    /** view is the main widget which represents
     *  your working area. The View
     *  class should handle all events of
     *  the view widget.  It is kept empty so
     *  you can create your view according to your
     *  application's needs by
     *  changing the view class.
     */
    SimpleappView *view;
    /** doc represents your actual document and
     *  is created only once. It keeps
     *  information such as filename and does
     *  the serialization of your files.
     */
    SimpleappDoc *doc;
};
```

The application-entity is then responsible for the further interaction with the operating system and the user. The application entity passes messages to other entities such as graphical user interfaace elements or vice versa to the operating system. The application entity itself realizes the meta-system for the rest of the entities which then handle the graphical user interface. The interface-functionality is then provided by the view-entity and the document-entity.

The view-entity realizes the viewer to a certain dataset which is contained in the document entity. At first sight, this approach seems to be useless for our type of applications - namely control and signal processing - because where do we handle documents?

Of course, we have to think in a more abstract manner: one major aspect of a control application is the visualization of data and also storage of e.g. control values.
Maybe we would like to attach more than one viewing entity to a set of data, e.g. one which shows a 2D-graph of the phase plane, another one, which plots the trajectory over time.

Assuming the presented needs, the concept of one data-containing entity with all methods included for storing, loading and so on is expedient - as well as the viewing entity which includes all the necessary stuff for visualization, printing, etc.

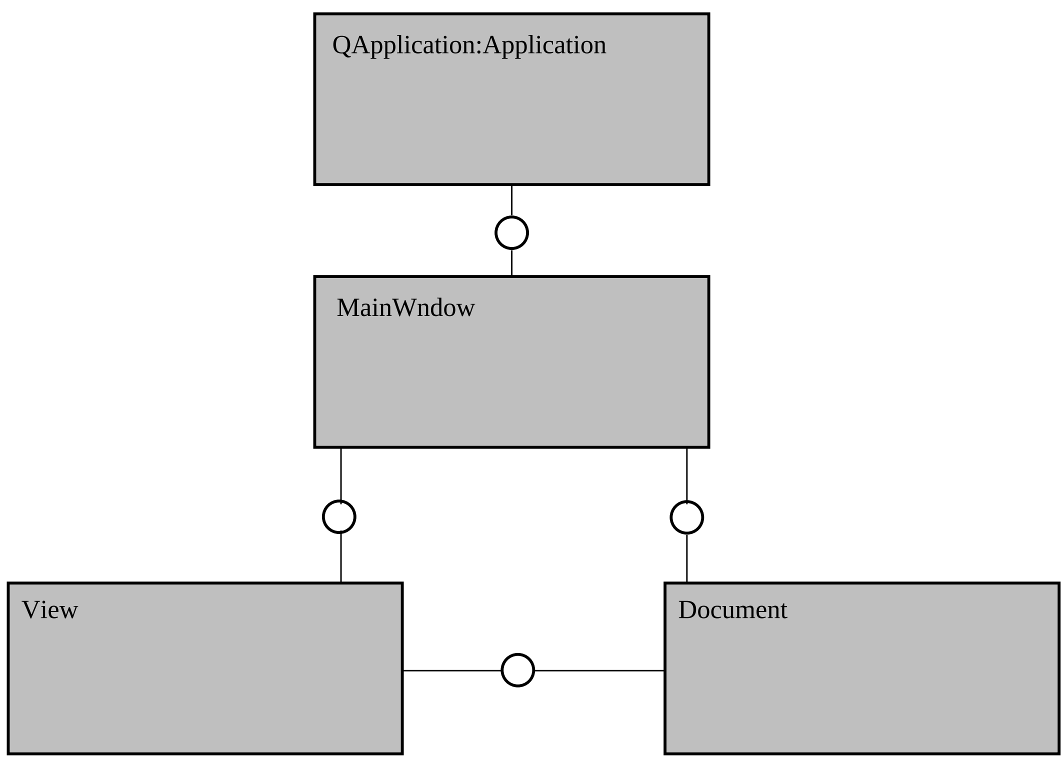


Figure 6.8: The standard application scheme

This approach can be seen as one attempt to give a unique solution to the user-interface problem and as a possibility to subdivide a user-interface into entities.

6.4 Message-Passing and Message-Filtering

the idea of message filtering is to select from a certain queue of messages those which are significant to the application or its parts.
The filtering process is due to the hierarchical concept (Application opens main

window, main window contains GUI-Elements - e.g. Buttons- etc.), which is depicted in Fig. 6.9.

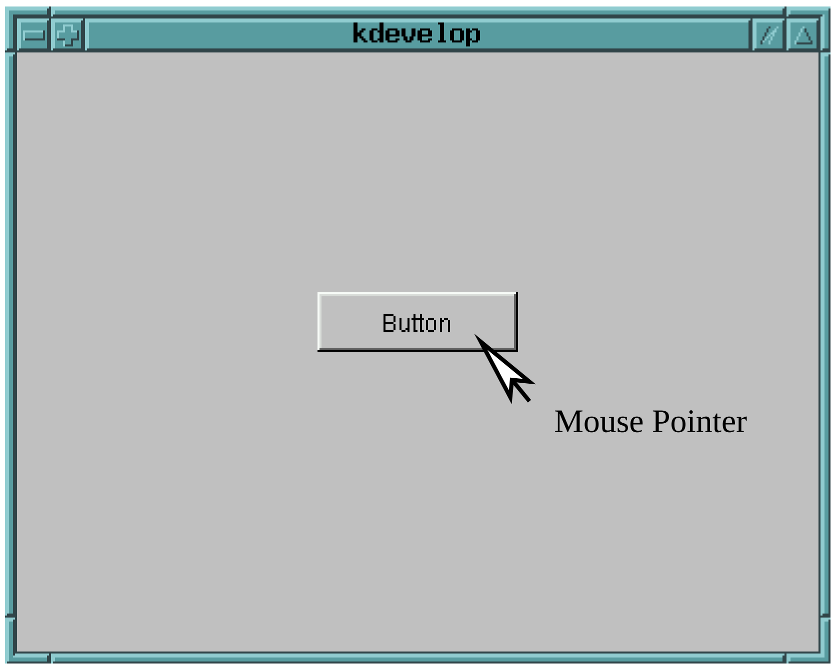


Figure 6.9: Message filtering and message passing

The message passing process can be described as follows: the message is generated by the OS (e.g. X11) and sent to the specific application - to the application of which the main window is in an active state. The application's message queue now contains the system's message. Now, the message will be hierarchically checked. In our example in 6.9, first the elements of the main window are checked, then the embedded elements such as the button. A click on the button was registered, so the message is removed from the queue and the appropriate function in the button-entity (a so called slot) is invoked.

The predescribed mechanism of message-filtering is most often not implemented in a pure, object oriented manner. As it is not our aim to build up new user-interface kits, we terminate the brief introduction of interface-internals here.

6.5 User Interface Design with QT

This section gives an introduction into practical user-interface programming. First of all, we set up the main application entity - in the same manner as in the previous section:

```
#include <qapplication.h>
```

```
#include <qfont.h>

#include "testgui.h"

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  a.setFont(QFont("helvetica", 12));

  Mainwindow *testgui=new Mainwindow();
  a.setMainWidget(testgui);

  testgui->setCaption("Buttons and More");
  testgui->show();

  return a.exec();
}
```

As we do not deal with data-handling here, we do not need any view or document entities. We just include our main window, which is of type MainWindow. The class definition for simpleapp is given as follows:

```
#include <qwidget.h>
#include <qpushbutton.h>
#include <qlcdnumber.h>
#include <qframe.h>

class Mainwindow : public QFrame
{
   Q_OBJECT
public:
  Mainwindow(QWidget *parent=0, const char *name=0);
  ~Mainwindow();

protected:
  void initDialog();
  //Elements from the QT-library
  QPushButton *QPushButton_Up;
  QLCDNumber *QLCDNumber_1;
  QPushButton *QPushButton_Down;

//For message handling
public slots:
  void UpPressed();
  void DownPressed();

//Data member
```

```
public:
  int m_value;
};
```

The interface elements are set up by the constructor (by calling initDialog()) of Mainwindow which is given below

```
#include "mainwindow.h"
#include "qlcdnumber.h"

Mainwindow::Mainwindow(QWidget *parent, const char *name) :
            QFrame(parent,name)
{
    //Here we first init the dialog
    initDialog();

    //Set additional styles
    QLCDNumber_1->setSegmentStyle(QLCDNumber::Filled);

    //Then we connect messages to slots
    connect(QPushButton_Up, SIGNAL(clicked()),
            this,SLOT(UpPressed()));
    connect(QPushButton_Down, SIGNAL(clicked()),
            this,SLOT(DownPressed()));

    //Reset data member m_value;
    m_value=0;
}

Mainwindow::~Mainwindow()
{
}

void Mainwindow::UpPressed()
{
 //Change value
 m_value++;
 //Change display
 QLCDNumber_1->display(m_value);
}

void Mainwindow::DownPressed()
{
 //Change value
 m_value--;
 //Change display
 QLCDNumber_1->display(m_value);
}
```

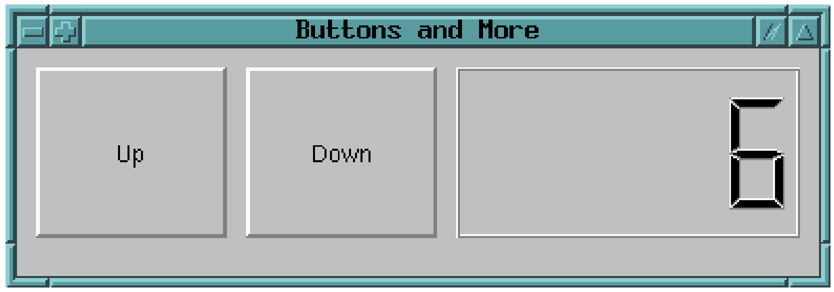


Figure 6.10: A GUI realized with QT-elements

Two important aspects can be found in the code above: the setup of the graphical user interface and in addition to that the connection of messages created by the buttons to slots (member functions which are called after a given signal - the message is obtained.

```
connect(QPushButton_Up, SIGNAL(clicked()),
        this,SLOT(UpPressed()));
connect(QPushButton_Down, SIGNAL(clicked()),
        this,SLOT(DownPressed()));
```

The following member function sets up the buttons and the LCD-display embedded in the main window:

```
void  Mainwindow::initDialog(){
  this->resize(420,120);
  this->setMinimumSize(0,0);
  QPushButton_Up= new QPushButton(this,"NoName");
  QPushButton_Up->setGeometry(10,10,100,90);
  QPushButton_Up->setMinimumSize(0,0);
  QPushButton_Up->setText("Up");

  QLCDNumber_1= new QLCDNumber(this,"NoName");
  QLCDNumber_1->setGeometry(230,10,180,90);
  QLCDNumber_1->setMinimumSize(0,0);
  QLCDNumber_1->display("0");

  QPushButton_Down= new QPushButton(this,"NoName");
  QPushButton_Down->setGeometry(120,10,100,90);
  QPushButton_Down->setMinimumSize(0,0);
  QPushButton_Down->setText("Down");
}
```

Chapter 7

A control loop simulator

This chapter deals with an example implementation of a control loop simulator. Herein we are going to apply all the informations and knowledge we collected till now. Our control loop simulator will be suited for sampled data systems. For this first approach to a complex system, we assume only single input single output systems.
Although, we are not implementing a fully dynamic user interface in the sense that the user can select and specify tranfer function blocks, we still integrate our control system into a dynamic structure with the meta-system and data handling concepts which were elaborated in the previous chapters.

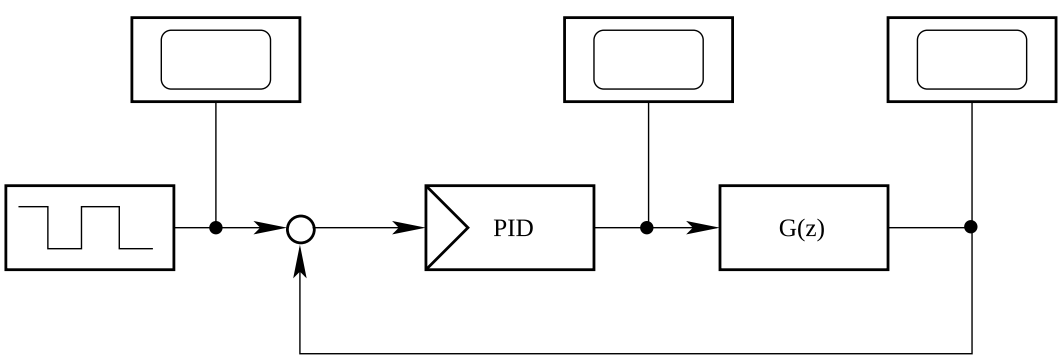


Figure 7.1: A basic control loop setup

Our aim is the simulation of structures as given in Fig. 7.1.

7.1 System theoretic approach

We are dealing with sampled data systems and thus with transfer functions in the $\mathcal{Z}$-domain[1] For the transfer function of a second order linear system in the

[1] For the derivation of relations between Laplace and $\mathcal{Z}$-domain, see Pandit [11] or Foellinger [4]

$\mathcal{Z}$-domain, we obtain the following difference-equation:

$$y(k) + a_1 y(k+1) + a_2 y(k+2) = b_0 u(k) + b_1 u(k-1) + b_2 u(k-2) \quad (7.1)$$

In most real world applications, the actual value of the output of a plant $y(k)$ in the k-th sample step is not influenced by the input (no feedthrough). Hence we can write:

$$y(k) + a_1 y(k+1) + a_2 y(k+2) = +b_1 u(k-1) + b_2 u(k-2) \quad (7.2)$$

this yields using $\mathcal{Z}$-transform:

$$Y(z)(1 + a_1 z^{-1} + a_2 z^{-2}) = (b_1 z^{-1} + b_2 z^{-2}) U(z) \quad (7.3)$$

and the resulting transfer function is given by:

$$\frac{Y(z)}{U(z)} = \frac{(b_1 z^{-1} + b_2 z^{-2})}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (7.4)$$

The output of a transfer function block in the k-th sample step is then given by:

$$y(k) = -a_1 y(k-1) - a_2 y(k-2) + b_1 u(k-1) + b_2 u(k-2) \quad (7.5)$$

As in continuous time systems, the poles of the tranfer function play the most important role with respect to stability analysis. Therefor we obtain:

$$\begin{aligned} G(z) &= \frac{b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \\ &= \frac{b_1 z + b_2}{z^2 + a_1 z + a_2} \end{aligned} \quad (7.6)$$

the poles of this transfer function are given by

$$z_{1,2} = -\frac{a_1}{2} \pm \sqrt{\frac{a_1^2}{4} - a_2} \quad (7.7)$$

Sampled data systems which are equivalent descriptions of stable linear continuous time systems have poles with positive real parts. The stability criterion for linear sampled data systems says, that the poles of a stable system are placed in the unit circle.

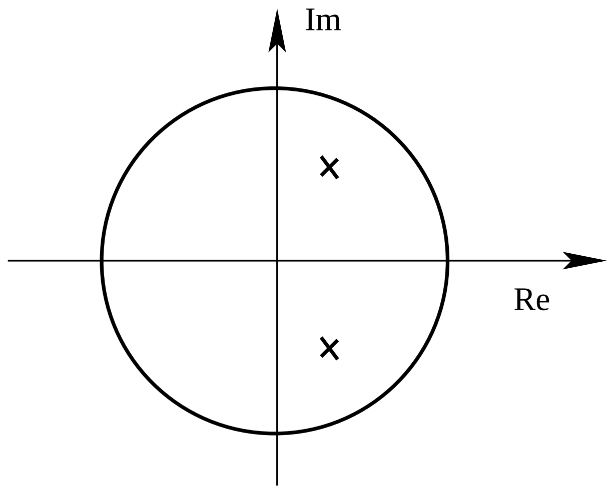


Figure 7.2: Poles of a linear sampled data system

As a sample setup, we select the following poles:

$$z_1 = 0.8 + j0.2 \quad (7.8)$$
$$z_2 = 0.8 - j0.2 \quad (7.9)$$

This yields:

$$\begin{aligned} D(z) &= (z - (0.8 + j0.2)) \cdot (z - (0.8 - j0.2)) && (7.10) \\ &= (z - 0.8 - j0.2)) \cdot (z - 0.8 + j0.2) && (7.11) \\ &= z^2 - 1.6z + 0.68 && (7.12) \end{aligned}$$

Using these parameters for the denominator of $G(z)$, we design the numerator of our plant under the assumption of a steady state amplification of 1 of the transfer block; in steady state, all $y(k+i) = y_\infty$ with $i \in \mathbb{N}_0$ and $i << k$. The same holds for $u(k+i)$. Thus we can write:

$$\begin{aligned} y(k) &= -a_1 y(k-1) - a_2 y(k-2) + b_1 u(k-1) + b_2 u(k-2) && (7.13) \\ y_\infty &= -a_1 y_\infty - a_2 y_\infty + b_1 u_\infty + b_2 u_\infty && (7.14) \\ \frac{y_\infty}{u_\infty} &= \frac{b_1 + b_2}{1 + a_1 + a_2} && (7.15) \end{aligned}$$

We assume $b_2 = \frac{1}{2} b_1$. Then we have:

$$\begin{aligned} \frac{y_\infty}{u_\infty} &= \frac{\frac{3}{2} b_1}{1 - 1.6 + 0.68} && (7.16) \\ &= \frac{\frac{3}{2} b_1}{0.08} \\ &= 1 \end{aligned}$$

From that we obtain:

$$
\begin{aligned}
b_1 &= \frac{1}{1.875} \qquad (7.17)\\
&= 0.053\\
b_2 &= 0.0265 \qquad (7.18)
\end{aligned}
$$

Now after designing our plant (in real applications, we obtain the model set up previously by identifying the dynamical system from mechanical, electrical, ... equations or by setting up physical differential equations).

In order to design the controller, we first have to set up the sampled data system equivalents of a differentiator, an integrator and a proportional factor:

- The Differentiator can be approximated by backward differentiation:

$$y_d(k+1) = e(k+1) - e(k) \qquad (7.19)$$

- The integrator is assumed as an Euler integrator:

$$y_i(k+1) = y_i(k) + e(k+1) \qquad (7.20)$$

- The proportional part of the PID controller is given by a factor:

$$y_p(k+1) = K_p \cdot e(k+1) \qquad (7.21)$$

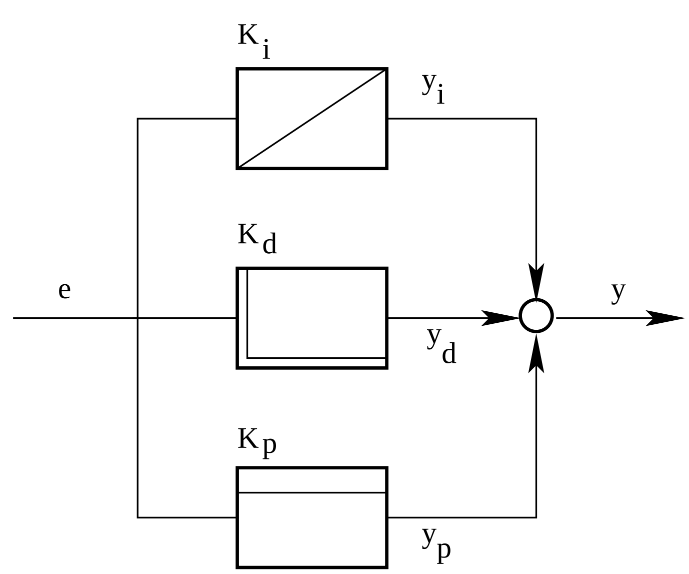


Figure 7.3: The block diagram of a PID controller

The transfer function of the given PID controller can be derived from the transfer functions of its parts. Therefor we obtain:

$$y_c(k+1) = K_i y_i(k+1) + K_d y_d(k+1) + K_p e(k+1) \tag{7.22}$$

Writing this in terms of transfer functions, we obtain:

$$G_i(z) = \frac{z}{z-1} \tag{7.23}$$

$$G_d(z) = \frac{z-1}{z} \tag{7.24}$$

$$G_p(z) = K_p \tag{7.25}$$

This yields:

$$Y(z) = E(z) \cdot \underbrace{(G_i(z) + G_d(z) + G_p(z))}_{G_c(z)} \tag{7.26}$$

$$= \frac{(K_p + K_i + K_d)z^2 - z(K_p + 2K_d) + K_d}{z^2 - z} \tag{7.27}$$

In our design, we cancel out the poles of the second order system such that the open loop transfer function of the circuit gives:

$$G_o(z) = G_c(z) \cdot G_p(z) \tag{7.28}$$

By comparing the numerator of the controller transfer function with the denominator of the plant's transfer function, we obtain

$$(K_p + K_i + K_d) = 1 \tag{7.29}$$

$$-(K_p + 2K_d) = -1.6 \tag{7.30}$$

$$K_d = 0.68 \tag{7.31}$$

$$K_p = 0.24 \tag{7.32}$$

$$K_i = 0.08 \tag{7.33}$$

We then obtain

$$G_o(z) = \frac{1}{z^2 - z} \tag{7.34}$$

$$G_{cl}(z) = \frac{G_o(z)}{1 + G_o(z)} \tag{7.35}$$

$$= \frac{1}{z^2 - z + 1} \tag{7.36}$$

The poles of the closed loop system are now given by:

$$z_{1,2} = \frac{1}{2} \pm j\frac{\sqrt{3}}{2} \tag{7.37}$$

7.2 Modeling and object oriented entitys

The next step in the design of our control loop simulator is setting up an appropriate structure. As we already discussed in chapter 4, the setup of the software for a dynamical and block oriented control system simulator resembles the structure of the real control setup. In addition to the structure, every operator should comprise the possibility for parametrization. The overall structure is again the whole setup, the control loop.

7.2.1 The basic unstructured operator block

The basic unstructured operator block is built up according to the results of chapter 4.

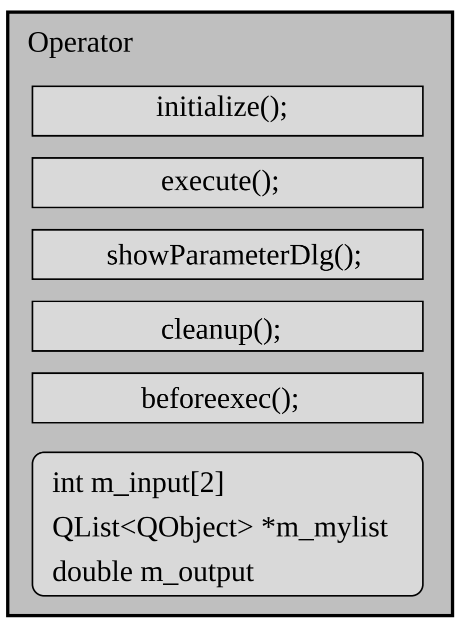


Figure 7.4: Entity Block-Diagram of the basic control operator

The following class definition determines the structure of the generic operator as given in Fig. 7.4.

```
class Operator : public QObject
{
public:
           Operator();
  virtual ~Operator();

public: //Overridables
  virtual void initialize();
```

```
  virtual void execute();
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public:

//Hard-coded outputs (no list as in the general case)
  double m_output;
  QString m_name;

//Input m_output from this operator
  int m_input[2];

//The metasystem-list
  QList<QObject> *m_mylist;
};
```

The implementation of the generic basis class contains empty functions:

```
Operator::Operator()
{
  initialize();
}

Operator::~Operator()
{
  cleanup();
}

void Operator::initialize()
{
  m_output=0;
  m_name="not available - base class";
}

void Operator::execute()
{}

void Operator::showParameterDlg()
{}

void Operator::cleanup()
{}

void Operator::beforeexec()
{}
```

The basic operator does not contain any user interface functionality.

7.2.2 The Signal source

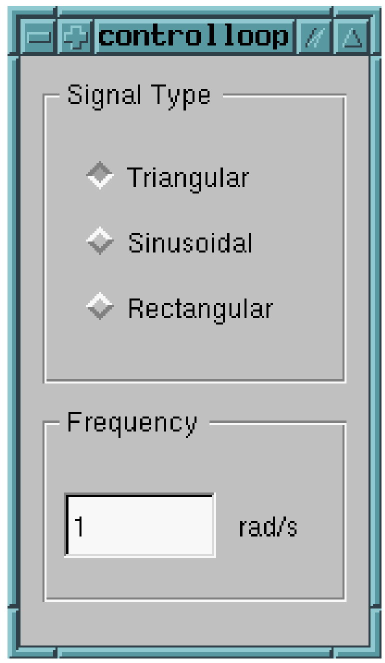


Figure 7.5: The parametrization window of the signal source

The signal source provides other operators with input signals. Three input signals will be provided: sinusoidal, triangular and rectangular. The operator csource inherits the basic operator-structure and overrides all of the basis functions. Its structure is given

```
#define MODETRIANG 0
#define MODESINE   1
#define MODERECT   2

#define DIRECTIONUP   0
#define DIRECTIONDOWN 1

class CSource : public Operator
{
public:
  CSource();
  ~CSource();

public:
  virtual void initialize();  //Override basic functionality
  virtual void execute();
```

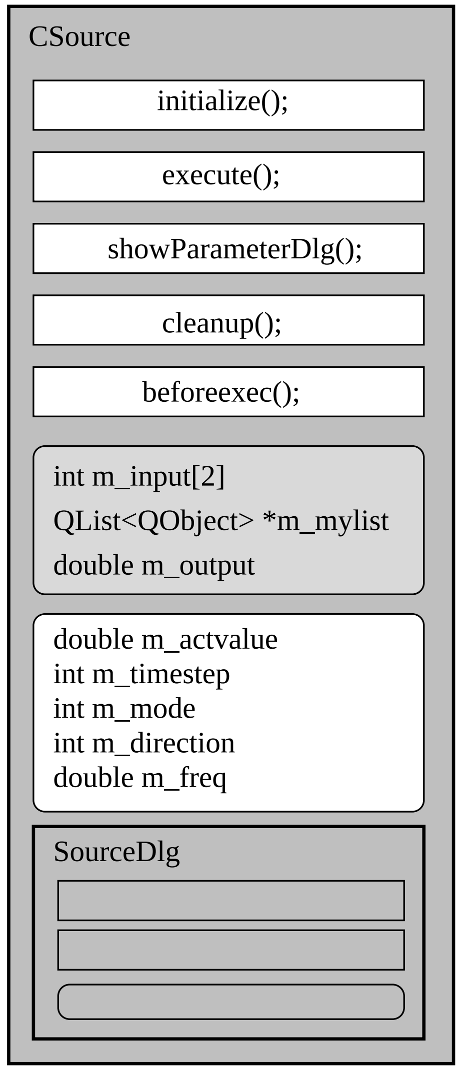


Figure 7.6: Entity Block-Diagram of the signal source

```
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public: //internal data
  double m_actvalue;
  int    m_timestep;
  int    m_mode;
  double m_freq;
  int    m_direction;

public: //Parametrization dialog
  SourceDlg m_mydlg;
};
```

The implementation of the CSource - Entity:

```
CSource::CSource()
{
  initialize();
}

CSource::~CSource()
{
  cleanup();
}

void CSource::initialize()
{
 Operator::initialize(); //Call Baseclass

 m_timestep=0;
 m_freq=1;
 m_mode=MODETRIANG;
 m_direction=DIRECTIONUP;
 m_name="Parametric Source";
 m_mydlg.setOperator(this);
}

void CSource::execute()
{
  m_timestep++;
  switch (m_mode)
  {
     case MODETRIANG :
     {
        if ((m_actvalue>100)&&(m_direction==DIRECTIONUP))
           m_direction=DIRECTIONDOWN;
        if ((m_actvalue<-100)&&(m_direction==DIRECTIONDOWN))
           m_direction=DIRECTIONUP;

        if (m_direction==DIRECTIONUP)
           m_actvalue++;
        else m_actvalue--;

        m_output=m_actvalue;
        break;
     }
     case MODESINE :
     {
        m_output=sin(m_freq*m_timestep/100);
        break;
```

```
     }
     case MODERECT :
     {
        if ((m_actvalue>100)&&(m_direction==DIRECTIONUP))
           m_direction=DIRECTIONDOWN;
        if ((m_actvalue<-100)&&(m_direction==DIRECTIONDOWN))
           m_direction=DIRECTIONUP;

        if (m_direction==DIRECTIONUP) m_actvalue++;
        else m_actvalue--;

        if (m_actvalue>0) m_output=100;
        else m_output=-100;

        break;
     }
  }
  //increment Time-step counter
  m_timestep++;
  Operator::execute(); //Call Baseclass
}

void CSource::showParameterDlg()
{
  m_mydlg.show();
  Operator::showParameterDlg(); //Call Baseclass
}

void CSource::cleanup()
{
  Operator::cleanup(); //Call Baseclass
}

//Execute this before calling exec (partly initializes)
void CSource::beforeexec()
{
 Operator::beforeexec(); //call base class
 m_timestep=0;
 m_output=0;
 m_actvalue=0;
 m_direction=DIRECTIONUP;
}
```

The source is parametrized through the adjoint dialog window.

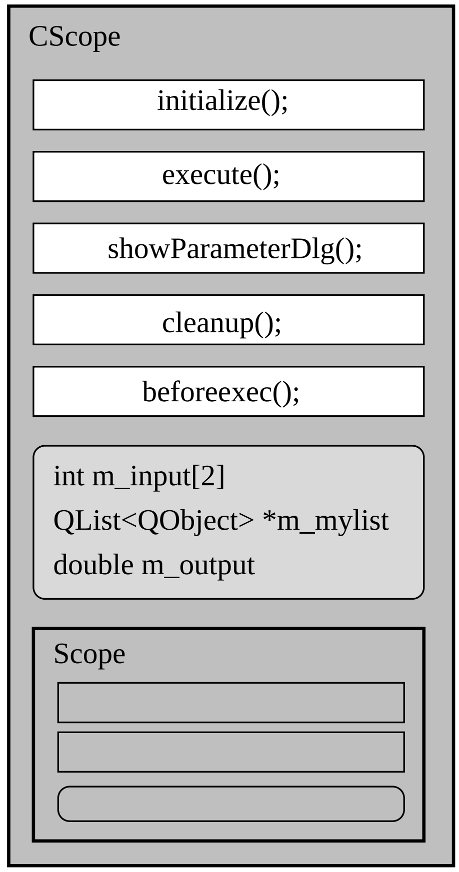


Figure 7.7: Entity Block-Diagram of the signal sink

7.2.3 The Signal sink (Oscilloscope)

```
class CScope : public Operator
{
public:
        CScope();
        ~CScope();

public:
  virtual void initialize();  //Override basic functionality
  virtual void execute();
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public:
  Scope m_myscope; //Scope-screen
};

CScope::CScope()
```

```
{
  initialize();
}

CScope::~CScope()
{
  cleanup();
}

void CScope::initialize()
{
 Operator::initialize(); //Call Baseclass
 m_name="Oscilloscope Screen";
 m_myscope.RemoveValues();

}

void CScope::execute()
{
  double value=((Operator*)m_mylist->at(m_input[0]))->m_output;
  m_myscope.AppendValue(value);
  m_myscope.update();
  Operator::execute(); //Call Baseclass
}

void CScope::showParameterDlg()
{
  m_myscope.show();
  Operator::showParameterDlg(); //Call Baseclass
}

void CScope::cleanup()
{
  Operator::cleanup(); //Call Baseclass
}

//Execute this before calling exec (partly initializes)
void CScope::beforeexec()
{
 Operator::beforeexec(); //call base class
 m_myscope.RemoveValues(); //Cleanup Scope
}
```

7.2.4 The PID controller

```
class CPIDController : public Operator
```

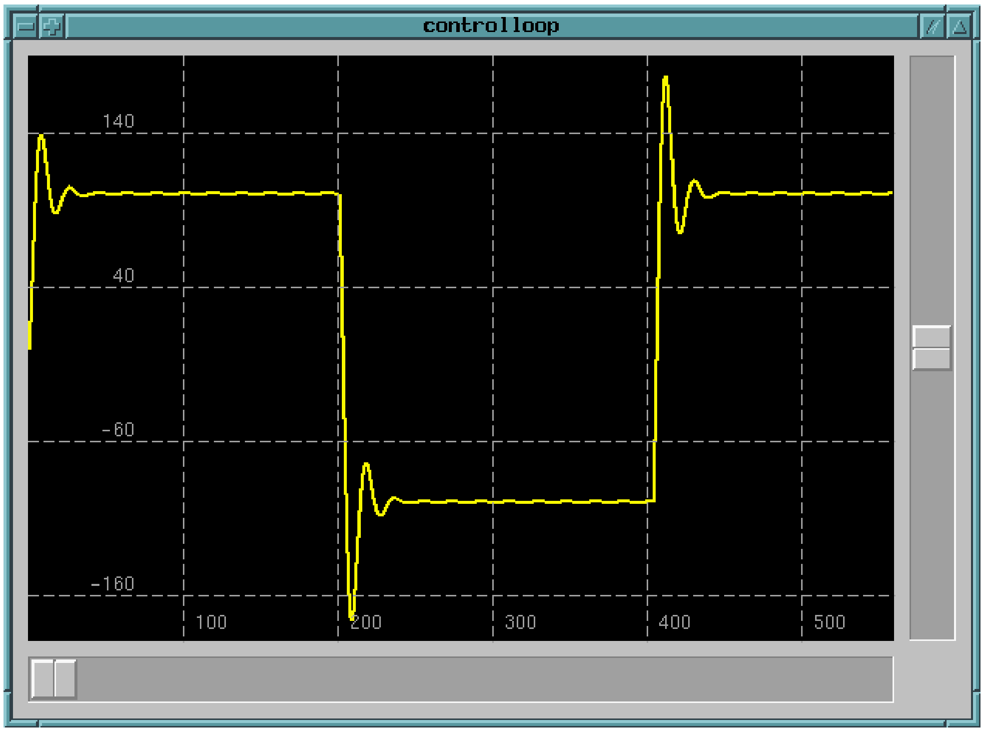


Figure 7.8: The signal sink - a dynamic oscilloscope screen

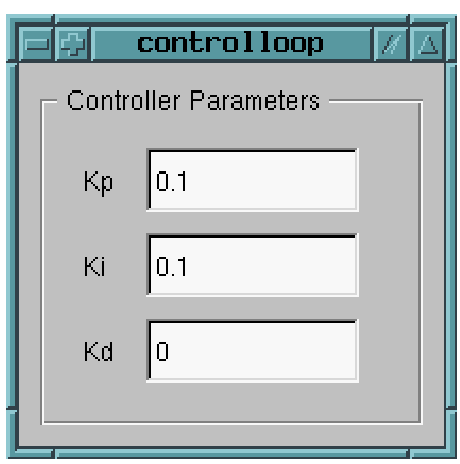


Figure 7.9: parametrization of the PID controller

```
{
public:
        CPIDController();
        ~CPIDController();

public:
  virtual void initialize();  //Override basic functionality
  virtual void execute();
```

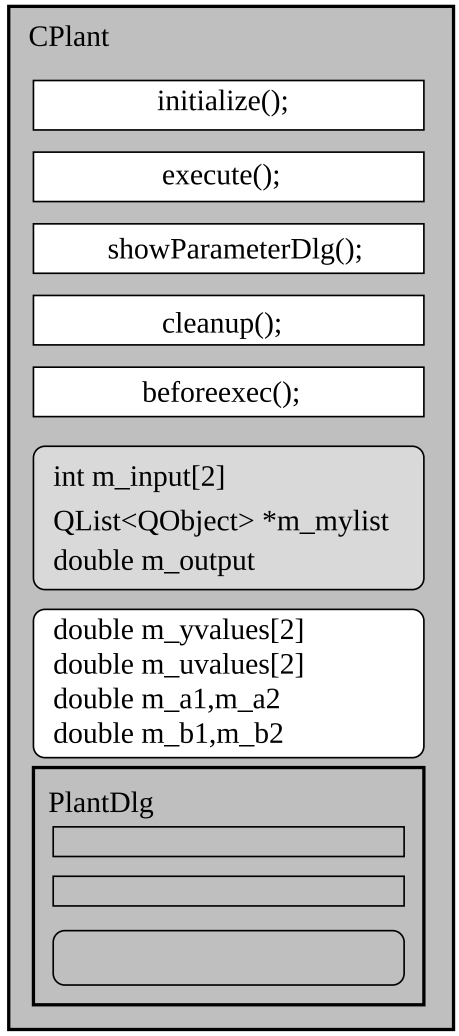


Figure 7.10: Entity Block-Diagram of the PID-controller

```
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public:
  double m_integral;
  double m_evalues[2];
  double m_kp,m_ki,m_kd;
public:
        PidDlg m_mydlg;
};
```

Implementation:

```
CPIDController::CPIDController()
```

```
{
  initialize();
}

CPIDController::~CPIDController()
{
  cleanup();
}

void CPIDController::initialize()
{
 Operator::initialize(); //Call Baseclass
 m_name="PID-Controller";
 m_evalues[0]=0;
 m_evalues[1]=0;
 m_kp=0.1;
 m_kd=0;
 m_ki=0.1;
 m_integral=0;
 m_mydlg.setOperator(this);
}

void CPIDController::execute()
{
  //The difference-equation of the plant
  //out=kp*e(t) + ki *integral(e(t))+ kd*differential(e(t))

  //First shift e's
  for(int i=0;i<1;i++)
  {
   m_evalues[i+1]=m_evalues[i];
  }
  //Get actual control error value
  m_evalues[0]=((Operator*)m_mylist->at(m_input[0]))->m_output;
  //Integrate control error
  m_integral+=m_evalues[0];

  m_output=m_kp*m_evalues[0]+
           m_kd*(m_evalues[0]-m_evalues[1])+
           m_ki*m_integral;

  Operator::execute(); //Call Baseclass
}

void CPIDController::showParameterDlg()
{
  m_mydlg.show();
```

```
  Operator::showParameterDlg(); //Call Baseclass
}

void CPIDController::cleanup()
{
  Operator::cleanup(); //Call Baseclass
}

//Execute this before calling exec (partly initializes)
void CPIDController::beforeexec()
{
 Operator::beforeexec(); //call base class
 m_evalues[0]=0;
 m_evalues[1]=0;
 m_integral=0;  //Reset Integrator
}
```

7.2.5 The Plant model

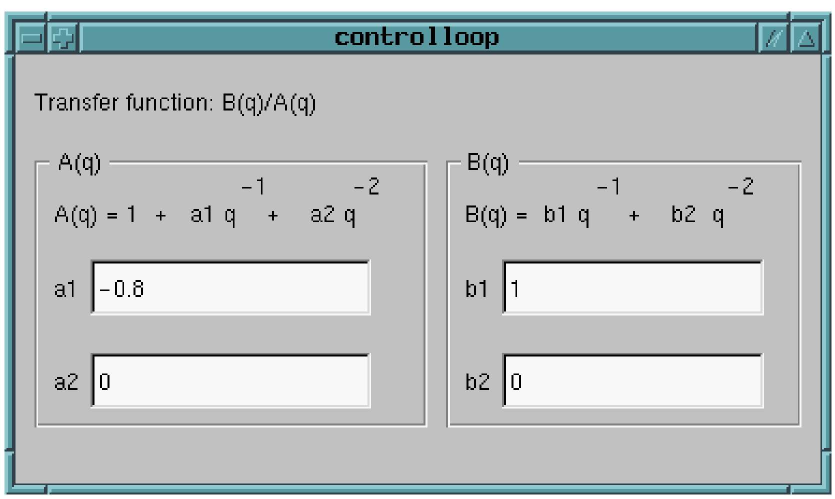


Figure 7.11: Configuration of the transfer function window

```
class CPlant : public Operator
{
public:
        CPlant();
        ~CPlant();

public:
  virtual void initialize();  //Override basic functionality
  virtual void execute();
```

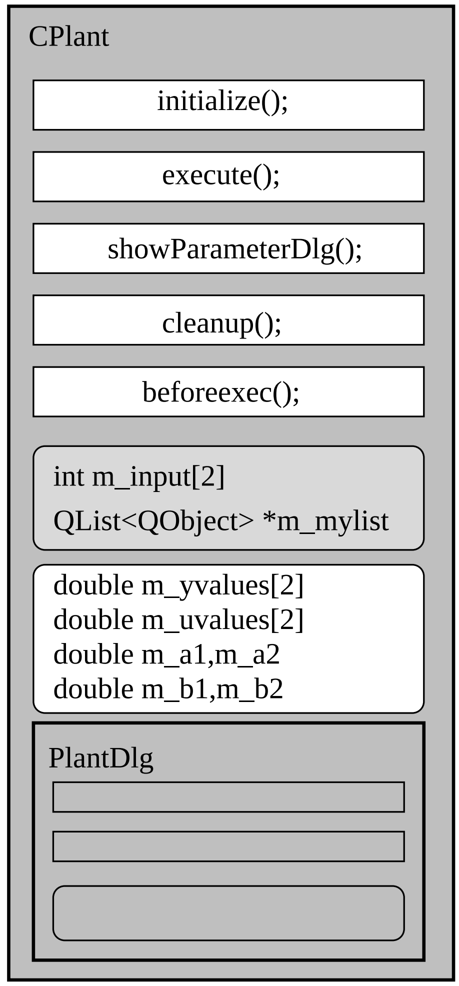


Figure 7.12: Entity Block-Diagram of the plant model

```
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public:
  double m_yvalues[2];
  double m_uvalues[2];
  double m_a1,m_a2,m_b1,m_b2;

public:
        PlantDlg m_mydlg;
};
```

Implementation of the plant model:

```
CPlant::CPlant()
{
```

```
  initialize();
}

CPlant::~CPlant()
{
  cleanup();
}

void CPlant::initialize()
{
 Operator::initialize(); //Call Baseclass
 m_name="PT2-System";
 m_yvalues[0]=0;
 m_yvalues[1]=0;
 m_uvalues[0]=0;
 m_uvalues[1]=0;
 m_a1=-0.8;
 m_a2=0;
 m_b1=1;
 m_b2=0;
 m_mydlg.setOperator(this);
}

void CPlant::execute()
{
  //The difference-equation of the plant
  //y(t)+a1*y(t-1)+a2*y(t-2)=b1*u(t-1)+b2*u(t-2)
  //y(t)=b1*u(t-1)+b2*u(t-2)-a1*y(t-1)-a2*y(t-2)

  //First shift u's and y's
  for(int i=0;i<1;i++)
  {
   m_yvalues[i+1]=m_yvalues[i];
   m_uvalues[i+1]=m_uvalues[i];
  }
  //Insert input and last output


  m_output=-m_a1*m_yvalues[0]-m_a2*m_yvalues[1]+
            m_b1*m_uvalues[0]+m_b2*m_uvalues[1];

  m_yvalues[0]=m_output;
  m_uvalues[0]=((Operator*)m_mylist->at(m_input[0]))->m_output;

  Operator::execute(); //Call Baseclass
}
```

```
void CPlant::showParameterDlg()
{
  m_mydlg.show();
  Operator::showParameterDlg(); //Call Baseclass
}

void CPlant::cleanup()
{
  Operator::cleanup(); //Call Baseclass
}

//Execute this before calling exec (partly initializes)
void CPlant::beforeexec()
{
 Operator::beforeexec(); //call base class
 m_yvalues[0]=0;
 m_yvalues[1]=0;
 m_uvalues[0]=0;
 m_uvalues[1]=0; //Reset stored values
}
```

7.2.6 The sum point

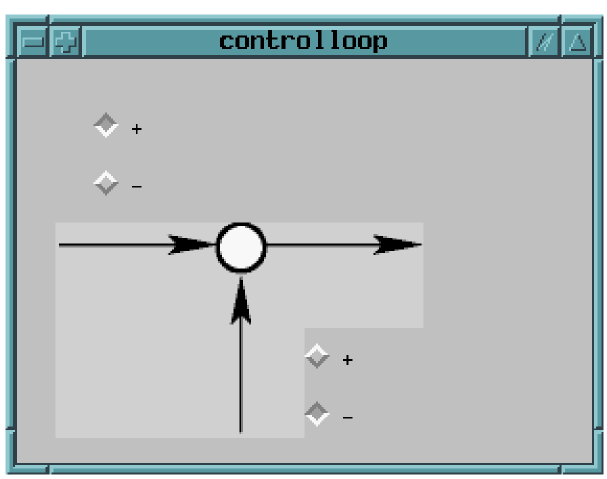


Figure 7.13: Parametrization for the summation point

```
class CSumPoint : public Operator
{
public:
        CSumPoint();
        ~CSumPoint();
  virtual void initialize();  //Override basic functionality
```

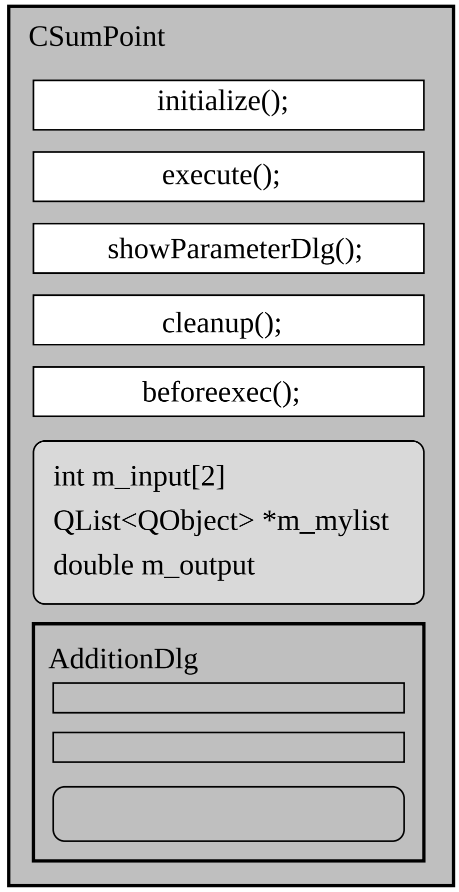


Figure 7.14: The entity diagram for the sumpoint operator

```
  virtual void execute();
  virtual void showParameterDlg();
  virtual void cleanup();
  virtual void beforeexec();

public:
  AdditionDlg m_mydlg;
  int m_yvalues[3];
  int m_uvalues[3];
  int m_sign[2];
};

CSumPoint::CSumPoint()
{
  initialize();
}

CSumPoint::~CSumPoint()
{
```

```
  cleanup();
}

void CSumPoint::initialize()
{
 Operator::initialize(); //Call Baseclass
 m_name="Summation Point";
 m_sign[0]=+1; //Set Signs
 m_sign[1]=-1;
 m_mydlg.setOperator(this);
}

void CSumPoint::execute()
{
  double value1=((Operator*)m_mylist->at(m_input[0]))->m_output;
  double value2=((Operator*)m_mylist->at(m_input[1]))->m_output;

  m_output=m_sign[0]*value1+m_sign[1]*value2;
    Operator::execute(); //Call Baseclass
}

void CSumPoint::showParameterDlg()
{
  m_mydlg.show();
  Operator::showParameterDlg(); //Call Baseclass
}

void CSumPoint::cleanup()
{
  Operator::cleanup(); //Call Baseclass
}

//Execute this before calling exec (partly initializes)
void CSumPoint::beforeexec()
{
 Operator::beforeexec(); //call base class
 m_output=0;
}
```

7.2.7 Meta System Construction

The meta system for the control loop simulator has to execute the operators operation kernel, to initialize the operators and to cleanup after stopping the simulation. The execution is performed using the framework of an object oriented list containing the operator blocks ad furthermore using the concepts of dynamic coupling of the operators by providing the operators with a pointer to the operator list and by providing the index of the operator in the list of which the parameters have to be retrieved. Figure 7.15 shows the structure.

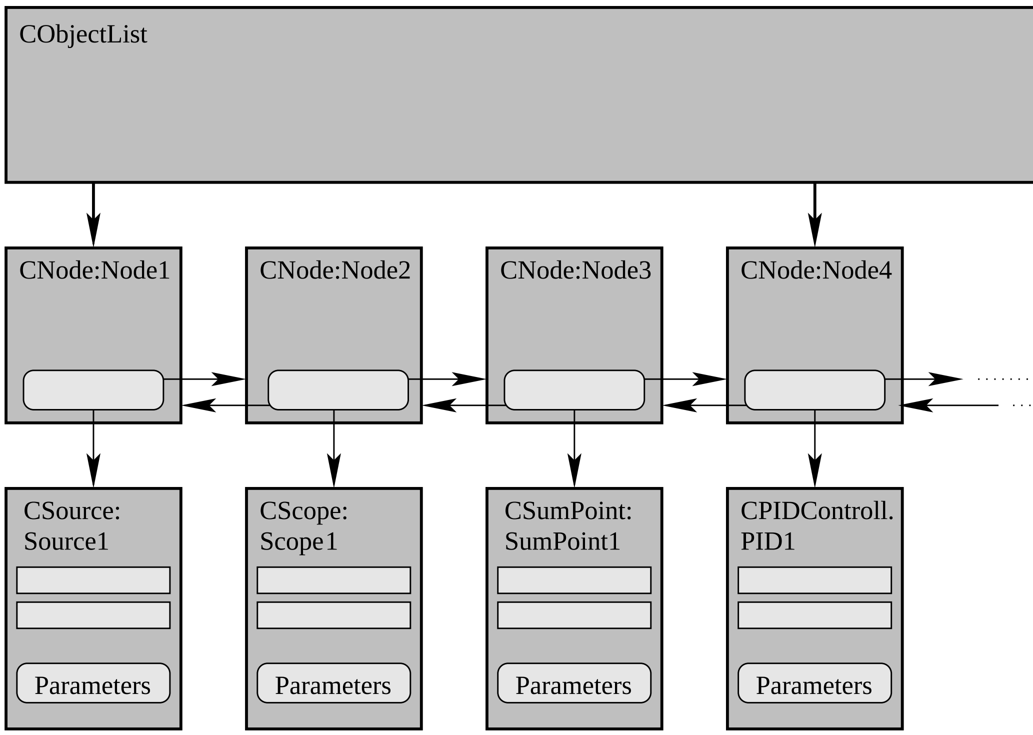


Figure 7.15: Entity Block-Diagram of the meta system

```
class Simulator : public QWidget
{
   Q_OBJECT
public:
 Simulator(QWidget *parent=0, const char *name=0);
 ~Simulator();

protected:
 void initDialog();

 QSpinBox *QSpinBox_1;
 QLabel *QLabel_1;
 QPushButton *QPushButton_run;
 QListBox *QListBox_1;

protected:
 void paintEvent(QPaintEvent* Event);

public:
  QPixmap m_Blockdiag;
```

```
  //The list for execution of the operators
  QList<Operator>  m_Operatorlist;

  bool  m_wasstopped;
  QTimer m_timer;
  int m_timesteps;
  int m_length;

public slots:
  void Simulength();
  void Runandstop();
  void ShowConfDialog();
  void execSlot();
};
```

The implementation

```
Simulator::Simulator(QWidget *parent, const char *name)
           : QWidget(parent,name)
{
  initDialog();

  //Load Image
  m_Blockdiag.load("../ctrllpsim.pnm");

  //Hard-wired control loop simulator
  m_Operatorlist.setAutoDelete(TRUE);

  //This functionality would be handled by the user-interface in the
  //fully dynamical case (creates hard-wired control loop)
  CSource* m_source=new(CSource);
  CSumPoint* m_sumpoint=new(CSumPoint);
  CPIDController* m_Pidcontroller=new(CPIDController);
  CPlant* m_plant=new(CPlant);
  CScope* m_scope1=new(CScope);
  CScope* m_scope2=new(CScope);
  CScope* m_scope3=new(CScope);

  //Insert into interpreted list
  m_Operatorlist.append(m_source);
  m_Operatorlist.append(m_sumpoint);
  m_Operatorlist.append(m_Pidcontroller);
  m_Operatorlist.append(m_plant);
  m_Operatorlist.append(m_scope1);
  m_Operatorlist.append(m_scope2);
  m_Operatorlist.append(m_scope3);
```

```
//Make m_Operatorlist known to blocks
m_source->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_sumpoint->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_Pidcontroller->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_plant->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_scope1->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_scope2->m_mylist=(QList<QObject>*)&m_Operatorlist;
m_scope3->m_mylist=(QList<QObject>*)&m_Operatorlist;

//Make connections between blocks
m_sumpoint->m_input[0]=0;       //output from source
m_sumpoint->m_input[1]=3; //output from plant

m_Pidcontroller->m_input[0]=1;  //output from sumpoint
m_Pidcontroller->m_input[1]=0;  //not connected

m_plant->m_input[0]=2;  //output from controller
m_plant->m_input[1]=0; //not connected

m_scope1->m_input[0]=0; //Output from generator
m_scope1->m_input[1]=0; //not connected

m_scope2->m_input[0]=2; //Output from controller
m_scope2->m_input[1]=0; //not connected

m_scope3->m_input[0]=3; //Output from plant
m_scope3->m_input[1]=0; //not connected


//Insert Items in ListBox
unsigned int i;
for (i=0;i<m_Operatorlist.count();i++)
{
  //Insert from inside operator
  QListBox_1->insertItem(m_Operatorlist.at(i)->m_name,i);
}

//Create a timer for periodic execution
m_timer.start(10,FALSE); //Start timer

//Set initial number of timesteps
m_length=1000;
QSpinBox_1->setValue(m_length);

//Connection of Slots
connect(QSpinBox_1,SIGNAL(valueChanged(int)),this,SLOT(Simulength()));
connect(QPushButton_run,SIGNAL(clicked()),this,SLOT(Runandstop()));
```

```
  connect(QListBox_1,SIGNAL(selected(int)),this,SLOT(ShowConfDialog()));
  connect(&m_timer,SIGNAL(timeout()),this,SLOT(execSlot()));

  m_wasstopped=TRUE;
}

Simulator::~Simulator()
{
 unsigned int i;
 //Stop execution
 m_wasstopped=TRUE;

 //Stop Timer
 m_timer.start(50,FALSE);

 //Remove list and entries (due to autodelete)
 for(i=0;i<m_Operatorlist.count();i++)
  m_Operatorlist.removeLast();
}

void Simulator::paintEvent(QPaintEvent* Event)
{
 QPainter paint( this );
 paint.drawPixmap(10,55, m_Blockdiag,0,0,-1,-1 );
}

//User-Interface functionality
void Simulator::Simulength()
{
 m_length=QSpinBox_1->value();
}
```

The following program code is also part of the meta system. Herein the operators are prepared and made "ready to run":

```
void Simulator::Runandstop()
{
 unsigned int i;
 if (QPushButton_run->isOn())
 {
   if (m_wasstopped==TRUE)
   {
     for (i=0;i<m_Operatorlist.count();i++)
     {
      m_Operatorlist.at(i)->beforeexec(); //Prepare for execution
      m_timesteps=0;
     }
```

```
    m_wasstopped=FALSE;
  }
 }
 else
 {
    m_wasstopped=TRUE;
 }
}
```

We introduced user interface dialogs for the configuration of the operator blocks. These configuration windows are opened if the respective operator is selected in the simulator's main window:

```
void Simulator::ShowConfDialog()
{
  int index=QListBox_1->currentItem();

  if (index>-1) //nothing selected ?
    m_Operatorlist.at(index)->showParameterDlg(); //Open Parameter window
}
```

The following routine is executed using a timer triggering. Actually for simulation purposes, it could also be executed in a cyclic manner. The timer method however allows to change the processor load by decreasing or increasing the timer intervals. In this routine, one can see that the execute(); member function of the operator blocks is called for all of the operators of the list.

```
/** Timer shot */

void Simulator::execSlot()
{
 unsigned int i;
 if ((m_wasstopped==FALSE)&&(m_timesteps<m_length))
 {
     for (i=0;i<m_Operatorlist.count();i++)
     {
      m_Operatorlist.at(i)->execute();      //Execute Operators
     }
 }
 if (m_timesteps>=m_length)
 {
   QPushButton_run->setOn(FALSE);
   Runandstop();
 }
 m_timesteps++;
}
```

7.3 The User Interface

The construction of the User Interface will be shown only for one example, the scope window.

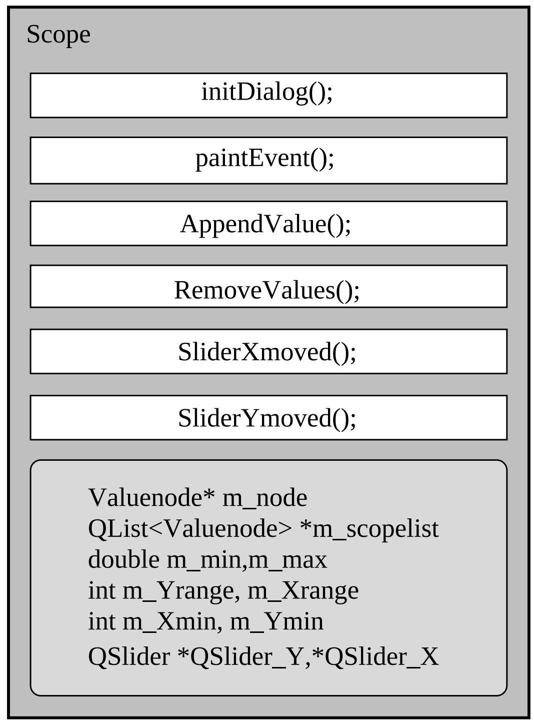


Figure 7.16: Entity Block-Diagram of the scope entity (user interface for CScope)

Definition:

```
class Scope : public QWidget  {
  Q_OBJECT

public:
 Scope(QWidget *parent=0, const char *name=0);
 ~Scope();

protected:
 void initDialog();

 QSlider *QSlider_Y;
 QSlider *QSlider_X;

protected:
  void paintEvent(QPaintEvent* myEvent);

public:
```

```
  //Contains a QList
  QList<Valuenode> m_scopelist;
  Valuenode* m_node;

//Set maximum possible values
  void Setmaxvalues();
  void setSliderRanges();

public slots:
  void AppendValue(double value);
  void RemoveValues();
  void SliderXmoved();
  void SliderYmoved();

public:
  double m_min;
  double m_max;

  int m_Yrange;
  int m_Ymin;
  int m_Xrange;
  int m_Xmin;

};
```

Implementation:

```
Scope::Scope(QWidget *parent, const char *name) : QWidget(parent,name)
{
  int i;
  initDialog();

  //Setze Werte m_min und m_max
  m_min=+10E10;
  m_max=-10E10;

  //Setze richtige Werte für Slider

  setSliderRanges();

  //Allow auto-deletion of List-elements
  m_scopelist.setAutoDelete( TRUE );


  //Alle Events verbinden
  connect(QSlider_Y, SIGNAL(valueChanged(int)), this, SLOT(SliderXmoved()));
  connect(QSlider_X, SIGNAL(valueChanged(int)), this, SLOT(SliderYmoved()));
```

```
}

Scope::~Scope()
{
}

//Paint the values according to settings
void Scope::paintEvent(QPaintEvent* myEvent)
{
 int i;
 //QWidget::paintEvent(myEvent);
 QPainter paint(this);

 //Make Background black
 QBrush blackbrush(black);
 paint.fillRect(10,10,560,380, blackbrush);

 //First calculate Scaling factors

 QPen yellowpen(yellow);
 yellowpen.setWidth(2);

 paint.setPen(yellowpen);

 int startx=QSlider_X->value();
 int stopx=QSlider_X->value()+560;

 int maxy=QSlider_Y->value()+380/2;
 int miny=QSlider_Y->value()-380/2;

 double value;
 bool move=FALSE;

 //Do not access beyond end of list
 if (stopx>m_scopelist.count()) stopx=m_scopelist.count();

 for(i=startx;i<stopx;i++)
 {
     value=m_scopelist.at(i)->m_value;
     if (i==startx)
     {
       paint.moveTo(10,10+380/2-value+QSlider_Y->value());
     }
     else
     {
       //Check for Y-Range
       if ((value>maxy)||(value<miny))
```

```
       {
         move=TRUE;
         paint.moveTo(10+i-startx,10+380/2-value+QSlider_Y->value());
       }
       else
       {
         if (move==FALSE)
         {
          paint.lineTo(10+i-startx,10+380/2-value+QSlider_Y->value());
         }
         else
         {
          paint.moveTo(10+i-startx,10+380/2-value+QSlider_Y->value());
          move=FALSE;
         }
       }
     }
}

 //Finally draw lines and write values
 QPen graypen(gray);
 graypen.setStyle(DashLine);
 paint.setPen(graypen);

 QString string;
 //Horizontal grid
 i=50;
 while(i<380)
 {
    string.setNum(190-i+QSlider_Y->value());
    paint.drawText(0,i-5,80,20, AlignRight,string,-1,0,0);
    paint.moveTo(10,10+i);
    paint.lineTo(570,10+i);
    i+=100;
 }
 //Vertical Grid
 i=100;
 while(i<560)
 {
    string.setNum(i+QSlider_X->value());
    paint.drawText(i-40,370,80,20, AlignRight,string,-1,0,0);
    paint.moveTo(10+i,10);
    paint.lineTo(10+i,390);
    i+=100;
 }
}
```

```
//Add a value to storage
void Scope::AppendValue(double value)
{
  Valuenode* m_myvalue;
  m_myvalue=new Valuenode(value);
  m_scopelist.append(m_myvalue);
  if (m_scopelist.last()->m_value<m_min)
      m_min=m_scopelist.last()->m_value;
  if (m_scopelist.last()->m_value>m_max)
      m_max=m_scopelist.last()->m_value;
  setSliderRanges();
}

//Remove stored values
void Scope::RemoveValues()
{
  while (m_scopelist.count()>0)
  {
   m_scopelist.removeLast();
  }
}

//SliderX moved
void Scope::SliderXmoved()
{
 update();
}
//SliderY moved
void Scope::SliderYmoved()
{
 update();
}

//Set maximum possible values
void Scope::Setmaxvalues()
{
}

void Scope::setSliderRanges()
{
 QSlider_X->setRange(0,m_scopelist.count());
 QSlider_Y->setRange(m_min,m_max);
}
```

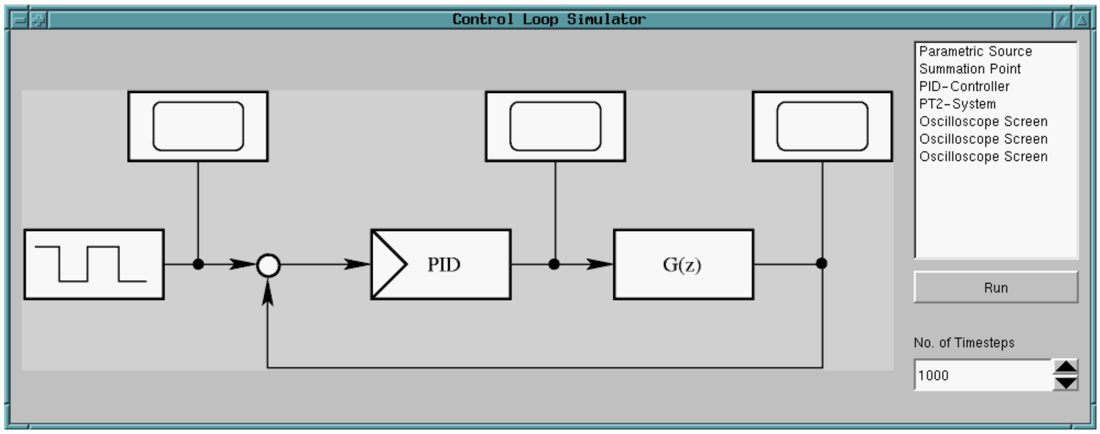


Figure 7.17: The user interface of the control loop simulator

7.4 Extensions for the fully dynamical case

The presented control loop simulator shows in principle how one could give a meaningful and dynamic structure to a signal processing system and how a complex system can be built up. However the selected approach is not fully dynamic. That is due to the non dynamic outputs and the constraints given by the user interface.

The fully dynamic approach comprises:

- A fully dynamic user interface as a part of the meta system
- Fully dynamic output handling of the blocks (list based)

However, with the knowledge of the previous chapters, the reader should be capable of extending the given platform according to his or her needs.

Chapter 8

Realtime systems and Multitasking

Before we can talk about real time systems, we have to clarify what the expression *real time* means. The most frequently occuring mistake is, that application designers talk about speed and mean realtime or vice versa. In fact, realtime can be slow and does not at all refer to speeds at all. The expression “realtime” only specifies, that a certain sample time is kept stable and stationary.

This is a very important factor, especially if we develop basis systems for higher level control systems. Especially learning or adaptive systems require a stable sampling interval.

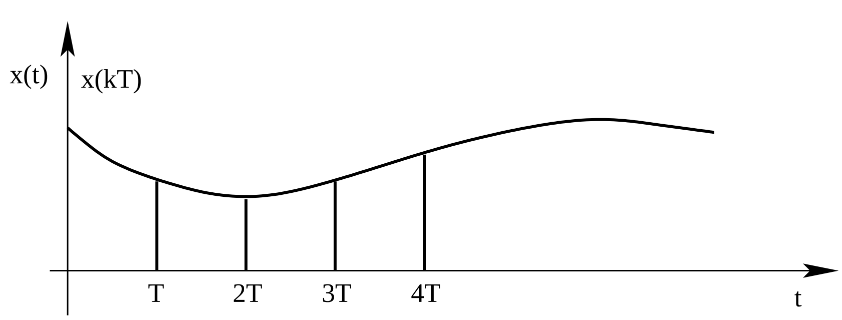


Figure 8.1: Sampling with a realtime system

8.1 The necessity of Realtime

To find out why we definitely need real time for certain classes of systems - especially in control and signal processing we must analyze the effect of non realtime i.e. a time varying sample time on the underlying plant- or system-dynamics.

First we recollect: real-time means aequidistant sampling. Especially for A/D or D/A-conversion containing applications, realtime is a very necessary aspect.

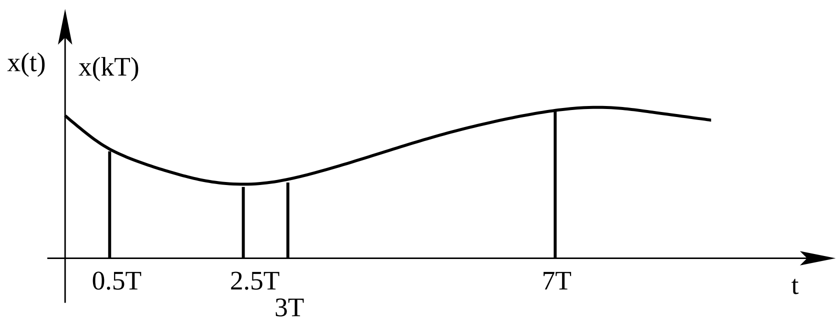


Figure 8.2: Sampling with a nonrealtime system, although the system reacts faster than required, its reaction time is not predictable and thus we cannot use it for e.g. control- or signal-processing purposes.

The possibility of guaranteeing real-time by using an adequate hardware and buffering the sampled values is only possible if we consider open loop systems with offline signal processing such as iterative learning control systems. For online control systems, a hard real-time for the whole system (except for the user-interface) is necessary.

The system will be fed with the output of a zero-order hold equivalent of a sequence which is the plant's input $u(t)$ (as you can see in Fig. 8.3)

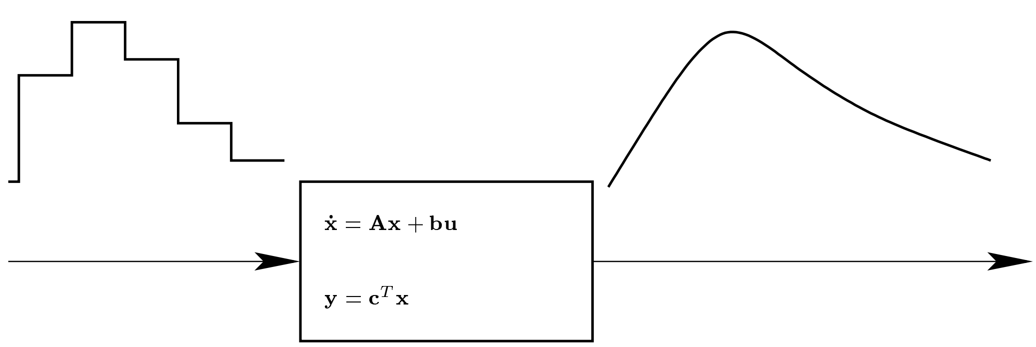


Figure 8.3: A continuous linear time-invariant state space system

This necessity will be traced in a more mathematical manner in the following example.

Example: A general way to show the effect of time-varying sample times can be shown using a continuous 2nd-order dynamical system - a type of dynamical system with which we have to deal in most control system applications:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{b}u(t) \tag{8.1}$$
$$\mathbf{y}(t) = \mathbf{c}^T\mathbf{x}(t) \tag{8.2}$$

Further, the system has to be SISO (single input, single output).

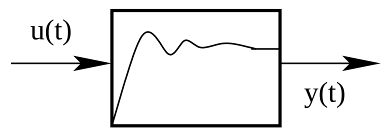


Figure 8.4: A continuous linear time-invariant state space system

For $\mathbf{A}$, $\mathbf{b}$ nd $\mathbf{c}^T$, we select:

$$\mathbf{A} = \begin{pmatrix} -1.5 & 0.11 \\ 0 & -2 \end{pmatrix} \tag{8.3}$$
$$\mathbf{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{8.4}$$
$$\mathbf{c}^T = \begin{pmatrix} 0.5 & 0.2 \end{pmatrix} \tag{8.5}$$

this is an overdamped system with poles -0.9 and -2 in the left half plane. For a unique sampling rate, we obtain:

$$\mathbf{\Phi}(T_a) = e^{\mathbf{A}T_a} \tag{8.6}$$
$$\mathbf{H}(T_a) = \mathbf{A}^{-1}\left[e^{\mathbf{A}T_a} - \mathbf{I}\right]\mathbf{B} \tag{8.7}$$

Using this approach, one has obtained the state-space representation of the sampled data system:

$$\mathbf{x}(k+1) = \mathbf{\Phi}\mathbf{x}(k) + \mathbf{H}u(k) \tag{8.8}$$
$$\mathbf{y}(k) = \mathbf{c}^T\mathbf{x}(k) \tag{8.9}$$

From equation 8.6 one gets a clear impression how the sampling time influences the matrices and hence the eigenvalues of a sampled data system. This is an important fact as e.g. a discrete state space controller which was once developed for a certain system is - under the assumption of a varying T_a - now operating on a time varying system which can cause instability or more probable will cause bad control action and system behaviour with respect to desired dynamics.

Especially “higher” control algorithms as iterative learning or predictive control or as well adaptive control - generally speaking all dynamic control algorithms which

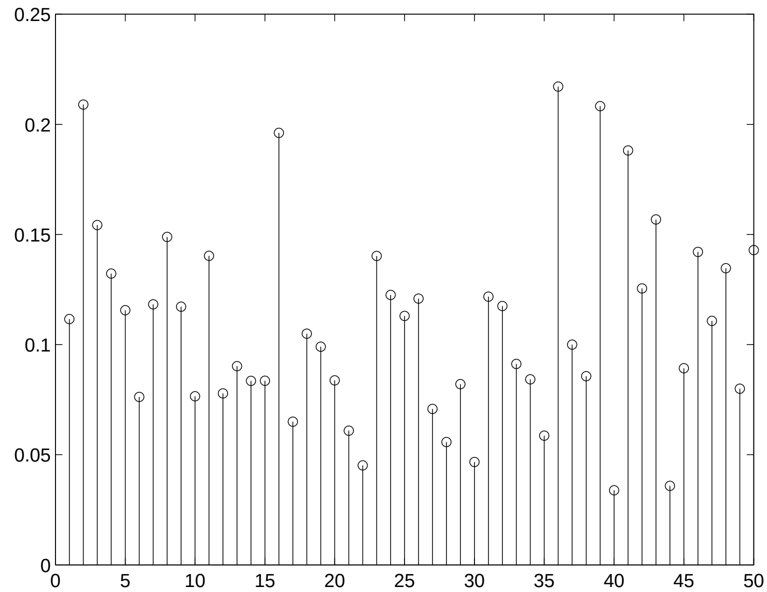


Figure 8.5: A continuous linear time-invariant state space system sampled at different increments of time. Every stem shows the increment T for the actual sampling step

use system models or specific properties of the controlled system or identification algorithms will be endangered by using unstable sample-time sampling.
The following matlab-code simulates the time-varying sample time and calculates the eigenvalues for the aequivalent sampled data system for the system given above:

```
%% Cleanup first
close all
clear all

%% Definition for a continuous time system

A=[-1.5 0.11; 0 -3]
b=[0;1]
c=[0.5 0.2]
d=0

%% Basis-Sample Time 0.1 sec
Ta=0.1

%%Deterioration of Sampling rate through gaussian noise

noise=(Ta/2)*randn(50,1);
noise=noise.*(noise>-Ta);   %avoid negative sampling times
Samplingtime=Ta+noise;
```

```
for i=1:50,
        [Phinew,Hnew] = c2d(A,b,Samplingtime(i,1));
        eigenvalsnew  = eig(Phinew);
        eigenvals     = [eigenvals eigenvalsneu];
        H             = [H Hnew];
end

figure(1);
stem(eigenvals(1,:)')

figure(2);
stem(eigenvals(2,:)')

figure(3);
stem(Samplingtime)
```

One might argument, that if we knew the sample-time or at least the time at the instants when the sample was taken, we could "repair" these effects by e.g. interpolation or by designing a special controller - although this is possible for linear systems, the problem is that this does not work for general systems and further time-varying systems sometimes require special treatment. From the technical point of view such workarounds are not at all supportable: for difficult plant-dynamics we need higher level control systems with a clean realtime basis.

8.2 Provision of real time in practice

The easiest way to build up a realtime system is not to use any preemtive multitasking systems. One could e.g. use DOS and as programming language C with RT-Kernel, a realtime scheduler extension.

As only one user can work at the same time, the computational load is deterministic and can be predetermined. The realtime extension takes over the interrupt handling and all device I/O. Thus, the whole system ressources are reserved for one process (containing one or more tasks with different priorities).

Another possibility is to use a special software-kit which provides realtime. There are different kits provided for all platforms. Though all of them promise best realtime performance, many of these kits have to face strong limitations which are partly created by the kit itself, partly by the underlying OS. If e.g. message queues can overflow or harddisk access is delayed extremely, the underlying OS is not well suited to realtime applications, even with an additional kit. Suitable solutions are e.g. RTLinux as a special low level Linux Kernel realtime environment or RTKernel for DOS.

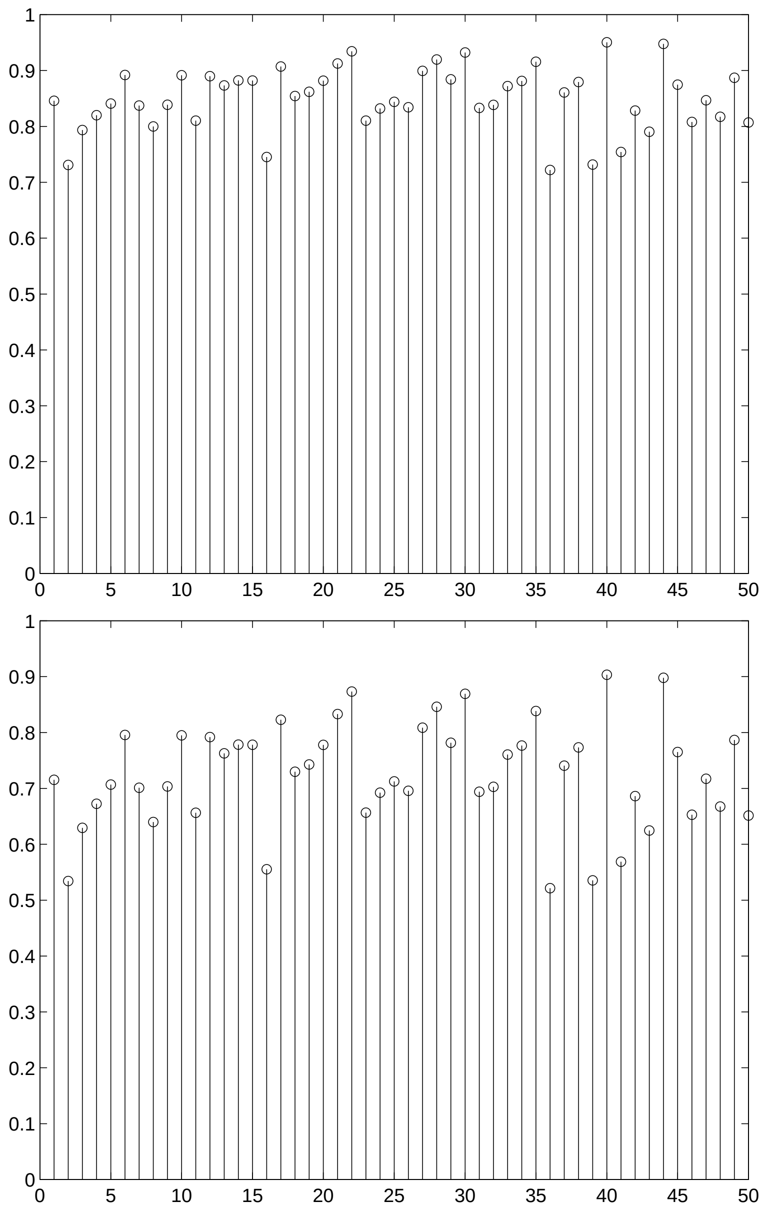


Figure 8.6: Top: first eigenvalue of the sampled data system, bottom: second eigenvalue of this system (in the sampled data domain) - these values differ for every sample step due to changes in sample time

8.3 Multitasking

In order to provide real time, multitasking is a versatile tool. Generally, we distinguish between two different types of multitasking: cooperative and preemtive multitasking.

8.3.1 Cooperative Multitasking

In cooperative multitasking, the tasks have to co-operate; this aspect is as well advantageous as disadvantageous because if one task hangs (has entered a deadlock state), the execution of the whole program will be stopped and the system will fail.

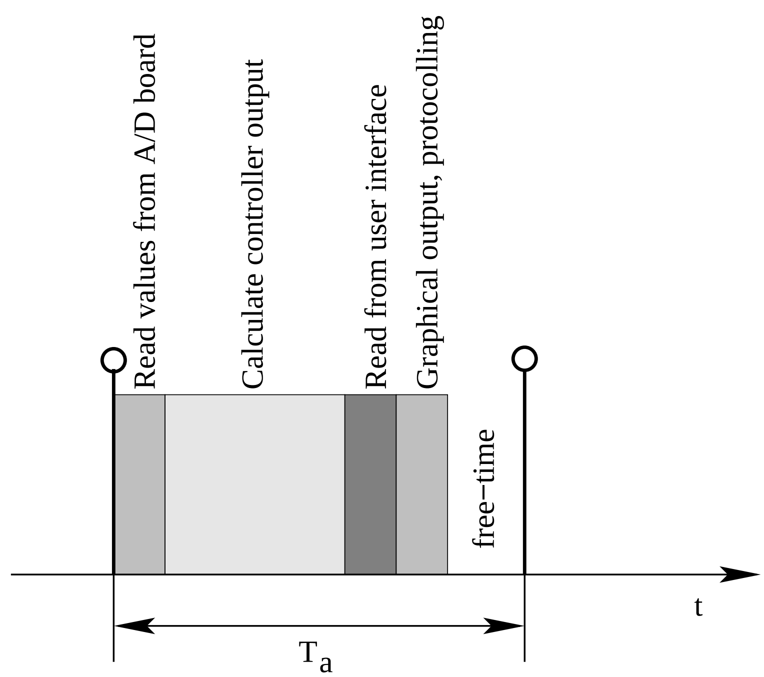


Figure 8.7: Timing and cooperative Multitasking

The advantage is, that we do not need to worry about unwanted task changes and timing can be fixed in a deterministic manner.
We can thus plan the algorithms in separately from each other and evaluate the timings for each of these in order to determine the overall duration and also the lowest possible sampling time T regarding the algorithms.

Example: We have to acquire data from an analogous to digital converter board. The board data then has to be integrated and communicated back to the board, where it is reconverted to the analog domain and fed to the plant.
This data processing has to be done in parallel to user input processing and evaluation.

For the given setting, we can easily determine timings and derive the maximum sample rate which is possible for a given amount of calculation power.

- Assume a maximum time delay of $0.2ms$ for the A/D-Board to get ready:

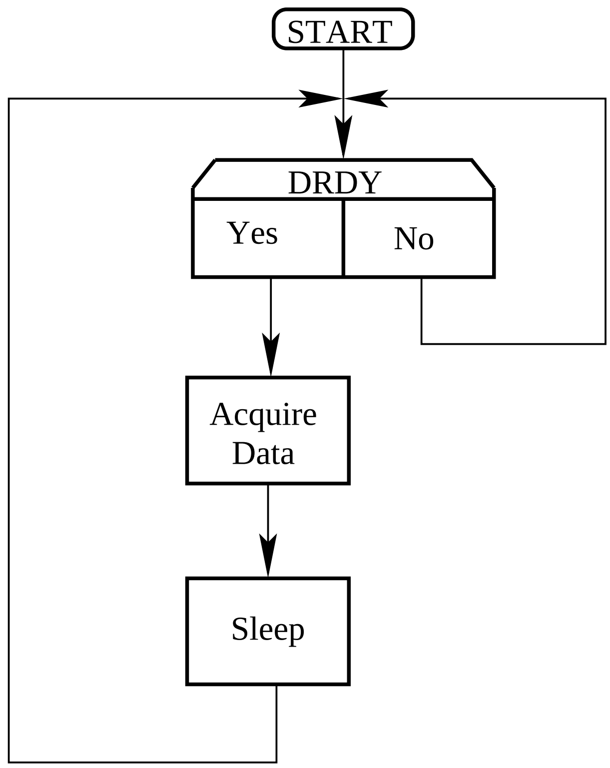


Figure 8.8: The flowcharts of the data-acquisition task (DRDY=data ready)

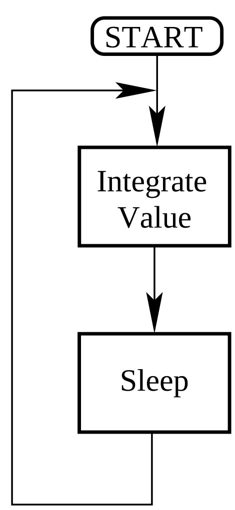


Figure 8.9: The flowcharts of the integration task

$\tau_1 = 0.2 \cdot 10^{-3}$s

- The acquisition routine takes (in our example) $\tau_2 = 0.01 \cdot 10^{-3}$s
- Let the keyboard task consume about $\tau_3 = 0.05 \cdot 10^{-3}$s

For the overall execution time, we obtain:

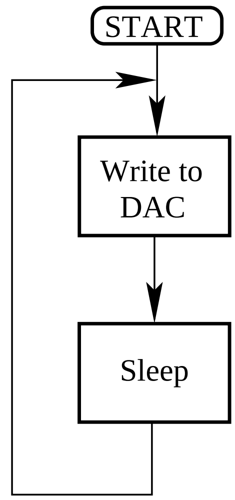


Figure 8.10: The flowcharts of the data-output task

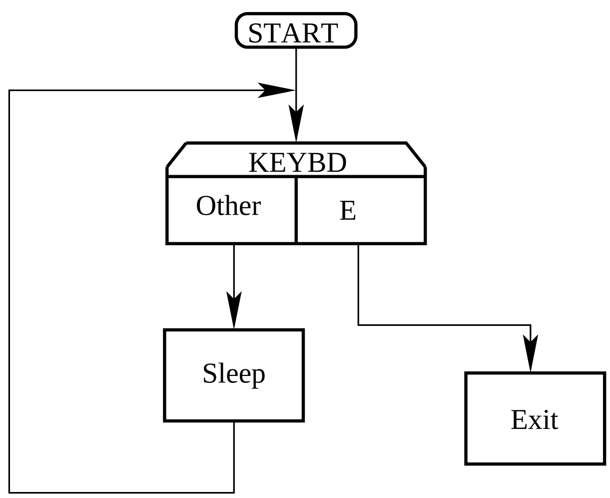


Figure 8.11: The flowcharts of the main and keyboard task (KEYBD=keyboard)

$$\tau = \sum_{i=1}^{3} \tau_i = 2.07\text{ms} \tag{8.10}$$

The minimum sample time for which we can guarantee, that the algorithm does keep all provided samples and stays in the given sampling time raster is $T = 2.1$ms.

In many cases, this approach yields good results. One problem is however, that the structure of this kind of multitasking systems is not at all flexible and beyond a certain level of complexibility, time consuming processes have to be broken into less complicated pieces.

8.3.2 Preemtive Multi tasking

A more general approach which is in fact more flexible but also more difficult concerning its handling and also creates more processor load is preemtive multi tasking.

The given example problem can also be treated using cooperative multitasking. Preemtive multitasking is a higher developped multi tasking based on a scheduler that decides whether a certain task is running or not and is also allowed to interrupt tasks at every point of time.

The differences between cooperative and preemtive multitasking can be classified as follows:

A system for preemtive multitasking contains a scheduler. The scheduler is allowed to stop execution of a task and to continue with the execution of another task without the "cooperation" of the task being stopped.
A simple preemtive multitasking scheduler usually needs the following informations for execution of the tasks:

- the priority of a task
- the status of a task (ready to run, blocked, waiting, ...
- Interruptibility

The scheduler derives from these informations whether to interrupt a task or to continue the execution of another task.
Preemtive multitasking requires in addition to the presented data a more complicated structuring of variable handling and inter-task communication. Using cooperative multi tasking, it is still possible to use e.g. global variables for data-exchange. In preemtive multitasking that cannot be afforded.

Example: Assume two tasks, both using a certain global variable.

The first task has to do some calculations:

```
TASK task1
{
  //Initialization part
  a=0;
  ...

  //Execution part
  while(1)
  {
```

```
    a=a+5;
    .
    .
    .
    a=a-2;
    .
    .
    .
    result=a;
    //Give back to scheduler
    //explicityl
    sleep();
   }
}
```

The second task uses the results of calculations of the first task:

```
TASK task2
{
  //Initialization part
  ...

  //Execution part
  while(1)
  {
    a+=1;
    .
    .
    .
    a=a-2;
    .
    .
    .
    printf(a);
    //Give back to scheduler
    //explicityl
    sleep();
   }
}
```

The global variable a which is used as a means of communication would not cause problems if used in a cooperative multitasking environment (see Fig. 8.12). Using preemtive multitasking, task 1 could be interrupted by the scheduler and the execution could be continued in task 2 (due to the overall state of the multitasking system). In this case, once reactivated, task 1 would continue its calculations using wrong values (a has changed meanwhile). There are - generally speaking - two possibilites to overcome this problem:

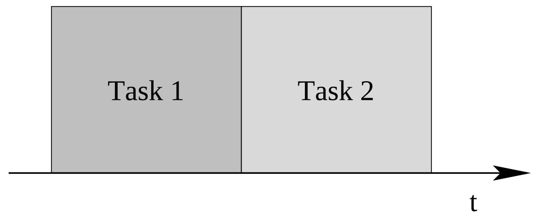


Figure 8.12: Task 1 and 2 and their timing using cooperative multitasking

- Critical sections
- Semaphores and flags

Critical sections are pieces of program code of which execution must not be interrupted by the scheduler. Critical sections are a step back into the direction of cooperative multitasking. Critical sections are usually implemented in tasks which handle protocols with hardware components and of which interruption is not allowed. For the predescribed problem, the second solution - semaphores and flags - are used. Inter-task communication can be made "multitasking"-safe using this method of blocking task changes:

```
TASK task1
{
  //Initialization part
  a=0;
  ...
  signal(twoready); //Signal yourself
  //Execution part
  while(1)
  {
    wait(twoready);
    a=a+5;
    .
    .
    .
    a=a-2;
    .
    .
    .
    result=a;

    signal(oneready);
   }
}
```

The second task again uses the results of calculations of the first task but now waits for the flag oneready:

```
TASK task2
{
  //Initialization part
  ...

  //Execution part
  while(1)
  {
    wait(oneready);
    a+=1;
    .
    .
    .
    a=a-2;
    .
    .
    .
    printf(a);
    signal(twoready);
   }
}
```

In the previous code, communication was again performed using global variables. Now we use semaphores for the same problem:

```
TASK task1
{
  //Initialization part
  int a=0;
  ...
  sematwo->value=0;
  send(sematwo);

  //Execution part
  while(1)
  {
    wait(sematwo);
    a=sematwo->value;

    a=a+5;
    .
    .
    .
    a=a-2;
    .
    .
    .
```

```
    semaone->value=a;
    send(semaone);
   }
}
```

The second task evaluates the contents of the semaphore sent by the first task and uses local variables for its calculations only.

```
TASK task2
{
  //Initialization part
  ...
  int b=0;
  ...

  //Execution part
  while(1)
  {
    wait(semaone);
    b=semaone->value;
    b+=1;
    .
    .
    .
    b=b-2;
    .
    .
    .
    sematwo->value=b;
    send(sematwo);
   }
}
```

8.4 Approach using embedded Systems or Network Connections

Real-time can be provided by both, an appropriate software and - also possible - a specific hardware. These approaches are concerning their aim and results aequivalent but different in price.
The hardware-approach is more expensive because one needs "intelligent" hardware; most often this will be an embedded system (embedded board). An embedded system board uses its own operating system which is realtime capable and also contains its own processor. In embedded systems, the realtime component (the embedded component) takes full control of all necessary realtime tasks. Visualization itself is performed by another programme which is running on a

graphical operation system on the embedding PC.

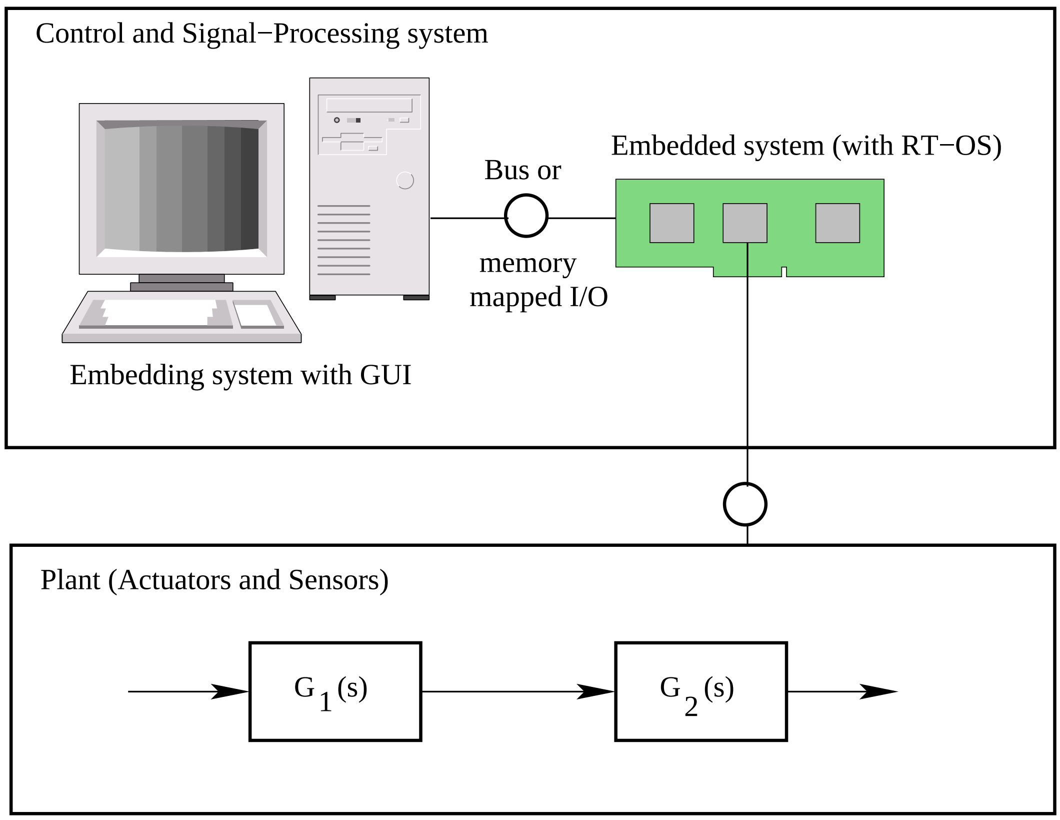


Figure 8.13: The informational structure of an embedded system

The coupling between both systems is realized using the ISA or PCI-bus. The communication with the realtime board can be done using either dual ported memory or by directly reading from the board. For visualization purposes, most of the data has to be stored on the board before it is then fetched for visualization by the embedding system.

Another possibility is to connect the realtime part of the system is using a network connection. This yields the same results as the previously discussed solution.

Chapter 9

Application Testing, Soft- and Hardware Approach

After an application or a complex software system has been realized and implemented, it is necessary to find out whether it works under the conditions specified in advance and to make it bug free. Assuming that our aim is not to sell bugs as features, we have to assure a very high precision concerning error-freeness of our system. Although the step in development which is described in this chapter is one of the last steps in software/systems-design, it is one of the most important steps.

One problem is, that the whole testing process does not yield anything besides error-freeness. Many developers do in fact not see, that a program which is working in 85% of the specified cases is *worse* than no program. Especially in process control and all other industrial processes, a 100% solution is required.

9.1 Aspects of Testing Systems

As we have assumed in the previous chapters, our aim is the development of a more or less complicated system-structure for automization purposes. It is obvious, that the resulting software cannot be tested as a text-processing program. For a successful test, we have to include all components which occur in the final system.

However, one problem is, that e.g. the testing of a signal processing environment with a signal-acquisition card cannot be done on a single computer system. For that purpose, one has to set up a whole test environment; if a sufficient hardware basis is not available, another software system which simulates the hardware has to be set up.

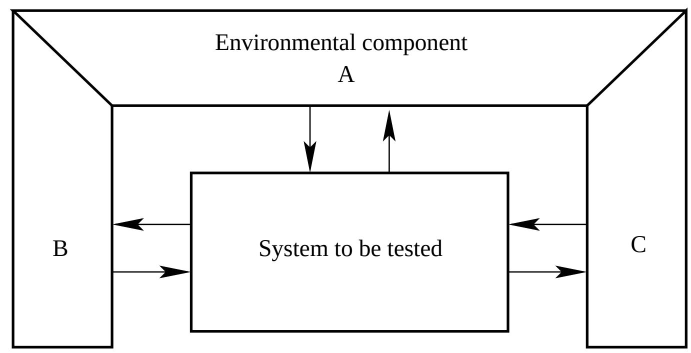


Figure 9.1: View from the perspective of the system which has to be tested

- The expected environmental structure around the system has to be set up
- The system has to be operated under real life circumstances
- Software testing alone is in most cases not sufficient

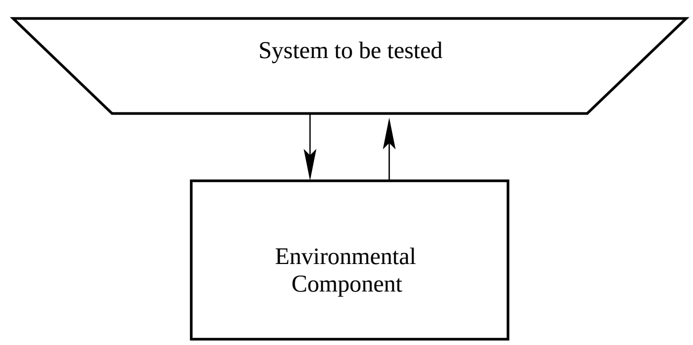


Figure 9.2: View from one of the systems which realize the environment for the system to be tested

This again makes clear, that input output specifications are of high necessity. If these specifications exist, one system can be tested to behave accordingly before it is used as a part of the environmental structure of the system which has to be tested.

9.2 Building up systems for Simulation purposes

For our investigation of systems for Simulation purposes, we will now define an example specification (plant model). Let us assume a feedforward control system as given in Figure 9.3.

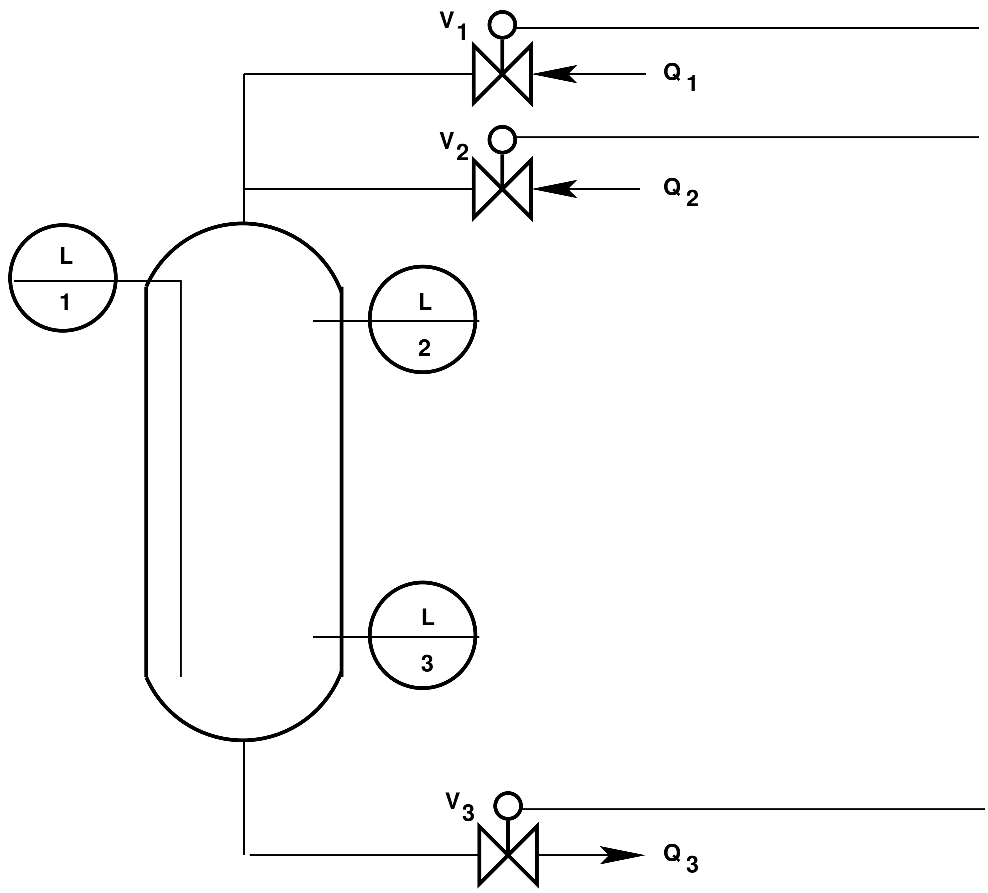


Figure 9.3: A sample processing plant

The plant consists of a tank, three level sensors L_1, L_2, L_3 of which two (L_1,L_2) are binary sensors which indicate 1 if the fluid level has reached (or is higher) than the sensor; these are the important sensor signals for our feedforward control. L_3 is analogous and fed to an indicator, which gives the actual level in %.

9.2.1 Dynamic model of the tank

The first step in our software design process will be to analyze the dynamic behaviour of the tank-system. As we know from basic control-systems and system modeling theory, a fluid tank can be modelled as an integrator with the initial value the initial volume and the input and outut flows as $Q_i = \dot{V}(t)$:

$$V = \int_0^{t_1} \dot{V} dt \tag{9.1}$$

$$= V(t_0) + \int_{t_0}^{t_1} \dot{V} dt \tag{9.2}$$

As the system is of first order and the sensors as well as the actors are binary, we may omit a nonlinear analysis of the tank although it is not purely cylindrical. For a purely cylindrical tank, level and volume are related through

$$V(t) = \pi r^2 \cdot l(t) \tag{9.3}$$

with r the radius of the tank and l the level. Thus we obtain finally:

$$l(t) = \frac{V(t_0) + \int_{t_0}^{t_1} \dot{V} dt}{\pi r^2} \tag{9.4}$$

The dynamic behaviour of this system is given by the block diagram in Figure 9.4. This we need to design a simulator of the plant which acts according to the dynamic features of the real plant.

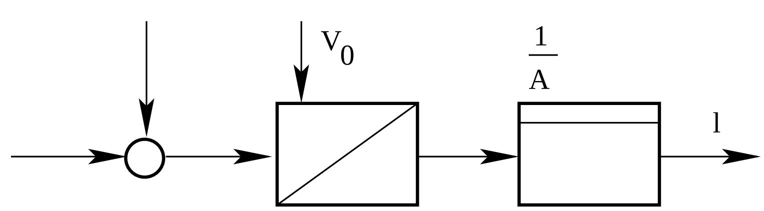


Figure 9.4: The block-diagram for dynamic simulation, $\frac{1}{A}$ is the area of the cylindric tank's base

Now the next step is to make up a design for the logic control part. The user specification is given as follows:

- The tank level should be kept around 90%. The first sensor, L_1 indicates 90% fluid level, the second sensor, L_2 indicates 35%.
- The actors are operated as binary actors - the valves V_1,V_2 and V_3 are either open or closed.
- Whenever the level in the tank reduces below 90%, the first valve has to be opened, after the level of 35% is reached, the second valve should be also opened.

- the output stream of the output valve is 1.5 times the input stream of both the input valves.

The control algorithm for the given plant will be designed using an interpreted Petri net as proposed in [7].

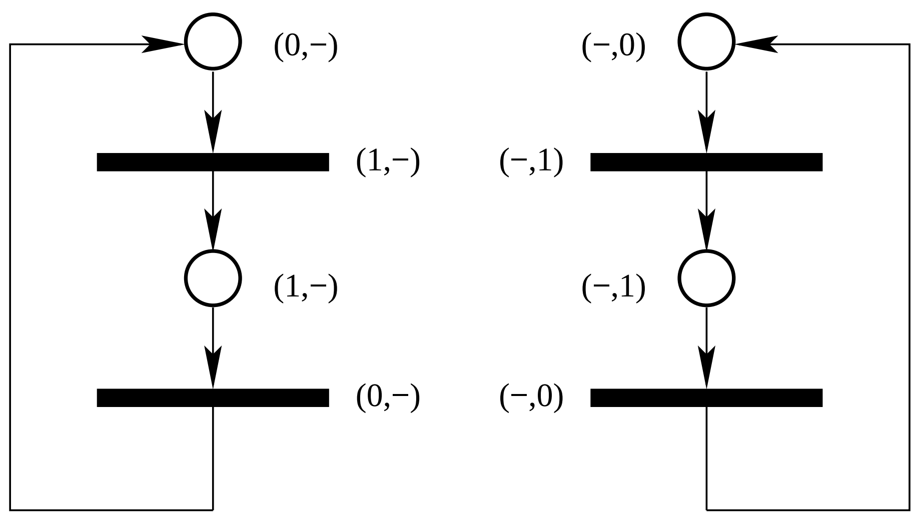


Figure 9.5: The petri net for the control algorithm

As the plant cannot be accessed (because of some special reasons, e.g. production must not be disturbed, poisonous gas, etc.), we have to set up the controller and the plant. One very important point in making a simulation model is that the terminal specification, such as which sensor signals and actor signals exist, must be known in advance. Only a good terminal specification grants errorfree control systems if they are designed in advance.

In the next section we will set up both, the feedforward controller and the plant in software. We will first leave out the hardware layer which is then added in the final section of this chapter. As for our approach we only need a very rough dynamical model, we do not need a good realtime basis; that is why we do not necessarily need a special interface and thus can use ordinary system timers.

9.3 Single system solution

9.3.1 Simulation of the plant

Our first task is to set up a simulation entity for our tank-valve-system. For the tanks's dynamics we already set up a rough model. As the plant dynamics are very slow and we do not consider a closed loop system, real-time simulation is not required (although a stable timing is needed).

In order to simulate the input signal states and to visualize the output signals of the plant's sensors and actors, we introduce graphical buttons. The visualization of the plant is obtained by opening an image file and visualizing it in the background of the simulation window.

The integration of the amount of fluid in the tank is performed using Euler-integration:

$$y(t) = \int_0^t x(t)dt \tag{9.5}$$

which can be expressed in the discrete time domain as a sum:

$$\begin{aligned} y(k) &= T_a \cdot \sum_{i=1}^{k} x(k) \\ y(k+1) &= T_a \cdot \sum_{i=1}^{k+1} x(k) \\ &= T_a \cdot \sum_{i=1}^{k} x(k) + x(k+1) \end{aligned}$$

Thus the integration is performed using the following equation:

$$y(k+1) = y(k) + x(k+1) \tag{9.6}$$

This is in the $\mathcal{Z}$-domain:

$$\begin{aligned} y(k+1) - y(k) &= x(k+1) \\ (z-1) \cdot Y(z) &= z \cdot X(z) \\ Y(z) &= \frac{z}{z-1} X(z) \end{aligned} \tag{9.7}$$

The above integration algorithm is easily implementable using a timer-routine which is called every sample step T_a:

```
void Flowactor::Integrate()
{
  //Integrate Flows
  m_Volume+=m_Ta*(m_Q1+m_Q2+m_Q3)*0.05;

  if (m_Volume>120) m_Volume=120; // Do not overflow
  if (m_Volume<0) m_Volume=0;     // (closed Tank)
```

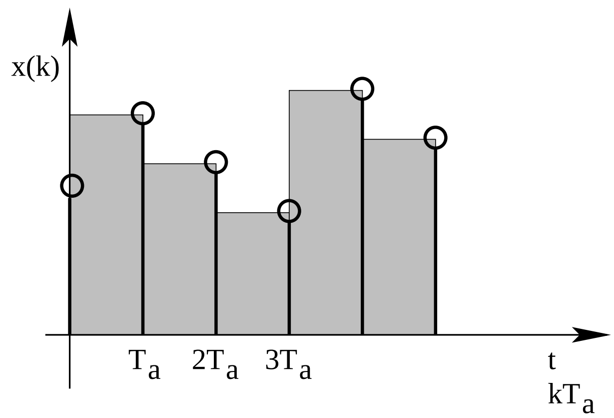


Figure 9.6: Euler-Integration using a sampled signal

```
  if (m_oldvolume>m_Volume) QProgressBar_1->reset();
  m_oldvolume=m_Volume;
  QProgressBar_1->setProgress(m_Volume*10);

  //Check Fluid-Level and setze/reset
  //L1 and L2 accordingly

  if (m_Volume>35) QPushButtonA2->setOn(TRUE);  //35 % of maximum Level
  else QPushButtonA2->setOn(FALSE);
  if (m_Volume>90) QPushButtonA1->setOn(TRUE);  //90 % of maximum Level
  else QPushButtonA1->setOn(FALSE);

}
```

The input flows to the plant are determined by the buttons pressed by the user (for test purpose). Therefor the following slots have been implemented:

```
/** Button E1/A1 pressed */
void Flowactor::Ventil1Offen()
{
  if (QPushButtonE1->isOn())
  {
                m_Q1=75;
        }
        else
        {
                m_Q1=0;
        }
}

/** Button E2/A2 pressed */
```

```
void Flowactor::Ventil2Offen()
{
  if (QPushButtonE2->isOn())
  {
                m_Q2=75;
        }
        else
        {
                m_Q2=0;
        }
}

/** Button E3/A3 pressed */
void Flowactor::Ventil3Offen()
{
  if (QPushButtonE3->isOn())
  {
                m_Q3=-100;
        }
        else
        {
                m_Q3=0;
        }
}
```

The initialization of the timer and communication of the buttons is performed in the constructor of the flowactor-entity:

```
Flowactor::Flowactor(QWidget *parent, const char *name)
           : QWidget(parent, name)
{
        initDialog();
        //Load image and show
        m_Reaktor.load("actor.pnm");
        m_Reaktordraw.convertFromImage(m_Reaktor);
        m_oldvolume=0;

        //Set initial volume to 50%
        m_Volume=50;
        m_Ta=0.1;
        m_Q1=0;
        m_Q2=0;
        m_Q3=0;

        //Slots
        connect(QPushButtonE1, SIGNAL(clicked()),
        this, SLOT(Ventil1Offen()));
```

```
        connect(QPushButtonE2, SIGNAL(clicked()),
        this, SLOT(Ventil2Offen()));
        connect(QPushButtonE3, SIGNAL(clicked()),
        this, SLOT(Ventil3Offen()));

        //Timer init
        m_Timer = new QTimer( this );
        connect( m_Timer, SIGNAL(timeout()),
                 this, SLOT(Integrate()) );
        m_Timer->start( 100, FALSE );
        // 0.1 seconds=Ta=100msec
}
```

The visualization of the level in the tank is obtained using a progress bar and for visualization of the signals, we introduced the buttons:

```
void Flowactor::initDialog(){
  this->resize(680,620);
  this->setMinimumSize(0,0);
  QProgressBar_1= new QProgressBar(this,"NoName");
  QProgressBar_1->setGeometry(40,560,510,20);
  QProgressBar_1->setMinimumSize(0,0);
  QProgressBar_1->setTotalSteps(1000);

  QLabel_1= new QLabel(this,"NoName");
  QLabel_1->setGeometry(40,580,30,30);
  QLabel_1->setMinimumSize(0,0);
  QLabel_1->setText(("0%"));

  QLabel_2= new QLabel(this,"NoName");
  QLabel_2->setGeometry(520,580,40,30);
  QLabel_2->setMinimumSize(0,0);
  QLabel_2->setText(("100%"));

  QPushButtonE1= new QPushButton(this,"NoName");
  QPushButtonE1->setGeometry(570,50,70,30);
  QPushButtonE1->setMinimumSize(0,0);
  QPushButtonE1->setText(("E1/A1"));
  QPushButtonE1->setToggleButton(true);

  QPushButtonE2= new QPushButton(this,"NoName");
  QPushButtonE2->setGeometry(570,110,70,30);
  QPushButtonE2->setMinimumSize(0,0);
  QPushButtonE2->setText(("E2/A2"));
  QPushButtonE2->setToggleButton(true);

  QPushButtonE3= new QPushButton(this,"NoName");
```

```
  QPushButtonE3->setGeometry(570,480,70,30);
  QPushButtonE3->setMinimumSize(0,0);
  QPushButtonE3->setText(("E3/A3"));
  QPushButtonE3->setToggleButton(true);

  QLabel_3= new QLabel(this,"NoName");
  QLabel_3->setGeometry(40,10,120,30);
  QLabel_3->setMinimumSize(0,0);
  QLabel_3->setText(("Füllstandsstrecke"));

  QPushButtonA1= new QPushButton(this,"NoName");
  QPushButtonA1->setGeometry(300,210,70,30);
  QPushButtonA1->setMinimumSize(0,0);
  QPushButtonA1->setText(("A1/E1"));
  QPushButtonA1->setToggleButton(true);
  QPushButtonA1->setEnabled( FALSE );

  QPushButtonA2= new QPushButton(this,"NoName");
  QPushButtonA2->setGeometry(300,385,70,30);
  QPushButtonA2->setMinimumSize(0,0);
  QPushButtonA2->setText(("A2/E2"));
  QPushButtonA2->setToggleButton(true);
  QPushButtonA2->setEnabled( FALSE );

  QLabel_4= new QLabel(this,"NoName");
  QLabel_4->setGeometry(530,10,130,30);
  QLabel_4->setMinimumSize(0,0);
  QLabel_4->setText(("Sensor/Actuator-Signals"));
}
```

The Entity-Block-Diagram of the simulation entity is given in Fig. 9.7
The following class definition depicts the organization of the Entity given in Fig. 9.7:

```
class Flowactor : public QWidget  {
   Q_OBJECT
public:
        Flowactor(QWidget *parent=0, const char *name=0);
        ~Flowactor();

protected:
        void initDialog();

        QProgressBar *QProgressBar_1;
        QLabel *QLabel_1;
        QLabel *QLabel_2;
        QPushButton *QPushButtonE1;
```

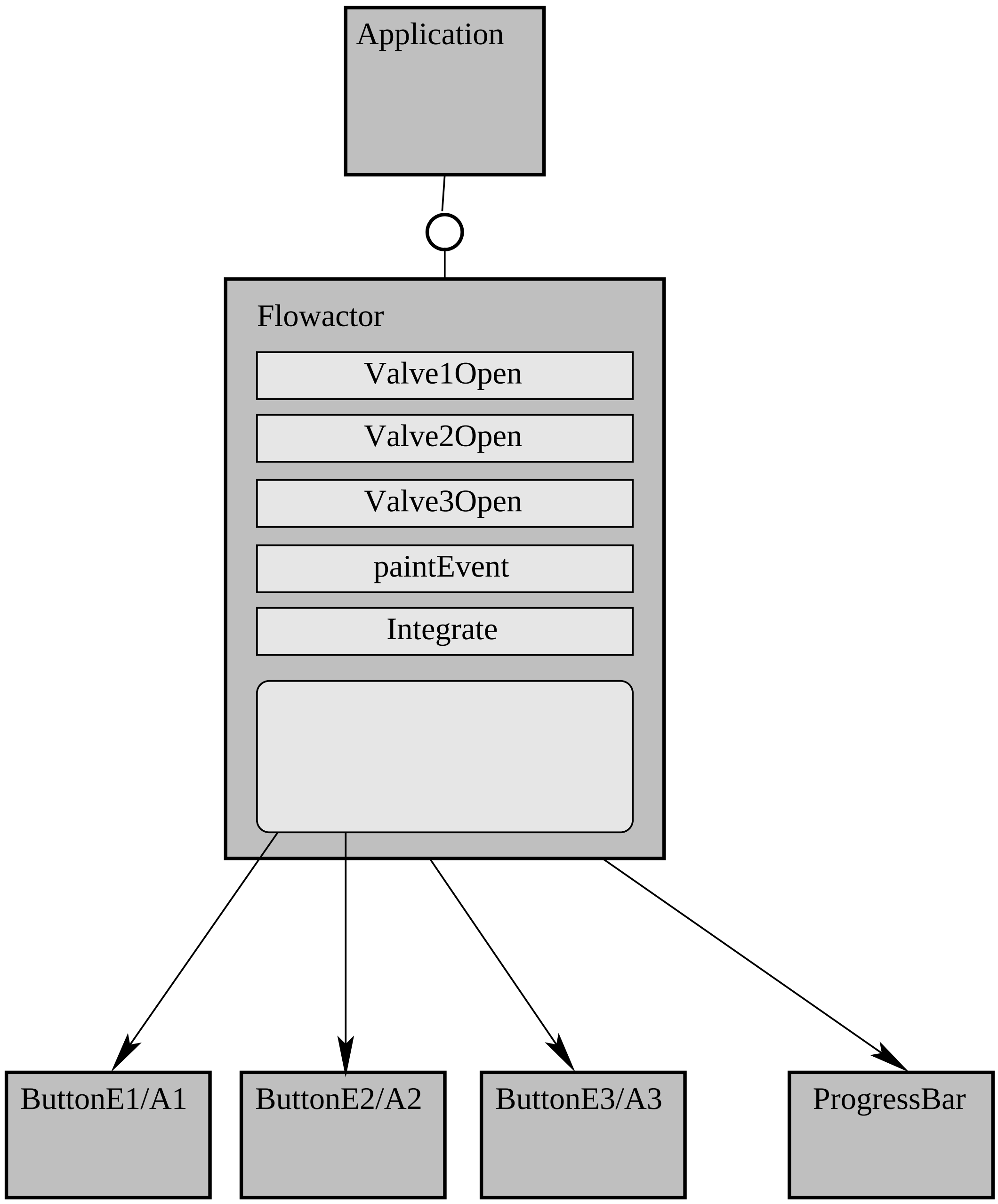


Figure 9.7: The Entity-Block-Diagram of the Actor-Simulation Entity

```
QPushButton *QPushButtonE2;
QPushButton *QPushButtonE3;
QLabel *QLabel_3;
QPushButton *QPushButtonA1;
QPushButton *QPushButtonA2;
QPushButton *QPushButtonA3;
QLabel *QLabel_4;
```

```
        //Darstellung des Reaktors
        QImage m_Reaktor;
        QPixmap m_Reaktordraw;
private:

public:

protected:
        void paintEvent(QPaintEvent* paintevent);
public slots: // Public slots
  /**  */
  void Ventil3Offen();
public slots: // Public slots
  /**  */
  void Ventil2Offen();
public slots: // Public slots
  /**  */
  void Ventil1Offen();
  void Integrate();

public:
  void ComError();

public:
  float m_Volume;
  float m_oldvolume;
  float m_Q1;
  float m_Q2;
  float m_Q3;
  float m_Ta;
  QTimer* m_Timer;
};
```

9.3.1.1 Timer and Plant Dynamics

The logic control is built up in a similar manner as the plant simulator. First of all, it consists mainly of the underlying petrinet which realizes the logic control algorithm.

The petrinet can be decomposed int its components:

- Arcs
- Transitions
- Places

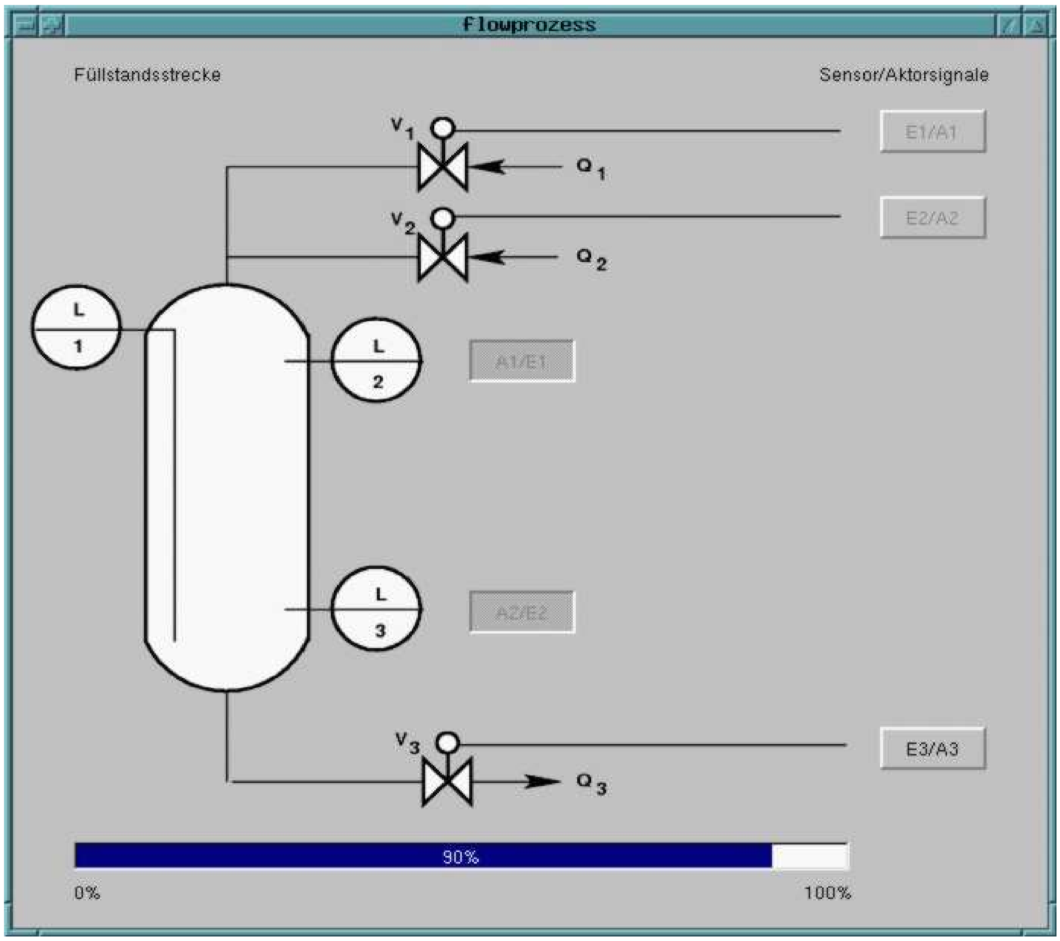


Figure 9.8: A sample processing plant

According to the principles of object orientation, this is possible - here we select an alternative but still object-oriented approach:
The entity Petrinet sets up all transitions and places. Their only functionality is to visualize themselves (in case of places: their state, etc.) The petrinet-entity guarantees the concession-rule and switches transitions.
In a general case, we would use an incidence-matrix which is built up from the information about which transitions are ready to switch and to determine the following state which is given by the set of all places and the token-positions (see Appendix A). The entity-diagram given in Fig. 9.9 The following class description gives the coded version of Fig. 9.9:

```
class Petrinetz : public QObject
{
public:
  Petrinetz();
  ~Petrinetz();

  void SetPosition(int x,  int y);
```

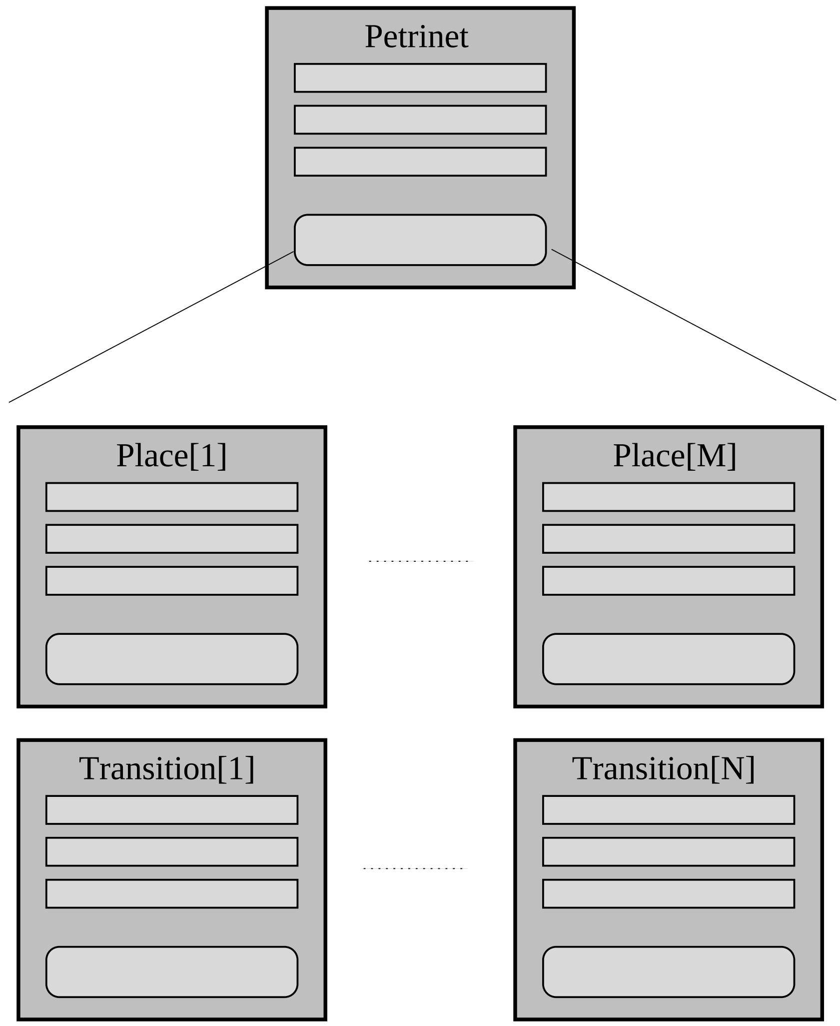


Figure 9.9: The Entity diagram of the petrinet entity

```
  void Paintme(QPainter* painter);
  void Grundstellung();

public:
 bool Switchthrough();
public:
 Place      m_Place[6];
 Transition m_Trans[4];
 bool       m_e1;
 bool       m_e2;
 bool       m_a1;
 bool       m_a2;

public:
 int m_PosX;
```

```
 int m_PosY;
};
```

The following code shows the implementation of the switching conditions for the petrinet:

```
bool Petrinetz::Switchthrough()
{
  bool erg=FALSE;

  //Transition 0 **********************************************
  if ((m_Place[0].m_active==TRUE)&&(m_Place[2].m_active==FALSE))
  // Transition 0 ready
  {
        if (m_e1==TRUE) //Check switching condition
        {
          m_Place[0].m_active=0; //deactivate places
          m_Place[2].m_active=1;
          erg=TRUE;
        }
  }
  //Transition 0 End *****************************************

  //Transition 1 **********************************************
  if ((m_Place[1].m_active==TRUE)&&(m_Place[3].m_active==FALSE))
  // Transition 1 schaltbereit
  {
        if (m_e2==TRUE)
        {
          m_Place[1].m_active=0;
          m_Place[3].m_active=1;
          erg=TRUE;
        }
  }
  //Transition 1 Ende ****************************************

  //Transition 2 **********************************************
  if ((m_Place[3].m_active==TRUE)&&(m_Place[1].m_active==FALSE))
  // Transition 2 schaltbereit
  {
        if (m_e2==FALSE) //Transition schalten
        {
            m_Place[3].m_active=0;
            m_Place[1].m_active=1;
            erg=TRUE;
        }
  }
```

```
  //Transition 2 Ende ****************************************

  //Transition 3 *********************************************
  if ((m_Place[2].m_active==TRUE)&&(m_Place[0].m_active==FALSE))
  // Transition 3 schaltbereit
  {
        if (m_e1==FALSE) //Transition schalten
        {
           m_Place[2].m_active=0;
           m_Place[0].m_active=1;
           erg=TRUE;
        }
  }
  //Transition 3 Ende ****************************************

  //Setze ausgänge passend
  if (m_Place[0].m_active==1) m_a1=0;
  if (m_Place[2].m_active==1) m_a1=1;
  if (m_Place[1].m_active==1) m_a2=0;
  if (m_Place[3].m_active==1) m_a2=1;
  return erg;
 }

//Initial setting (safe mode)
void  Petrinetz::Grundstellung()
{
  m_Place[0].m_active=1;
  m_Place[1].m_active=1;
  m_Place[2].m_active=0;
  m_Place[3].m_active=0;

  //Setze the place related output
  //signals
  if (m_Place[0].m_active==1) m_a1=0;
  if (m_Place[2].m_active==1) m_a1=1;
  if (m_Place[1].m_active==1) m_a2=0;
  if (m_Place[3].m_active==1) m_a2=1;
}
```

The activity of the Petri net is again triggered using a timer in the flowcontrol-entity that calls the Switchthrough()-function.

```
void Flowcontrol::timerDone()
{
        bool erg;
        erg= m_sipn.Switchthrough();
        if (erg)
```

```
        {
                if (m_sipn.m_a1==TRUE)
                {
                        QPushButtonA1->setOn(TRUE);
                }
                else
                {
                        QPushButtonA1->setOn(FALSE);
                }
                if (m_sipn.m_a2==TRUE)
                {
                  QPushButtonA2->setOn(TRUE);
                }
                else
                {
                        QPushButtonA2->setOn(FALSE);
                }
                update();
        }

        //User Input (drain valve)

                if (m_sipn.m_e3==TRUE)
                {
                  QPushButtonE3->setOn(TRUE);
                }
                else
                {
                        QPushButtonE3->setOn(FALSE);
                }
}
```

The signals are again set or rest using buttons and the Petri net is provided with the necessary information using the following slots:

```
  //Sensor signals :
  connect( QPushButtonE1, SIGNAL(clicked()),this, SLOT(e1kommt()) );
  connect( QPushButtonE2, SIGNAL(clicked()),this, SLOT(e2kommt()) );
```

The states of the signals (symbolized by the buttons) is then used to trigger the petrinet as descibed above:

```
/** Sensor e1  */
void Flowcontrol::e1kommt()
{
 if (QPushButtonE1->isOn())
 {
                m_sipn.m_e1=1;
```

```
 }
 else
 {
                m_sipn.m_e1=0;
 }
}

/** Sensor e2 */
void Flowcontrol::e2kommt()
{
 if (QPushButtonE2->isOn())
 {
                m_sipn.m_e2=1;
 }
 else
 {
                m_sipn.m_e2=0;
 }
}
```

9.3.1.2 Visualization layer

The visualization of the petrinet and its elements is again hierachically ordered. The petrinet entity makes the entities comprised in the petrinet-entity draw themselves:

```
void Petrinetz::Paintme(QPainter* Painter)
{
 int i;
 for(i=0;i<4;i++) m_Place[i].Paintme(Painter);
 for(i=0;i<4;i++) m_Trans[i].Paintme(Painter);
 ...
}
```

9.3.2 Connection of Controller and Plant

Now the test of the plant and the logic control working together has to be performed - now we have to connect both applications. We have two possibilities:

- Connection through an entity which takes over the data-transfer and realizes a coupling
- Connection using e.g. serial port hardware and a cable.

The object oriented approach is a versatile tool. The entity which connects both applications and sets the signal-variables according to the outputs respectively

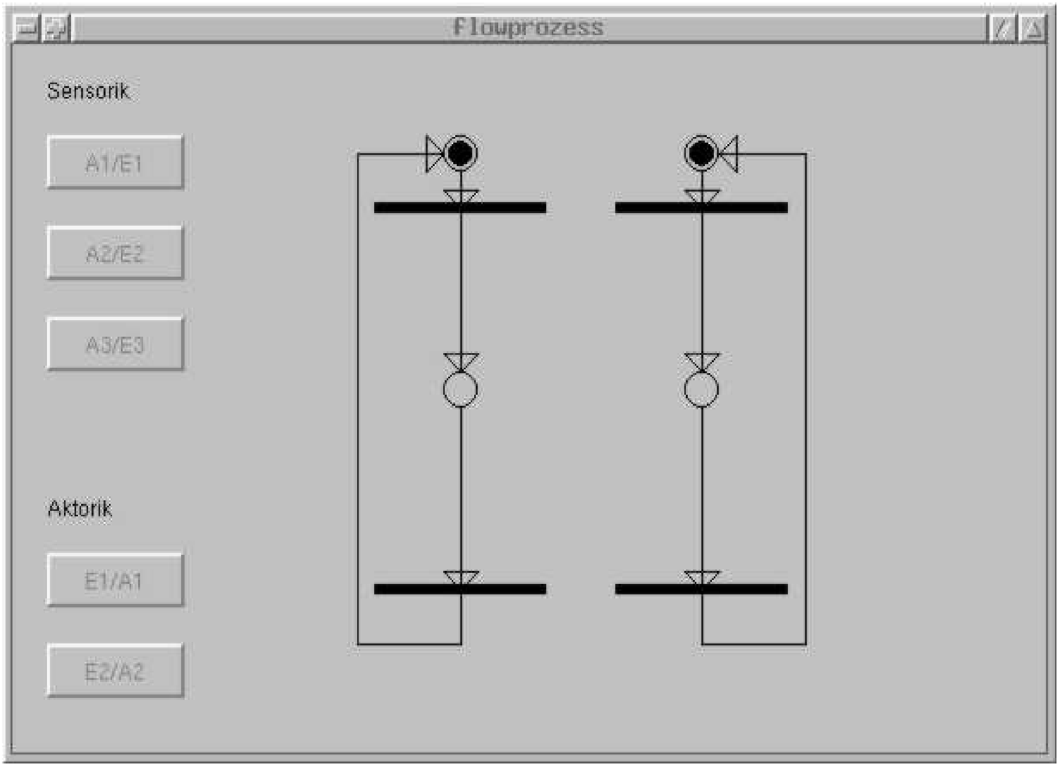


Figure 9.10: The visualized petrinet and the logic control for the tank process

inputs of either of the entities will be replaced later by a communication entity which sets these variables according to read or sent data.

Besides the flowactor we set up the flowcontrol entity. The buttons for the user have to be disables except for the button A3/E3 which is left to the user to simulate draining of the tank.
All signals are generated by either the plant's sensors or are outputs of the Petri net in the logic control application.

The values have to be updated regularly. In this case, we do not want to change neither the function-code of the logic control entity nor those of the actor entity. Thus we use a timer in the connecting entity which updates the signals' values.

These tasks are taken over by the meta-system for both entities: the FlowProcess entity.

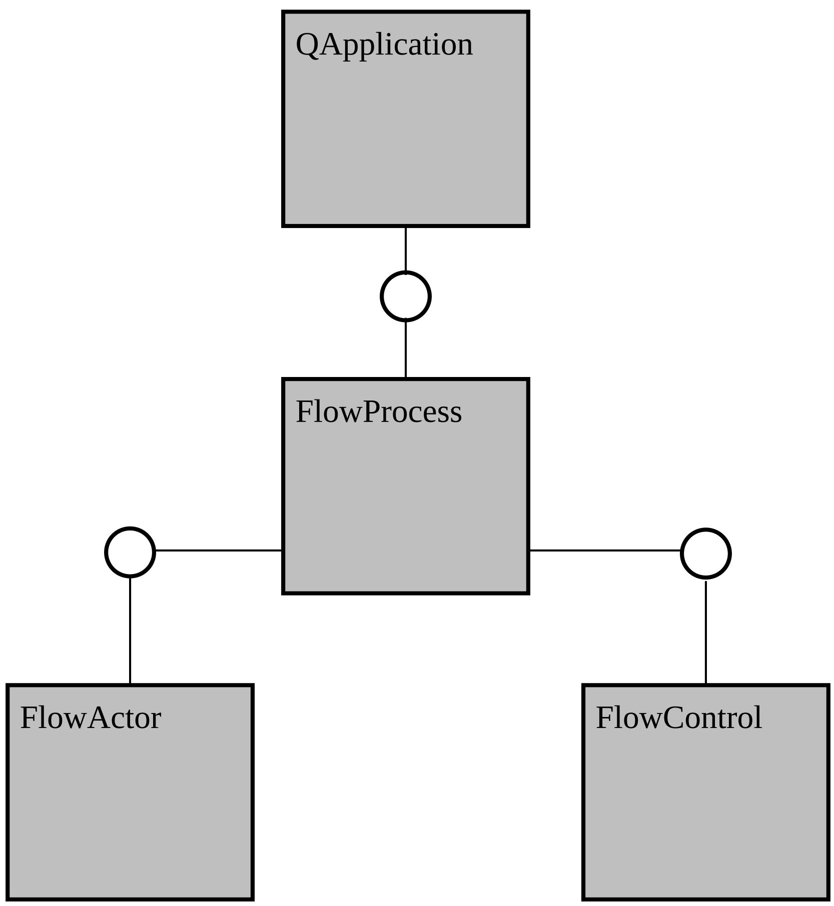


Figure 9.11: Connection using object oriented principles

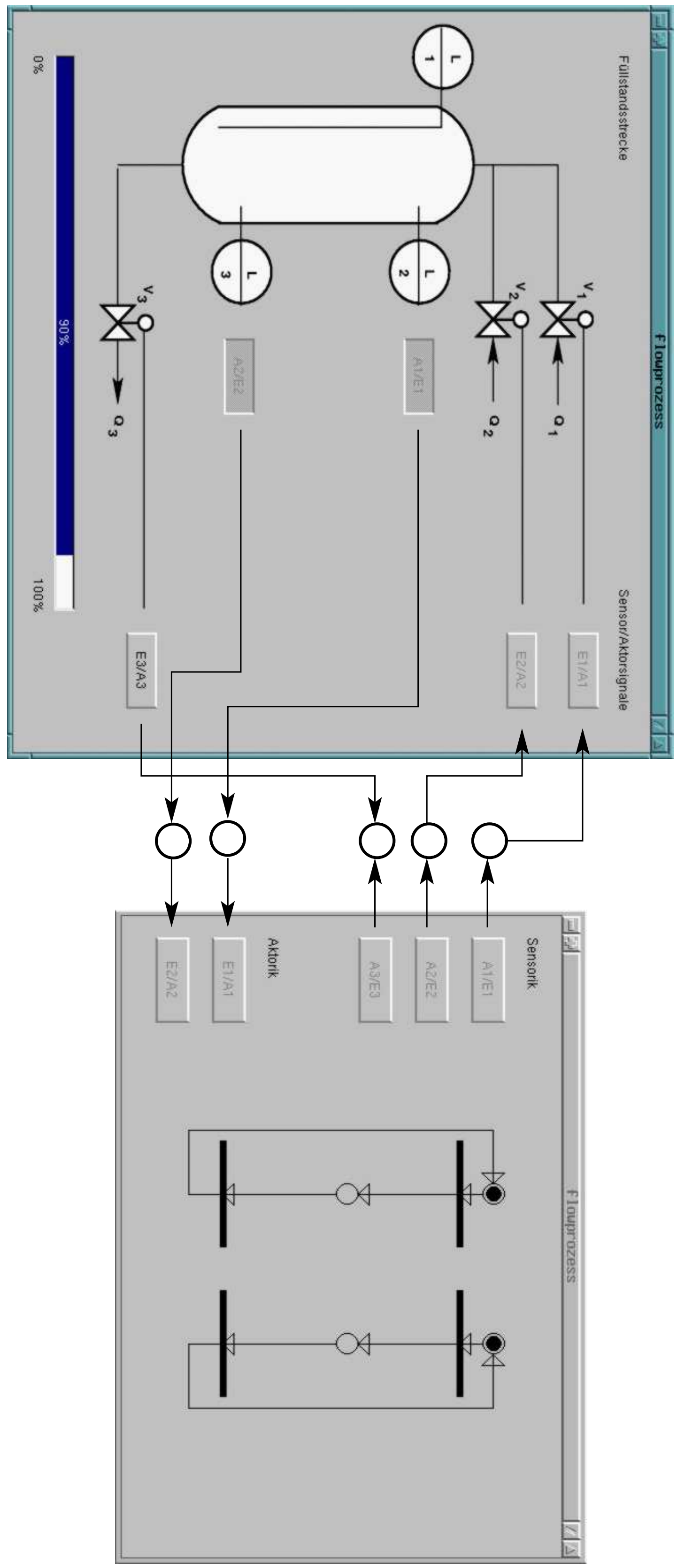


Figure 9.12: Basic connection of the Plant and the logic control

9.4 Solution using two systems

If two systems are used for hardware testing, the setup can be made more realistic. One machine simulates the process, one machine simulates the controller (logical or feedback). As the terminal descriptions of plants can be taken from their user manuals, the developer need not operate on the real plant but can instead develop and test the control algorithm on the simulated plant.

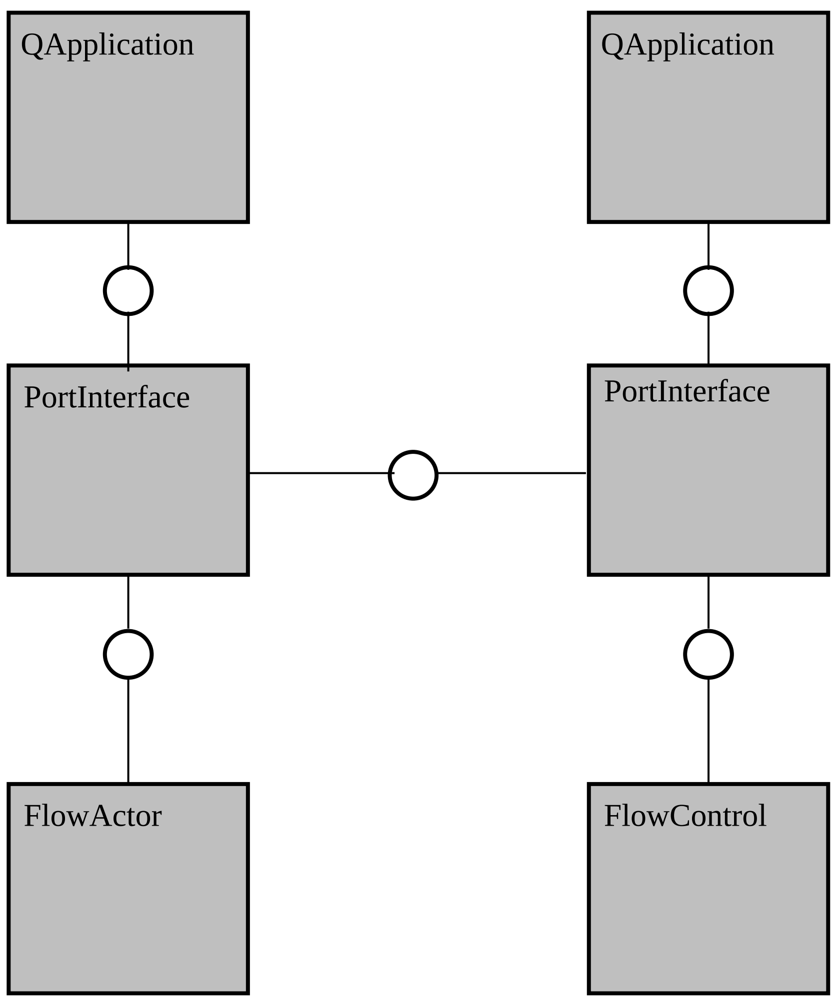


Figure 9.13: Connection of the plant and the logic control using communication entities

The two systems solution is of course more close to the real setup. Using our object oriented approach, we just have to replace the CFlowProcess entity and replace it by an appropriate entity that provides the communication features and feeds data to the entities for on one hand process control and on the other machine to the process simulation entity.

Figure 9.13 shows the entity block diagram of the systems (both simulation of the plant and logic controller) which are now interconnected using the serial port. The hardware interface now takes over the duties of the entity FlowProcess in Fig. 9.11.

The following code realizes the hardware interface:

```
class Flowcontrol_serialcom
{
  public:
    /** construtor */
     Flowcontrol_serialcom();
    /** destructor */
     ~Flowcontrol_serialcom();

  public: //Parameters for communication

   Flowcontrol* m_control;
   bool m_a1;   //Signals to the actor
   bool m_a2;   //(from controller)

   bool m_e1;   //Signals to controller
   bool m_e2;
   bool m_e3;

   int m_serial; //File-Descriptor serial Port

   //Data Structures for port handling
   struct termios m_oldtio,m_newtio;

   //Member functions for port handling
   void OpenSerial();
   void CloseSerial();

   QTimer* m_Timer; //Timer

 public slots:
  void Communicate(); //Function for communication
};
```

A similar entity is set up for the plant. The implementation for the serial communication is given as follows:

```
Flowcontrol_serialcom::Flowcontrol_serialcom()
{
  .
  .
```

```
  .
  //Open serial port
  OpenSerial();
  .
  .
  //Set timer
  m_Timer = new QTimer( this );
        connect( m_Timer, SIGNAL(timeout()), this, SLOT(Communicate()) );
  m_Timer->start( 50, FALSE );                 // 0.1 seconds
}

Flowcontrolmitseriell::~Flowcontrolmitseriell()
{
 //Close the serial port
 CloseSerial();
 .
 .
 .
 //Remove timer
 if (m_Timer) delete m_Timer;
}

void Flowcontrolmitseriell::Communicate()
{
  char buffer[255];
  m_Timer->stop();

  m_a1=m_control->m_sipn.m_a1;
  m_a2=m_control->m_sipn.m_a2;

 //Write outputs to port
  buffer[0]=(char)(m_a1);
  buffer[1]=(char)(m_a2);
  write(m_serial,buffer,2);

 //Read inputs from port
  read(m_serial,buffer,3);
  m_e1=(bool)(buffer[0]);
  m_e2=(bool)(buffer[1]);
  m_e3=(bool)(buffer[2]);

  m_control->m_sipn.m_e1=m_e1;
  m_control->m_sipn.m_e2=m_e2;
  m_control->m_sipn.m_e3=m_e3;

  m_Timer->start( 50, FALSE );
}
```

```
void Flowcontrolmitseriell::OpenSerial()
{
 if(-1==(m_serial=::open("/dev/ttyS0", O_RDWR | O_NOCTTY)))
 {
   warning("/dev/ttyS0 could not be opened");
 }
 tcgetattr(m_serial,&m_oldtio); //Alte Einstellungen holen

 bzero(&m_newtio,sizeof(m_newtio)); //Neue Einstellungen machen
 m_newtio.c_cflag = B38400 | CS8 | CLOCAL | CREAD ;
 m_newtio.c_iflag = IGNPAR | ICRNL;
 m_newtio.c_oflag = IGNPAR | ICRNL;
 m_newtio.c_lflag = 0;
 m_newtio.c_cc[VTIME]=0;
 m_newtio.c_cc[VMIN]=2;

 //Clean line
 tcflush(m_serial,TCIFLUSH); //Input Buffer
 tcflush(m_serial,TCOFLUSH); //Output Buffer
 tcsetattr(m_serial,TCSANOW,&m_newtio);
}

void Flowcontrolmitseriell::CloseSerial()
{
  if (m_serial!=-1)
  {
    tcsetattr(m_serial,TCSANOW,&m_oldtio);
    close(m_serial);
  }
}
```

If a successful operation of the controller on the plant can be guaranteed, the controller can be tried on the real plant as depicted in Fig. 9.14.

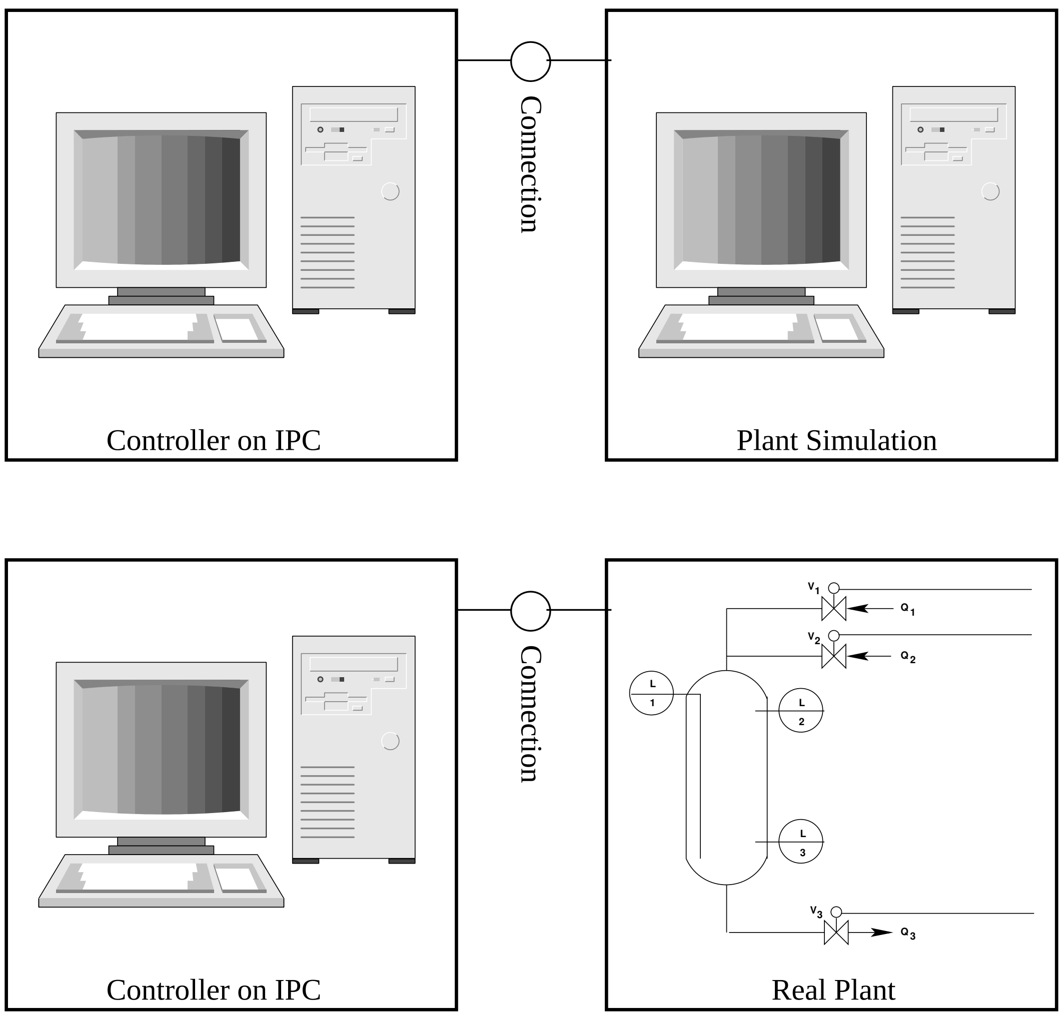


Figure 9.14: A setup with data communication and simulation or real plant coupling

Chapter 10

User Requirements and Design Document

In this chapter, we deal with a topic which is neither scientific nor directly related to the paradigms of software-development. Producing a good design document and transferring the user requirements into a suitable solution is a very important task. For the design engineer, the design document is important as far as she/he derives her/his software-design and the realization effort for a certain project from a design document.

Leaving this simple step out can lead to terrible misunderstandings and will also give room for speculations about the functionality of the delivered software.

However, formulating a design document from the User Requirements is an important but also time consuming task.

Unfortunately, the user requirements and the design document are considered beeing very similar and thus setting up the user requirements document is sometimes left out. This often encountered mistake in practice is one of the frequently occuring reasons for complaints: if a design document has been elaborated, then the customer (the one who uses the software package later) can decide whether the planned application suits her/his needs.

The process is thus itself algorithmic and iterative and is given in a rough form in Figure 10.1

10.1 Acquiring Information and User Requirements

In fact, the different types of applications can be subdivided in such ones where terminal specifications and process specifications are given and others where e.g.

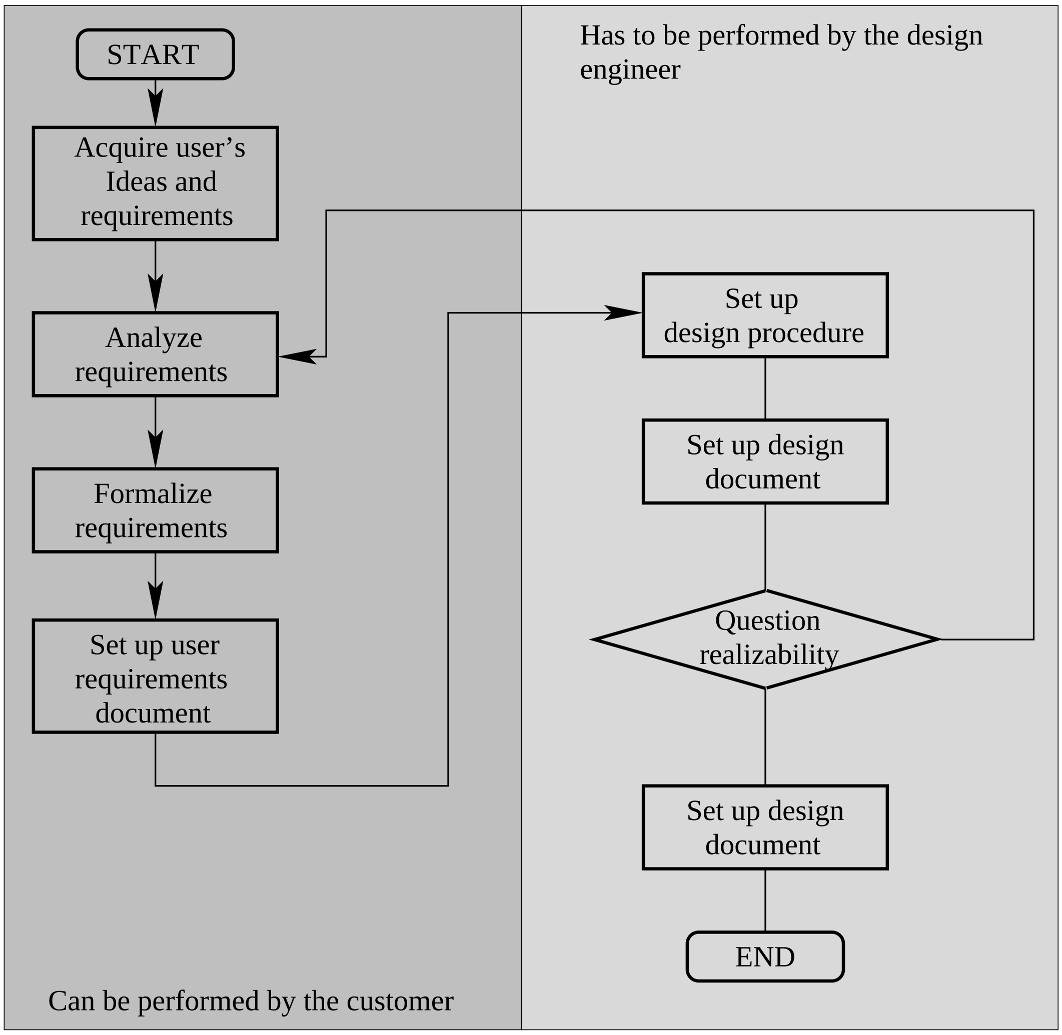


Figure 10.1: Forming the design document

a user interface has to be designed.

In either case, the specifications have to be studied exactly. Concerning the so called expert interview, it is more important to have a good imagination of the customers needs. It is very important to quickly find out the neuralgic points in the concepts and requirements of the user and to give impulses in such a way, that necessary workarounds are not postponed till the design process of an application is already over.

Many neuralgic points can only be decovered if the ideas of both, customer and interviewer are discussed.

Further, the interviewer has to be an expert, as deeply familiar with e.g. nonlinear control systems, problems in sampled data systems or even the disadvantages of different bus systems as he/she has to be an expert in software development.

If one of these points of view are omitted, a very bad mistake can happen: promises which cannot be kept because of technical reasons are made.

The next step is to make a more precise model of what to do from the inquired data: the design document.

10.2 Setting up the design document

The design document is set up based on the facts, needs, terminal specifications of the customer. Now it is the duty of the design engineer to find solutions of the users requests.

This process is not so straight and many circumstances influence the whole design process; for example one could have to choose a special hardware to fulfill a certain user request which is not available at the moment, for a realtime system, a special realtime kernel has to be chosen etc.

If changes concerning the required functionalities have to be made, it is time to talk to the customer again and to make clear why something has to be realized in a different manner.

Appendix A

Petri Nets

Petri nets are well suited to visualize, analyze or synthesize process control procedures. Petri nets consist of the following elements:

Symbol	Name	Meaning in Procss control
○	Place	Situation (Time consuming)
▬	Transition	No Time-Consumption
→	directional arc	Creation of structure
●	Token	activity information

In the mathematical sense, a Petri net is a bipartite Digraph consisting of places and transitions as nodes.

A digraph ist a graph with directed arcs, in case of the Petri net, transitions and places are joined with arcs in a mutually exclusive way: a transition can only be connected to a place and a place can only be connected to a transition. Transitions can never be connected to transitions and places cannot be connected to places.

A.1 Dynamical View of a Petri net

Places hold tokens which are flowing through the net. The transitions take tokens from their pre-arcs and exhaust tokens into their post-arcs.
The state vector $\mathbf{S}$ is given by the vector made up of all tokens in the places where the places are related to whole numbers $\mathbb{N}_0$.

$$\mathbf{S} \rightarrow \mathbb{N}_0 \tag{A.1}$$

As we do not consider Petri nets with higher numbers of tokens than one in one place and also without higher arc-flowrates, we obtain for $\mathbf{S} = \begin{pmatrix} s_0 & s_1 & \dots & s_N \end{pmatrix}$ with $s_i \in \{0, 1\}$. The initial state is thus $\mathbf{S}_0$.

Concerning the transition from one state to another, leading to a flow of tokens in the net, we only take a look at the strict sense concession rule:

- A transition is activated, one says, it has concession if all places which are linked through pre-arcs to the transition are marked with a token and all places which are connected to the transition through post-arcs are free.

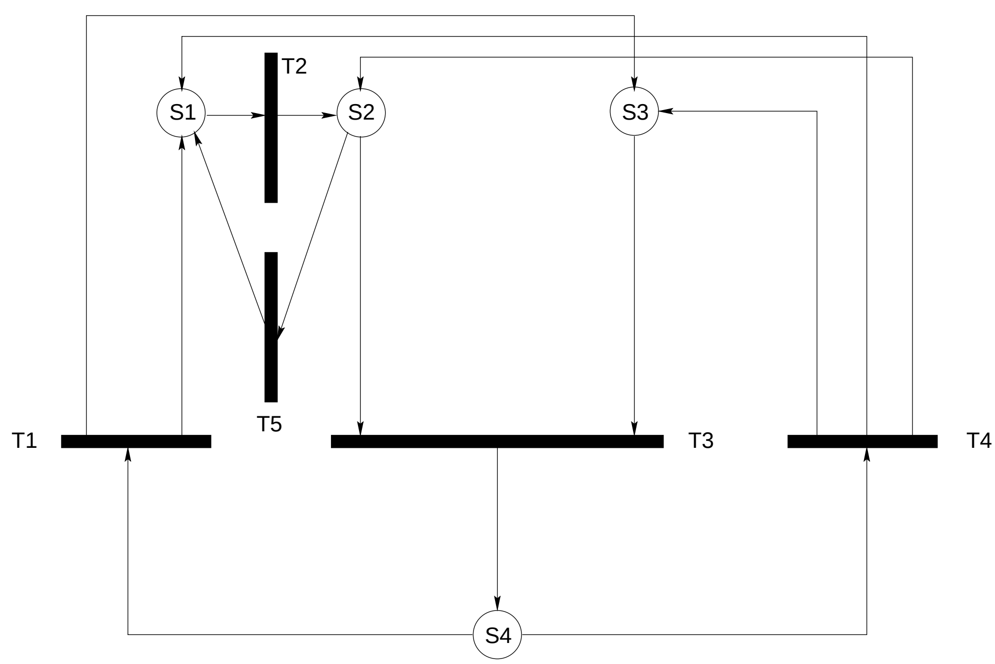
- During the switching process, all the places which are linked through pre-arcs to a transition are taken a token from and the ones which are linked through post-arcs are supplied with a token.

A.2 Struktural proprties

A.2.0.1 Reachability

A state $\mathbf{S}_i$ is reachable outgoing from the initial state $\mathbf{S}_0$ if a sequence of transitions exists which leads from $\mathbf{S}_0$ to $\mathbf{S}_i$.

Example:



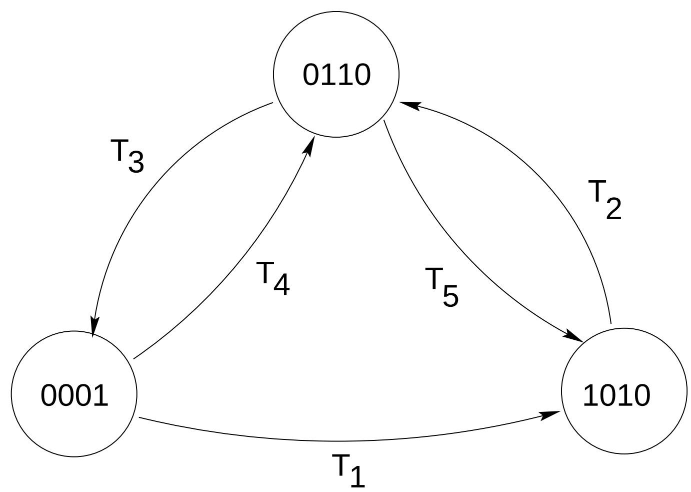
With the series of

$$\mathbf{S}_0 = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \quad (A.2)$$
$$\mathbf{S}_1 = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix} \quad (A.3)$$
$$\mathbf{S}_2 = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix} \quad (A.4)$$

All states which are reachable outgoing from $\mathbf{M}_0$ form the set of reachable states $\mathbb{RS}(\mathbf{S}_0)$. For the given Petri net, we obtain (considering the initial state) $\mathbf{S}_0 = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix}$ the following set of reachable states:

$$\mathbb{RS}\begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} = \left\{ \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \right\}$$

The graph that visualizes the reachable states is given in the following figure.



A.2.1 Deadlocks

A Petri net is called alive, if every transition of the net can be reactivated through a limited number of transitions. A deadlock is implied, if at least one transition cannot be activated, a total deadlock means that there exist states which cannot be exited, i.e. no transition which leads to another state can occur.

A.3 Algebraic View on Petri Nets

A Petri net can be described mathematically by giving its incidence matrix and the vector of transitions that have concession and are ready to switch.

$$\mathbf{S}_{k+1}^T = \mathbf{S}_k^T + \mathbf{N} \cdot \boldsymbol{\delta}_{k+1} \tag{A.5}$$

with:

$\mathbf{S}_k$	the vector of tokens in the places of the net in the k-th step
$\mathbf{S}_{k+1}$	the vector of tokens in the places of the net in the $k+1$-st step
$\mathbf{N}$	the incidence matrix
$\boldsymbol{\delta}$	the switching vector

Die gesamte Gleichung beschreibt den Markenflu"s in Abh"angigkeit von den schaltenden Transitionen. Die Inzidenzmatrix wird sinnvollerweise aufgeteilt, und zwar in:

- $\mathbf{N}^+$, describes the token flow of the post arcs, from the transitions to the places

- $\mathbf{N}^-$, describes the token flow of the pre acrs, from the places to the transitions

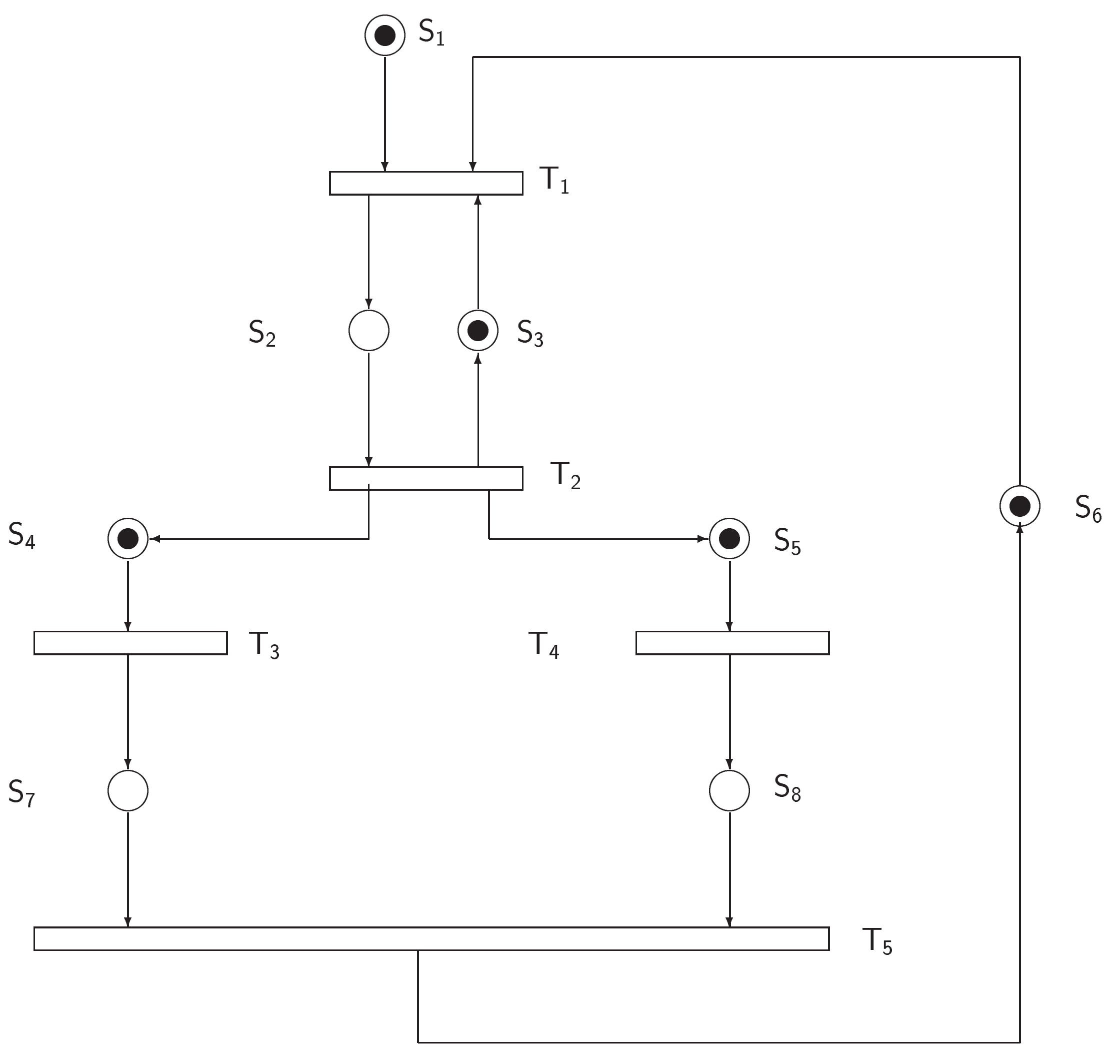
The incidence matrix $\mathbf{N}$ is calculated using $\mathbf{N}^+$ and $\mathbf{N}^-$, hence:

$$\mathbf{N} = \mathbf{N}^+ - \mathbf{N}^- \tag{A.6}$$

Generally, we have:

$$\begin{aligned} \mathbf{N}^- &= (S_i, T_j) && \text{(A.7)} \\ \mathbf{N}^+ &= (T_i, S_j)^T && \text{(A.8)} \end{aligned}$$

How one sets up the incidence matrices is described in the following *example*:



For the Petri net given above, we obtain the following incidence matrices:

1. Post arcs:

$$\mathbf{N}^{+} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{A.9}$$

2. Pre arcs:

$$\mathbf{N}^{-} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.10}$$

The incidence matrix is then:

$$\mathbf{N} = \mathbf{N}^{+} - \mathbf{N}^{-} \tag{A.11}$$

hence

$$\begin{aligned} \mathbf{N} &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix} \end{aligned} \tag{A.12}$$

For the calculation of the next state, given by the vector $\mathbf{S}$ of the net, we have to set up the switching vector; the switching vector contains the transitions which

are ready to switch in the $(k+1)$st step accordig to their switching condition. We denote the switching vector as $\boldsymbol{\delta}_{k+1}$.
Example: (continued) The initial state of the Petri net is given by $\mathbf{S}_0$ (initial marking)

$$\mathbf{S}_0 = \begin{pmatrix} S_{1_0} \\ S_{2_0} \\ S_{3_0} \\ S_{4_0} \\ S_{5_0} \\ S_{6_0} \\ S_{7_0} \\ S_{8_0} \end{pmatrix} \qquad \mathbf{S}_0 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{A.13}$$

Let us assume, transitions T_1, T_3 and T_4 are ready to switch (strict concession rule, all input places of transitions are marked, all output places are vacant):

$$\mathbf{S}_{k+1}^T = \mathbf{S}_k^T + \mathbf{N} \cdot \boldsymbol{\delta}_{k+1} \tag{A.14}$$

$$\mathbf{S}_{k+1} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{A.15}$$

The same result can be obtained by graphical processing of the net from the figure above.

A.4 Interpretation of Petri Nets

Interpreted Petri nets contain additional information. Whereas a non interpreted Petri net is a pure mathematical construction, interpreted Petri nets are frequently used in technical sciences. Many different interpretations are possible. Interpreted Petri nets are $\mathbb{IPN}$. In process control, interpreted Petri nets (see Litz [7]) $\mathbb{IPN}$ give important syntax extensions for the in- and output of information.

A.4.1 Interpretation for process control applications

1. The transitions are extended by introducing switching conditions - that are boolean or numerical expressions. These are called $SC(T_i)$. A switching

condition is fulfilled, if $SC(T_i) = 1$ - if the condition is true according to the boolean algebra.

2. The set of rules (concession rule, switching conditions) is extended by the so called synchronization rule: a transition in an interpreted Petri net switches if it is activated and if its switching condition is fulfilled. A transition is called activated if the concession rule is fulfilled (all places connected by pre arcs are marked, all places connected by post arcs are vavant. Marked places produce outputs. The whole output signal vector $(a_1, a_2, \ldots, a_n)$ is formed by the outputs which are generated by the set of marked places.

A.4.2 Interpretation as flow diagrams

Petri nets can also be used to depict the behaviour of algorithms. In this case the transitions contain the executed parts of the algorithm. The switching condition is given by e.g. values of variables and in general, places have no outputs.

Appendix B

Development Excercises

B.1 Object oriented programming

B.1.1 A simple example

Design three entities for polled i/o:

1. An i/o-board (ref. chapter 3) simulation entity
2. An i/o-board-handler entity
3. An entity which couples both the previously given entities and realizes a meta-system for them

B.1.2 An object oriented list

According to the details given in chapter 5 develop a list class that deals with the list's nodes. First, draw the object oriented entity diagram.

Your implementation should contain:

- Adding nodes
- Inserting nodes
- Removing nodes

Describe which member functions deal with internal tasks of the list and which ones are realizing the list's API.

B.2 User interfaces

The following excercises deal with the development of user-interfaces and QT's signal and slot mechanisms for message-handling.

B.2.1 Some buttons, signals and slots

Design a GUI using 3 different buttons. These buttons have to be connected to member functions (slots) of the button-containing class. For every pressed button a dialog has to open in which the user has to type in the new text for the button-label.

B.2.2 A dynamical visualization screen (oscilloscope)

The next step is to put the developed elements and our knowledge about GUI-development together. This yields a dynamical visualization screen. The screen should contain scroll-bars and a possibility to change the zoom factor of the axis. Further, the user should be able to visualize an arbitrarily long trajectory.

1. Draw the object oriented entity block diagram
2. Derive your oscilloscope class from a basic widget class
3. Test your oscilloscope entity by providing different trajectories. The trajectory-containing entity has to be well encapsulated in the oscilloscope

B.3 Hardware interface - a Frame Grabber Driver

This section deals with writing an interface to a hardware module - in the present case, which is especially useful in image processing

B.3.1 Interface description and design

The following description (from http://kernelnotes.org/doc23/video4linux/API.html) gives a brief overview of the device driver basics for a framegrabber device:

1. Devices

   Video4Linux provides the following sets of device files. These live on the character device formerly known as /dev/bttv. /dev/bttv should be a symlink to /dev/video0 for most people.

   ```
   Device Name
             Minor Range
                                 Function
   /dev/video
             0-63
                       Video Capture Interface
   /dev/radio
             64-127
                       AM/FM Radio Devices
   /dev/vtx
             192-223
                       Teletext Interface Chips
   /dev/vbi
             224-239
                       Raw VBI Data (Intercast/teletext)
   ```

   Video4Linux programs open and scan the devices to find what they are looking for. Capability queries define what each interface supports. The described API is only defined for video capture cards. The relevant subset applies to radio cards. Teletext interfaces talk the existing VTX API.

2. Capability Query Ioctl
   The VIDIOCGCAP ioctl call is used to obtain the capability information for a video device. The struct video_capability object passed to the ioctl is completed and returned. It contains the following information:

   ```
   name[32]
           Canonical name for this interface
   type
           Type of interface
   ```

channels
: Number of radio/tv channels if appropriate

audios
: Number of audio devices if appropriate

maxwidth
: Maximum capture width in pixels

maxheight
: Maximum capture height in pixels

minwidth
: Minimum capture width in pixels

minheight
: Minimum capture height in pixels

The type field lists the capability flags for the device. These are as follows

Name	Description
VID_TYPE_CAPTURE	Can capture to memory
VID_TYPE_TUNER	Has a tuner of some form
VID_TYPE_TELETEXT	Has teletext capability
VID_TYPE_OVERLAY	Can overlay its image onto the frame buffer
VID_TYPE_CHROMAKEY	Overlay is Chromakeyed
VID_TYPE_CLIPPING	Overlay clipping is supported
VID_TYPE_FRAMERAM	Overlay overwrites frame buffer memory
VID_TYPE_SCALES	The hardware supports image scaling
VID_TYPE_MONOCHROME	Image capture is grey scale only
VID_TYPE_SUBCAPTURE	Capture can be of only part of the image

The minimum and maximum sizes listed for a capture device do not imply all that all height/width ratios or sizes within the range are possible. A request to set a size will be honoured by the largest available capture size whose capture is no large than the requested rectangle in either direction. For example the quickcam has 3 fixed settings.

3. Frame Buffer

Capture cards that drop data directly onto the frame buffer must be told the base address of the frame buffer, its size and organisation. This is a privileged ioctl and one that eventually X itself should set.

The VIDIOCSFBUF ioctl sets the frame buffer parameters for a capture card. If the card does not do direct writes to the frame buffer then this ioctl will be unsupported. The VIDIOCGFBUF ioctl returns the currently used parameters. The structure used in both cases is a struct video_buffer.

```
void *base
            Base physical address of the buffer
int height
            Height of the frame buffer
int width
            Width of the frame buffer
int depth
            Depth of the frame buffer
int bytesperline
            Number of bytes of memory between
            the start of two adjacent lines
```

Note that these values reflect the physical layout of the frame buffer. The visible area may be smaller. In fact under XFree86 this is commonly the case. XFree86 DGA can provide the parameters required to set up this ioctl. Setting the base address to NULL indicates there is no physical frame buffer access.

4. Capture Windows

The capture area is described by a struct video_window. This defines a capture area and the clipping information if relevant. The VIDIOCGWIN ioctl recovers the current settings and the VIDIOCSWIN sets new values. A successful call to VIDIOCSWIN indicates that a suitable set of parameters have been chosen. They do not indicate that exactly what was requested was granted. The program should call VIDIOCGWIN to check if the nearest match was suitable. The struct video_window contains the following fields.

```
x
        The X co-ordinate specified in X windows format.
y
        The Y co-ordinate specified in X windows format.
width
        The width of the image capture.
height
        The height of the image capture.
chromakey
        A host order RGB32 value for the chroma key.
```

```
flags
        Additional capture flags.
clips
        A list of clipping rectangles. (Set only)
clipcount
        The number of clipping rectangles. (Set only)
```

Clipping rectangles are passed as an array. Each clip consists of the following fields available to the user.

```
x
     X co-ordinate of rectangle to skip
y
     Y co-ordinate of rectangle to skip
width
     Width of rectangle to skip
height
     Height of rectangle to skip
```

Merely setting the window does not enable capturing. Overlay capturing is activated by passing the VIDIOCCAPTURE ioctl a value of 1, and disabled by passing it a value of 0.

Some capture devices can capture a subfield of the image they actually see. This is indicated when VIDEO_TYPE_SUBCAPTURE is defined. The video_capture describes the time and special subfields to capture. The video_capture structure contains the following fields.

```
x
        X co-ordinate of source rectangle to grab
y
        Y co-ordinate of source rectangle to grab
width
        Width of source rectangle to grab
height
        Height of source rectangle to grab
decimation
        Decimation to apply
flags
        Flag settings for grabbing
```

```
The available flags are

        Name
                         Description
VIDEO_CAPTURE_ODD
                     Capture only odd frames
```

```
VIDEO_CAPTURE_EVEN
                    Capture only even frames
```

5. Video Sources
Each video4linux video or audio device captures from one or more source channels. Each channel can be queries with the VDIOCGCHAN ioctl call. Before invoking this function the caller must set the channel field to the channel that is being queried. On return the struct video_channel is filled in with information about the nature of the channel itself.
The VIDIOCSCHAN ioctl takes an integer argument and switches the capture to this input. It is not defined whether parameters such as colour settings or tuning are maintained across a channel switch. The caller should maintain settings as desired for each channel. (This is reasonable as different video inputs may have different properties).

The struct video_channel consists of the following elements:

```
channel
      The channel number
name
      The input name - preferably reflecting
      the label on the card input itself
tuners
      Number of tuners for this input
flags
      Properties the tuner has
type
      Input type (if known)
norm
      The norm for this channel
```

The flags defined are

```
VIDEO_VC_TUNER
               Channel has tuners.
VIDEO_VC_AUDIO
               Channel has audio.
VIDEO_VC_NORM
               Channel has norm setting.
```

The types defined are

```
VIDEO_TYPE_TV
                   The input is a TV input.
VIDEO_TYPE_CAMERA
                   The input is a camera.
```

6. Image Properties

The image properties of the picture can be queried with the VIDIOCGPICT ioctl which fills in a struct video_picture. The VIDIOCSPICT ioctl allows values to be changed. All values except for the palette type are scaled between 0-65535.

The struct video_picture consists of the following fields:

```
brightness
        Picture brightness
hue
        Picture hue (colour only)
colour
        Picture colour (colour only)
contrast
        Picture contrast
whiteness
        The whiteness (greyscale only)
depth
        The capture depth (may need to match the frame buffer depth)
palette
        Reports the palette that should be used for this image
```

The following palettes are defined:

```
VIDEO_PALETTE_GREY
                      Linear intensity grey scale (255 is brightest).
VIDEO_PALETTE_HI240
                      The BT848 8bit colour cube.
VIDEO_PALETTE_RGB565
                      RGB565 packed into 16 bit words.
VIDEO_PALETTE_RGB555
                      RGV555 packed into 16 bit words,
                      top bit undefined.
VIDEO_PALETTE_RGB24
                      RGB888 packed into 24bit words.
VIDEO_PALETTE_RGB32
                      RGB888 packed into the low 3 bytes of 32bit words.
                      The top 8bits are undefined.
VIDEO_PALETTE_YUV422
                      Video style YUV422 - 8bits packed
                      4bits Y 2bits U 2bits V
VIDEO_PALETTE_YUYV
                      Describe me
VIDEO_PALETTE_UYVY
                      Describe me
```

```
VIDEO_PALETTE_YUV420
                      YUV420 capture
VIDEO_PALETTE_YUV411
                      YUV411 capture
VIDEO_PALETTE_RAW
                      RAW capture (BT848)
VIDEO_PALETTE_YUV422P
                      YUV 4:2:2 Planar
VIDEO_PALETTE_YUV411P
                      YUV 4:1:1 Planar
```

7. Tuning

   Each video input channel can have one or more tuners associated with it. Many devices will not have tuners. TV cards and radio cards will have one or more tuners attached.
   Tuners are described by a struct video_tuner which can be obtained by the VIDIOCGTUNER ioctl. Fill in the tuner number in the structure then pass the structure to the ioctl to have the data filled in. The tuner can be switched using VIDIOCSTUNER which takes an integer argument giving the tuner to use. A struct tuner has the following fields:

```
tuner
        Number of the tuner
name
        Canonical name for this tuner (eg FM/AM/TV)
rangelow
        Lowest tunable frequency
rangehigh
        Highest tunable frequency
flags
        Flags describing the tuner
mode
        The video signal mode if relevant
signal
        Signal strength if known - between 0-65535
```

   The following flags exist

```
VIDEO_TUNER_PAL
                       PAL tuning is supported
VIDEO_TUNER_NTSC
                       NTSC tuning is supported
VIDEO_TUNER_SECAM
                       SECAM tuning is supported
```

```
VIDEO_TUNER_LOW
                        Frequency is in a lower range
VIDEO_TUNER_NORM
                        The norm for this tuner is settable
VIDEO_TUNER_STEREO_ON
                        The tuner is seeing stereo audio
VIDEO_TUNER_RDS_ON
                        The tuner is seeing a RDS datastream
VIDEO_TUNER_MBS_ON
                        The tuner is seeing a MBS datastream
```

The following modes are defined

```
VIDEO_MODE_PAL
                   The tuner is in PAL mode
VIDEO_MODE_NTSC
                   The tuner is in NTSC mode
VIDEO_MODE_SECAM
                   The tuner is in SECAM mode
VIDEO_MODE_AUTO
                   The tuner auto switches
                   or mode does not apply
```

Tuning frequencies are an unsigned 32bit value in 1/16th MHz or if the VIDEO_TUNER_LOW flag is set they are in 1/16th KHz. The current frequency is obtained as an unsigned long via the VIDIOCGFREQ ioctl and set by the VIDIOCSFREQ ioctl.

8. Audio

TV and Radio devices have one or more audio inputs that may be selected. The audio properties are queried by passing a struct video_audio to VIDIOCGAUDIO ioctl. The VIDIOCSAUDIO ioctl sets audio properties.

The structure contains the following fields:

```
audio
      The channel number
volume
      The volume level
bass
      The bass level
treble
      The treble level
flags
      Flags describing the audio channel
```

```
name
      Canonical name for the audio input
mode
      The mode the audio input is in
balance
      The left/right balance
step
      Actual step used by the hardware
```

The following flags are defined

```
VIDEO_AUDIO_MUTE
                      The audio is muted
VIDEO_AUDIO_MUTABLE
                      Audio muting is supported
VIDEO_AUDIO_VOLUME
                      The volume is controllable
VIDEO_AUDIO_BASS
                      The bass is controllable
VIDEO_AUDIO_TREBLE
                      The treble is controllable
VIDEO_AUDIO_BALANCE
                      The balance is controllable
```

The following decoding modes are defined

```
VIDEO_SOUND_MONO
                     Mono signal
VIDEO_SOUND_STEREO
                     Stereo signal (NICAM for TV)
VIDEO_SOUND_LANG1
                     European TV alternate language 1
VIDEO_SOUND_LANG2
                     European TV alternate language 2
```

9. Reading Images

Each call to the read syscall returns the next available image from the device. It is up to the caller to set the format and then to pass a suitable size buffer and length to the function. Not all devices will support read operations.

A second way to handle image capture is via the mmap interface if supported. To use the mmap interface a user first sets the desired image size and depth properties. Next the VIDIOCGMBUF ioctl is issued. This reports the size of buffer to mmap and the offset within the buffer for each frame. The number of frames supported is device dependent and may only be one.

The video_mbuf structure contains the following fields:

```
size
     The number of bytes to map
frames
     The number of frames
offsets
     The offset of each frame
```

Once the mmap has been made the VIDIOCMCAPTURE ioctl sets the image size you wish to use (which should match or be below the initial query size). Having done so it will begin capturing to the memory mapped buffer. Whenever a buffer is "used" by the program it should called VIDIOCSYNC to free this frame up and continue. to add:VIDIOCSYNC takes the frame number you are freeing as its argument. When the buffer is unmapped or all the buffers are full capture ceases. While capturing to memory the driver will make a "best effort" attempt to capture to screen as well if requested. This normally means all frames that "miss" memory mapped capture will go to the display.

A final ioctl exists to allow a device to obtain related devices if a driver has multiple components (for example video0 may not be associated with vbi0 which would cause an intercast display program to make a bad mistake). The VIDIOCGUNIT ioctl reports the unit numbers of the associated devices if any exist. The video_unit structure has the following fields.

```
video
      Video capture device
vbi
      VBI capture device
radio
      Radio device
audio
      Audio mixer
teletext
      Teletext device
```

10. RDS Datastreams

For radio devices that support it, it is possible to receive Radio Data System (RDS) data by means of a read() on the device. The data is packed in groups of three, as follows:

```
First Octet
         Least Significant Byte of RDS Block
Second Octet
         Most Significant Byte of RDS Block
Third Octet
         Bit 7:
         Error bit. Indicates that an uncorrectable
         error occurred during reception of this block.

         Bit 6:
         Corrected bit. Indicates that an error was
         corrected for this data block.

         Bits 5-3:
         Received Offset. Indicates the offset
         received by the sync system.

         Bits 2-0:
         Offset Name. Indicates the offset applied
         to this data.
```

B.3.2 Frame Grabber Image Acquisition Application

For a test of the video-interface handling entity, you have to write an object oriented program which acquires frames from a certain video-channel and shows them on the screen. For this purpose build a main-widget-class which contains the necessary paint-methods and a button for acquiring a new image.

B.4 Control system operator blocks

The following sections are dedicated to the design of control loops, either for simulation purposes or additionally for coupling with hardware. The blocks are standard-blocks and integration as well as summation have to be set up for sampled data systems.

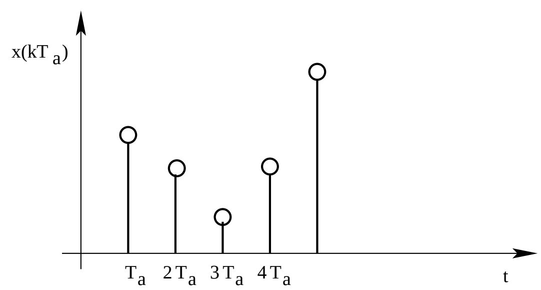


Figure B.1: A sampled data sequence

B.4.1 Integrator

An integrator block integrates the input value and passes the actual integrator value to the output of a system. There are different methods of integration.

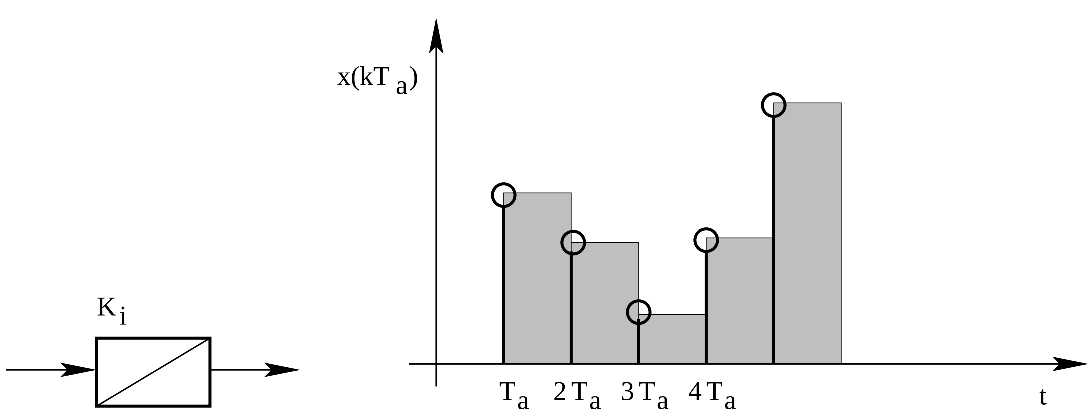


Figure B.2: Basic integration algorithm (numerical)

B.4.2 Differentiator

B.4.3 Summation Point

B.4.4 Scope

As a scope you can include the viewer entity which we designed in the beginning. Now you have to connect its functionality properly to use it as a scope operator.

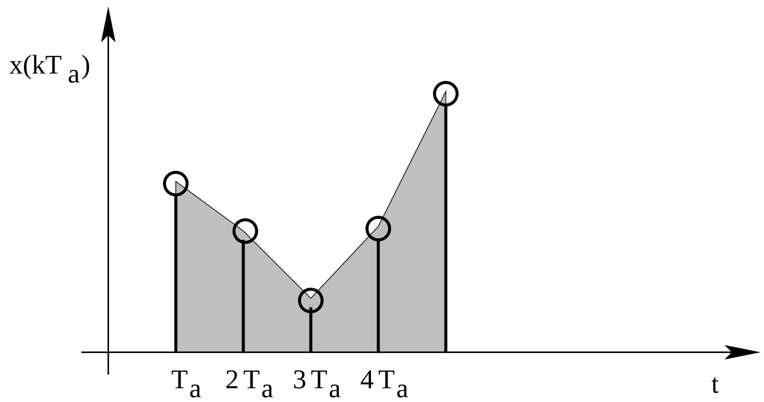


Figure B.3: A trapezoid integration algorithm

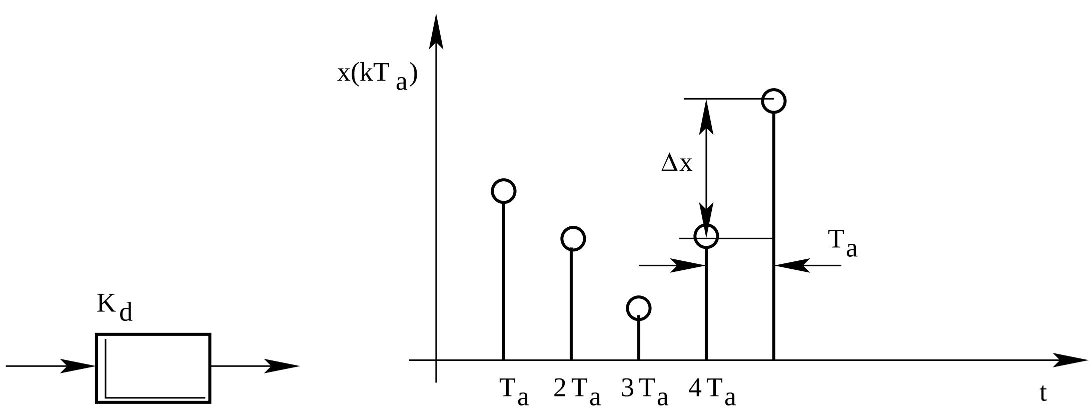


Figure B.4: Differentiation in the sampled data domain

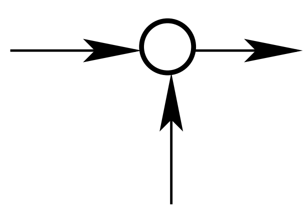


Figure B.5: A sum point for block diagrams

B.4.5 Graphical Input for the given blocks

The designed blocks are now ready for a user-configurable display. The user has to be able to select any specifications of the underlying block. These are:

- The amplification K in the differentiator and integrator block
- The signs for the inputs of the summation point

B.5 A Metasystem for a control loop

A metasystem - if we recall our definition - was a superior level system which had to invoke the appropriate member functions of the underlying entities (either by direct call or by passing appropriate messages). For a control loop, which is built up using the object oriented entity concept, it is further necessary to execute these functions in an appropriate manner and also provide workarounds for algebraic loops and so on. We will focus only on the first part, as the solutions to e.g. algebraic loops would be beyond our scope.

B.5.1 An execution structure for the control blocks

First, think of how the given sampled data-system structure could be executed. Think of data exchange in case of a multiple input multiple output system.

B.5.2 A hard wired control loop simulator

B.5.2.1 Simple case

Put up a fixed control-system structure with the plant given in Fig. B.6. first exchange your data by directly copying values from one to another operator block

B.5.2.2 General case

Additionally add dynamic data handling by providing the outputs of the operators in a dynamic list. Invent a special node entity which contains a name for the variable and the variable itself. Add in all your graphical interface entities the possibility to see the name of the selected variable.

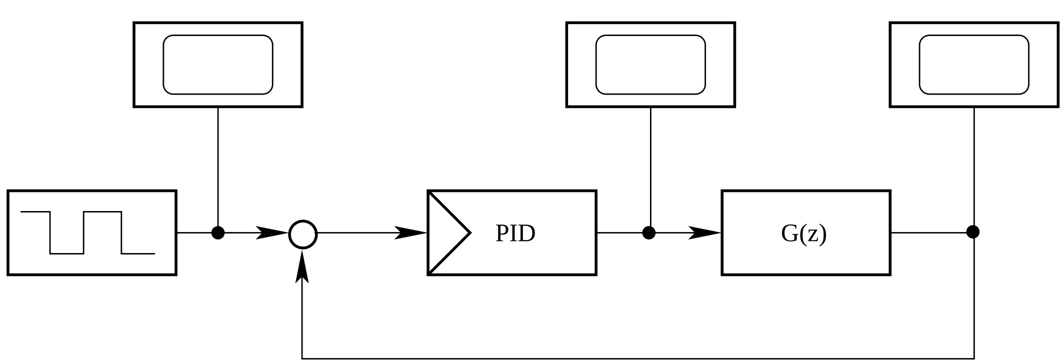


Figure B.6: A sample control loop - try to make modifications to your structure or to introduce a dynamic user interface

B.6 Realtime issues

B.6.1 Set and reset parallel port signals

Set and reset signals of the parallel port with a realtime algorithm. Make the timing faster and check with an oscilloscope whether the rectangular output varies or is inaccurate.

B.6.2 Data acquisition from an A/D-board

Acquire an input value from an A/D-board and write the same value to the output of the A/D-board.

B.6.3 Coupling Realtime and User Interface

Show the acquired trajectory on your dynamic visualization screen. Allow the user to specify a delay between acquisition and output and measure both, in- and output value (to and from the A/D-D/A board) on an oscilloscope to determine whether the timings are accurate

B.7 Application Testing

B.7.1 Separate Entities

Build up the tank system simulation entity and the Petri net for the logic control, both with an appropriate visualization. Introduce buttons for every signal which can be pressed by the user to test the given functionality. The integration for the tank entity does not require a realtime approach.

B.7.2 Object oriented way of combing Entities

Combine the predescribed entities and use member variables of the entities for data transfer. The only button left for the user is now the activation signal for Valve V_3.

B.7.3 Testing with hardware-cabling

The last step is to setup the application for logic control on one computer and on the other one the application which simulates the tank system. Both should exchange the values of their sensor signals using serial port.
The following text was taken from the Serial-Programming Howto by

1. Port Settings
   The devices /dev/ttyS* are intended to hook up terminals to your Linux box, and are configured for this use after startup. This has to be kept in mind when programming communication with a raw device. E.g. the ports are configured to echo characters sent from the device back to it, which normally has to be changed for data transmission.

   All parameters can be easily configured from within a program. The configuration is stored in a structure struct termios, which is defined in <asm/termbits.h¿:

```
#define NCCS 19
struct termios {
        tcflag_t c_iflag;          /* input mode flags */
        tcflag_t c_oflag;          /* output mode flags */
        tcflag_t c_cflag;          /* control mode flags */
        tcflag_t c_lflag;          /* local mode flags */
        cc_t c_line;               /* line discipline */
        cc_t c_cc[NCCS];           /* control characters */
};
```

   This file also includes all flag definitions. The input mode flags in c_iflag handle all input processing, which means that the characters sent from the device can be processed before they are read with read. Similarly c_oflag handles the output processing. c_cflag contains the settings for the port, as the baudrate, bits per character, stop bits, etc.. The local mode flags stored in c_lflag determine if characters are echoed, signals are sent to your program, etc.. Finally the array c_cc defines the control characters for end of file, stop, etc. Default values for the control characters are defined in

<

asm/termios.h>. The flags are described in the manual page termios(3). The structure termios contains the c_line (line discipline) element, which is not used in POSIX compliant systems.

2. Input Concepts for Serial Devices
   Here three different input concepts will be presented. The appropriate concept has to be chosen for the intended application. Whenever possible, do not loop reading single characters to get a complete string. When I did this, I lost characters, whereas a read for the whole string did not show any errors.

   (a) Canonical Input Processing
       This is the normal processing mode for terminals, but can also be useful for communicating with other dl input is processed in units of lines, which means that a read will only return a full line of input. A line is by default terminated by a NL (ASCII LF), an end of file, or an end of line character. A CR (the DOS/Windows default end-of-line) will not terminate a line with the default settings. Canonical input processing can also handle the erase, delete word, and reprint characters, translate CR to NL, etc.

   (b) Non-Canonical Input Processing
       Non-Canonical Input Processing will handle a fixed amount of characters per read, and allows for a character timer. This mode should be used if your application will always read a fixed number of characters, or if the connected device sends bursts of characters.

   (c) Asynchronous Input
       The two modes described above can be used in synchronous and asynchronous mode. Synchronous is the default, where a read statement will block, until the read is satisfied. In asynchronous mode the read statement will return immediatly and send a signal to the calling program upon completion. This signal can be received by a signal handler.

   (d) Waiting for Input from Multiple Sources
       This is not a different input mode, but might be useful, if you are handling multiple devices. In my application I was handling input over a TCP/IP socket and input over a serial connection from another computer quasi-simultaneously. The program example given below will wait for input from two different input sources. If input from one source becomes available, it will be processed, and the program will then wait for new input.
       The approach presented below seems rather complex, but it is important to keep in mind that Linux is a multi-processing operating system. The select

system call will not load the CPU while waiting for input, whereas looping until input becomes available would slow down other processes executing at the same time.

The following text contains programming examples for the given port-modes to make it easier to understand the specifications.

1. General
   All examples have been derived from miniterm.c. The type ahead buffer is limited to 255 characters, just like the maximum string length for canonical input processing (<linux/limits.h> or <posix1_lim.h>).
   See the comments in the code for explanation of the use of the different input modes. I hope that the code is understandable. The example for canonical input is commented best, the other examples are commented only where they differ from the example for canonical input to emphasize the differences.

   The descriptions are not complete, but you are encouraged to experiment with the examples to derive the best solution for your application.

   Don't forget to give the appropriate serial ports the right permissions (e. g.: chmod a+rw /dev/ttyS1)!

2. Canonical Input Processing

```
#include $<$sys/types.h$>$
#include $<$sys/stat.h$>$
#include $<$fcntl.h$>$
#include $<$termios.h$>$
#include $<$stdio.h$>$

/* baudrate settings are defined in $<$asm/termbits.h$>$, which is
included by $<$termios.h$>$ */
#define BAUDRATE B38400
/* change this definition for the correct port */
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
  int fd,c, res;
```

```
  struct termios oldtio,newtio;
  char buf[255];
/*
  Open modem device for reading and writing and not as
  controlling tty because we don't want to get killed if
  linenoise sends CTRL-C.
*/
 fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
 if (fd $<$0) {perror(MODEMDEVICE); exit(-1); }

/* save current serial port settings */
 tcgetattr(fd,&oldtio);

/* clear struct for new port settings */
 bzero(&newtio, sizeof(newtio));

/*
  BAUDRATE: Set bps rate. You could also use
            cfsetispeed and cfsetospeed.
  CRTSCTS : output hardware flow control
            (only used if the cable has
            all necessary lines)
  CS8     : 8n1 (8bit,no parity,1 stopbit)
  CLOCAL  : local connection, no modem contol
  CREAD   : enable receiving characters
*/
 newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;

/*
  IGNPAR  : ignore bytes with parity errors
  ICRNL   : map CR to NL (otherwise a CR input on
            the other computer will not terminate input)
  otherwise make device raw (no other input processing)
*/
 newtio.c_iflag = IGNPAR | ICRNL;

/*
 Raw output.
*/
 newtio.c_oflag = 0;

/*
  ICANON  : enable canonical input
  disable all echo functionality, and don't send signals
  to calling program
*/
 newtio.c_lflag = ICANON;
```

```
/*
  initialize all control characters
  default values can be found in /usr/include/termios.h,
  and are given in the comments, but we don't need them here
*/
 newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
 newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
 newtio.c_cc[VERASE]   = 0;     /* del */
 newtio.c_cc[VKILL]    = 0;     /* @ */
 newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
 newtio.c_cc[VTIME]    = 0;     /* inter-character
                                   timer unused */
 newtio.c_cc[VMIN]     = 1;     /* blocking read
                                   until 1 character
                                   arrives */
 newtio.c_cc[VSWTC]    = 0;     /* '\0' */
 newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
 newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
 newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
 newtio.c_cc[VEOL]     = 0;     /* '\0' */
 newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
 newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
 newtio.c_cc[VWERASE]  = 0;     /* Ctrl-w */
 newtio.c_cc[VLNEXT]   = 0;     /* Ctrl-v */
 newtio.c_cc[VEOL2]    = 0;     /* '\0' */

/*
  now clean the modem line and activate
  the settings for the port
*/
 tcflush(fd, TCIFLUSH);
 tcsetattr(fd,TCSANOW,&newtio);

/*
  terminal settings done, now handle input
  In this example, inputting a 'z' at the
  beginning of a line will
  exit the program.
*/
 while (STOP==FALSE) {
 /* loop until we have a terminating condition */

 /* read blocks program execution until
    a line terminating character is
    input, even if more than 255 chars are input.
    If the number of characters read is smaller
```

```
        than the number of chars available, subsequent
        reads will return the remaining chars. res will
        be set to the actual number of characters
        actually read */

        res = read(fd,buf,255);

    /* set end of string, so we can printf */

        buf[res]=0;
        printf(":%s:%d\n", buf, res);
        if (buf[0]=='z') STOP=TRUE;
     }
     /* restore the old port settings */
     tcsetattr(fd,TCSANOW,&oldtio);
    }
```

3. Non-Canonical Input Processing

   In non-canonical input processing mode, input is not assembled into lines and input processing (erase, kill, delete, etc.) does not occur. Two parameters control the behavior of this mode: c_cc[VTIME] sets the character timer, and c_cc[VMIN] sets the minimum number of characters to receive before satisfying the read.

   If MIN $>$ 0 and TIME $=$ 0, MIN sets the number of characters to receive before the read is satisfied. As TIME is zero, the timer is not used.

   If MIN $=$ 0 and TIME $>$ 0, TIME serves as a timeout value. The read will be satisfied if a single character is read, or TIME is exceeded (t = TIME *0.1 s). If TIME is exceeded, no character will be returned.

   If MIN $>$ 0 and TIME $>$ 0, TIME serves as an inter-character timer. The read will be satisfied if MIN characters are received, or the time between two characters exceeds TIME. The timer is restarted every time a character is received and only becomes active after the first character has been received.

   If MIN $=$ 0 and TIME $=$ 0, read will be satisfied immediately. The number of characters currently available, or the number of characters requested will be returned. According to Antonino (see contributions), you could issue a fcntl(fd, F_SETFL, FNDELAY); before reading to get the same result.

   By modifying newtio.c_cc[VTIME] and newtio.c_cc[VMIN] all modes described above can be tested.

```
    #include $<$sys/types.h$>$
    #include $<$sys/stat.h$>$
    #include $<$fcntl.h$>$
    #include $<$termios.h$>$
    #include $<$stdio.h$>$
```

```
#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  char buf[255];

 fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
 if (fd $<$0) {perror(MODEMDEVICE); exit(-1); }

 tcgetattr(fd,&oldtio); /* save current port settings */

 bzero(&newtio, sizeof(newtio));
 newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
 newtio.c_iflag = IGNPAR;
 newtio.c_oflag = 0;

 /* set input mode (non-canonical, no echo,...) */
 newtio.c_lflag = 0;

 newtio.c_cc[VTIME]    = 0;   /* inter-character
                                 timer unused */
 newtio.c_cc[VMIN]     = 5;   /* blocking read until
                                 5 chars received */

 tcflush(fd, TCIFLUSH);
 tcsetattr(fd,TCSANOW,&newtio);


 while (STOP==FALSE) {       /* loop for input */
   res = read(fd,buf,255);   /* returns after 5 chars
                                have been input */
   buf[res]=0;               /* so we can printf... */
   printf(":%s:%d\n", buf, res);
   if (buf[0]=='z') STOP=TRUE;
 }
 tcsetattr(fd,TCSANOW,&oldtio);
}
```

4. Asynchronous Input

```
   #include $<$termios.h$>$
   #include $<$stdio.h$>$
   #include $<$unistd.h$>$
   #include $<$fcntl.h$>$
   #include $<$sys/signal.h$>$
   #include $<$sys/types.h$>$

   #define BAUDRATE B38400
   #define MODEMDEVICE "/dev/ttyS1"
   #define _POSIX_SOURCE 1 /* POSIX compliant source */
   #define FALSE 0
   #define TRUE 1

   volatile int STOP=FALSE;

/* definition of signal handler */
   void signal_handler_IO (int status);
   int wait_flag=TRUE;  /* TRUE while no signal received */

   main()
   {
     int fd,c, res;
     struct termios oldtio,newtio;
     struct sigaction saio;   /* definition of signal action */
     char buf[255];

     /* open the device to be non-blocking
       (read will return immediatly) */
     fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
     if (fd $<$0) {perror(MODEMDEVICE); exit(-1); }

     /* install the signal handler before
        making the device asynchronous */
     saio.sa_handler = signal_handler_IO;
     saio.sa_mask = 0;
     saio.sa_flags = 0;
     saio.sa_restorer = NULL;
     sigaction(SIGIO,&saio,NULL);

     /* allow the process to receive SIGIO */
     fcntl(fd, F_SETOWN, getpid());
     /* Make the file descriptor asynchronous
       (the manual page says only O_APPEND and O_NONBLOCK,
        will work with F_SETFL...) */
     fcntl(fd, F_SETFL, FASYNC);
```

```
  tcgetattr(fd,&oldtio); /* save current port settings */
  /* set new port settings for canonical input processing */
  newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR | ICRNL;
  newtio.c_oflag = 0;
  newtio.c_lflag = ICANON;
  newtio.c_cc[VMIN]=1;
  newtio.c_cc[VTIME]=0;
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);

  /* loop while waiting for input.
     normally we would do something
     useful here */

  while (STOP==FALSE) {
    printf(".\n");usleep(100000);

   /* after receiving SIGIO,
      wait_flag = FALSE, input is available
      and can be read */

   if (wait_flag==FALSE) {
      res = read(fd,buf,255);
      buf[res]=0;
      printf(":%s:%d\n", buf, res);
      if (res==1) STOP=TRUE; /* stop loop if only
                               a CR was input */
      wait_flag = TRUE;      /* wait for new input */
    }
  }
  /* restore old port settings */
  tcsetattr(fd,TCSANOW,&oldtio);
}

/********************************************
* signal handler. sets wait_flag to FALSE    *
* to indicate above loop that               *
* characters have been received.            *
*********************************************/

void signal_handler_IO (int status)
{
  printf("received SIGIO signal.\n");
  wait_flag = FALSE;
}
```

5. Waiting for Input from Multiple Sources

This section is kept to a minimum. It is just intended to be a hint, and therefore the example code is kept short. This will not only work with serial ports, but with any set of file descriptors.

The select call and accompanying macros use a fd_set. This is a bit array, which has a bit entry for every valid file descriptor number. select will accept a fd_set with the bits set for the relevant file descriptors and returns a fd_set, in which the bits for the file descriptors are set where input, output, or an exception occurred. All handling of fd_set is done with the provided macros. See also the manual page select(2).

```
#include $<$sys/time.h$>$
#include $<$sys/types.h$>$
#include $<$unistd.h$>$

main()
{
   int    fd1, fd2;  /* input sources 1 and 2 */
   fd_set readfs;    /* file descriptor set */
   int    maxfd;     /* maximum file desciptor used */
   int    loop=1;    /* loop while TRUE */

   /* open_input_source opens a device,
      sets the port correctly, and
      returns a file descriptor */

   fd1 = open_input_source("/dev/ttyS1");   /* COM2 */
   if (fd1$<$0) exit(0);
   fd2 = open_input_source("/dev/ttyS2");   /* COM3 */
   if (fd2$<$0) exit(0);

   /* maximum bit entry (fd) to test */
   maxfd = MAX (fd1, fd2)+1;

   /* loop for input */
   while (loop) {
     FD_SET(fd1, &readfs);  /* set testing for source 1 */
     FD_SET(fd2, &readfs);  /* set testing for source 2 */
     /* block until input becomes available */
     select(maxfd, &readfs, NULL, NULL, NULL);

     if (FD_ISSET(fd1))
       /* input from source 1 available */
       handle_input_from_source1();
```

```
      if (FD_ISSET(fd2))
        /* input from source 2 available */
        handle_input_from_source2();
    }

}
```

The given example blocks indefinitely, until input from one of the sources becomes available. If you need to timeout on input, just replace the select call by:

```
int res;
struct timeval Timeout;

/* set timeout value within input loop */
Timeout.tv_usec = 0;  /* milliseconds */
Timeout.tv_sec  = 1;  /* seconds */
res = select(maxfd, &readfs, NULL, NULL, &Timeout);
if (res==0)
/* number of file descriptors with input = 0,
   timeout occurred. */
```

This example will timeout after 1 second. If a timeout occurs, select will return 0, but beware that Timeout is decremented by the time actually waited for input by select. If the timeout value is zero, select will return immediatly.

Appendix C

Solutions to selected Problems

C.1 A Framegrabber-Interface

In the following, only the most important pieces of code are printed.

1. Initializing the VideoInterface Entity which handles the video for Linux driver. First, the constructor is invoked - it sets the name of the device-driver to "/dev/video" and then starts the basis initialization of the driver by calling the member-function baseInit().

```
VideoInterface::VideoInterface()
{
 m_videodevice="/dev/video";
 m_defaultchannelnumber=1;//VIDEO_EINGANG;
 m_Image=new(QImage);
 m_Image-$>$create(768,576,32,0, QImage::IgnoreEndian );
 baseInit();
}
```

2. The code for basis-initialization of the framegrabber device

```
int VideoInterface::baseInit()
{
 //For basic configuration of the video interface
 //(if live-video is needed), we have to set
 //memory adresses which needs root-access

  QString kv4lcall("kv4lsetup -q -l ");
  kv4lcall+="/dev/bttv";

  kv4lcall.append(" -b ");
  QString sbpp;
  sbpp.setNum(16); //Set number of bits/pixel
```

```
  kv4lcall+=sbpp;

  switch ( system((const char *)kv4lcall) )
  {
  case -1:
    warning("could'nt start kv4lsetup!\n");
    break;
  case 0:
    break;
  default:
    warning("kv4lsetup had some trouble,
             trying to continue anyway.\n");
    break;
  }
  return 0;
}
```

Now the device-driver is set up by first opening the device

```
int VideoInterface::openDevice()
{
 if ( -1 == (m_devv4l=::open(m_videodevice,O_RDWR)) )
 {
  fatal(``Error opening video device",m_videodevice,strerror(errno));
 }

 return 0;
}
```

using the device-driver specific ioctl-commands, we setup the interface:

```
int VideoInterface::configureInterface()
{
 //Query Capabilities of Framegrabber
 if( -1 == ioctl(m_devv4l,VIDIOCGCAP,&m_videocap) )
    warning("videointerface: VIDIOC_G_CAP in ::videointerfaceif");

 //Obtain Frame buffer Parameters
  if (  -1 == ioctl(m_devv4l, VIDIOCGFBUF, &m_videobuf) )
    warning("videointerface: ioctl VIDIOCGFBUF in ::videointerfaceif");

 //check video-source and set correct input
 m_channel.channel=m_defaultchannelnumber;
 if (  -1 == ioctl(m_devv4l, VIDIOCGCHAN, &m_channel) )
    warning("videointerface: ioctl VIDIOCGCHAN in ::videointerfaceif");

 m_channel.channel=m_defaultchannelnumber;
```

```
 if (  -1 == ioctl(m_devv4l, VIDIOCSCHAN, &m_channel) )
    warning("videointerface: ioctl VIDIOCSCHAN in ::videointerfaceif");

 //Acquire information on image
 if (  -1 == ioctl(m_devv4l, VIDIOCGPICT, &m_picture) )
    warning("ioctl VIDIOCGPICT failed");

 //Query tuner

 if (m_channel.tuners$>$0)
 {
    if (  -1 == ioctl(m_devv4l, VIDIOCGTUNER, &m_tuner) )
    warning("ioctl VIDIOTUNER failed");
 }

 //Calculate the size of the framebuffer
 m_size=m_videobuf.height*m_videobuf.width*(24$>$$>$3);

 // Alloc memory for snapshot
 m_grabbermem=(char *)mmap(0,m_size,PROT_READ
                          |PROT_WRITE,MAP_SHARED,
                          m_devv4l,0);

 if ((char*)-1 == m_grabbermem)
   warning("unable to allocate memory for snap shots!");

return 0;
}
```

3. Images can be grabbed using the following member function

```
int VideoInterface::Grab()
{

  m_memorymap.frame=0; /* Frame (0 - n) for double buffer */
  m_memorymap.height=576;
  m_memorymap.width=768;
  m_memorymap.format=VIDEO_PALETTE_RGB24;

  if ( -1 == ioctl(m_devv4l,VIDIOCMCAPTURE,&m_memorymap))
  {
    if (errno == EAGAIN)
    {
      warning("videointerface: Grabber chip can't sync");
      return false;
    }
    else
```

```
    {
      fatal("videointerface: VIDIOCMCAPTURE in ::grabOne: %s",strerror(errno));
    }
  }

//Frame 0 starts at m_grabbermem.
//Wait for Frame

  if (-1 ==  ioctl(m_devv4l,VIDIOCSYNC,&m_memorymap.frame))
     warning("waiting not possible");

//Schreibe gegrabbtes Bild in QImage
 int i,j,index=0;
 unsigned int *rgb;
 m_Image-$>$fill ( 0x00dd00 );

 for(i=0;i$<$576;i++)
  for(j=0;j$<$768;j++)
   {
      rgb = (unsigned int *)m_Image-$>$scanLine(i) + j;
     *rgb = qRgb(m_grabbermem[index],m_grabbermem[index+1],
                 m_grabbermem[index+2]);

    index+=3;
   }

 return 0;
}
```

4. Finally, the videointerface is closed and all memory is freed after the video-device has been closed

```
int VideoInterface::closeDevice()
{
  //unmap memory mapped frame memory

  if ( ((char*)-1) != m_grabbermem) munmap(m_grabbermem,m_size);
  //close device
  if ( -1 == ::close(m_devv4l) )
  {
   warning("videointerface: Error closing video device"
           ,m_videodevice,strerror(errno));
  }
 return 0;
}

VideoInterface::~VideoInterface()
```

```
{
 closeDevice();
        if (  -1 == ioctl(m_devv4l, VIDIOCCAPTURE, &zero) )
        warning("ioctl VIDIOCCAPTURE impossible");

 delete (m_Image);
}
```

Appendix D

Sampled Data Systems

For sampled data systems, one can decide between approximation of continuous time systems (especially for simulation purposes) or a discrete time approach. The state space representation is immensely useful for deriving integration algorithms in order to approximate continuous time systems' behaviour. Both approaches are given in the following text[1].

D.1 Numerical integration algorithms

First order differential equations can be integrated using different methods. In the following sections, we derive the explicit Euler algorithm (the easiest integration algorithm) and also integration according to Heun. The difference is that Heun uses two reference points in order to calculate the derivative in the $(k+1)$st step. Integration according to Runge Kutta (RK) uses even more points (RK-3, RK-4, RK-5). Many systems (like MATLAB for example) use RK-4 and for supervision of the stepsize use RK-5.

D.1.1 Explicit Euler Integration

Assume a state space system as given below:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{D.1}$$

Required is the trajectory of $\mathbf{x}(t)$. The system time is then discretized assuming a step size h (see Fig. D.1:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t)\big|_{t=k\cdot h} \tag{D.2}$$

We obtain:

[1]See Tuttas [16] concerning discrete integration algoritms and Pandit [11] or Foellinger [4] comcerning sampled data control systems

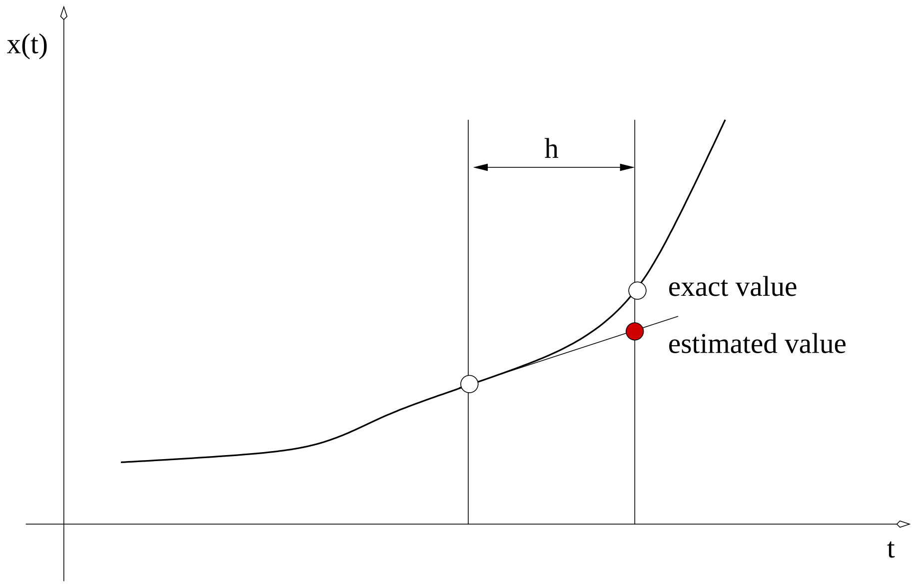


Figure D.1: Euler integration method

$$\dot{\mathbf{x}}(k \cdot h) = \mathbf{A} \cdot \mathbf{x}(k \cdot h) + \mathbf{B} \cdot \mathbf{u}(k \cdot h) \tag{D.3}$$

Now we calculate $\mathbf{x}((k+1) \cdot h)$ using h and the derivative $\dot{\mathbf{x}}(k \cdot h)$:

$$\mathbf{x}((k+1) \cdot h) = \mathbf{x}(k \cdot h) + h \cdot \dot{\mathbf{x}}(k \cdot h) \tag{D.4}$$

As the derivative is based on only one point on the curve and the input, the estimated value for $\mathbf{x}((k+1) \cdot h)$ is not very accurate which can only be overcome by the selection of a very small step size h (which increases the numerical errors beyond a certain limit).

Now we introduce our resukt into the state equation:

$$\mathbf{x}((k+1) \cdot h) = \mathbf{x}(k \cdot h) + h \cdot \mathbf{A} \cdot \mathbf{x}(k \cdot h) + h \cdot \mathbf{B} \cdot \mathbf{u}(k \cdot h) \tag{D.5}$$

We obtain for the Euler integration: $\boxed{\mathbf{x}((k+1) \cdot h) = \mathbf{x}(k \cdot h) + h \cdot I \cdot (\mathbf{A} \cdot \mathbf{x}(k \cdot h) + \mathbf{B} \cdot \mathbf{u}(k \cdot h))}$

This integration algorithm can be expressed in a block diagram (see Fig. D.2)

D.1.2 Numerical Intergration using the Heun Method

The drawbacks of Euler integration can be overcome by introducing more intermediate steps into the integration process. This makes it possible to increase the step size and to obtain more accurate results. The following derivation shows

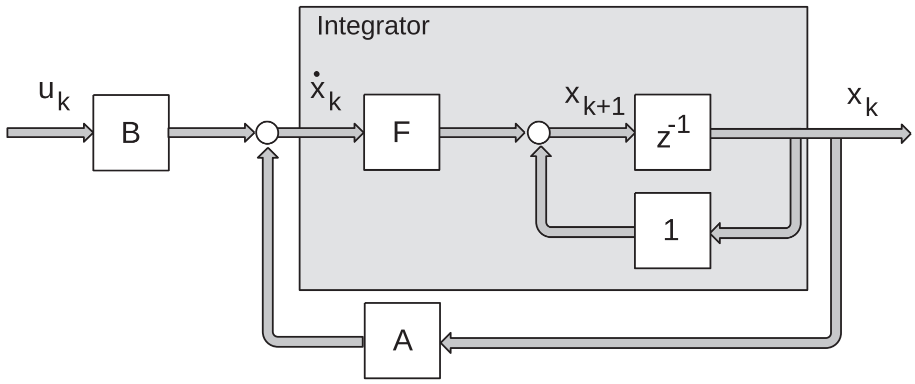


Figure D.2: Euler integration as a block diagram, $\mathbf{F} = h \cdot \mathbf{I}$

the derivation of the Heun method. The extension using one more point leads to RK-3. The procedure for the derivation is depicted in Fig. D.3. First, we calculate the estimated value for $\mathbf{x}(h \cdot (k+1))$; this value is enhanced in the next step - in the first step we perform an Euler integration in order to obtain the estimated value $\mathbf{x}(h \cdot (k+1))$. Using this estimated value $\mathbf{x}^p(h \cdot (k+1))$ we calculate the tangent slope is calculated by averaging the slopes in the point $\mathbf{x}(h \cdot k)$ (i.e. $\dot{\mathbf{x}}(h \cdot k)$) and in the estimated point $\mathbf{x}^p(h \cdot (k+1))$ (i.e. $\dot{\mathbf{x}}(h \cdot k)$).
Given is a state space system as follows:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{D.7}$$

We require the trajectory $\mathbf{x}(t)$. We sample the continuous trajectory using the step size h:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t)|_{t=k \cdot h} \tag{D.8}$$

We obtain:

$$\dot{\mathbf{x}}(k \cdot h) = \mathbf{A} \cdot \mathbf{x}(k \cdot h) + \mathbf{B} \cdot \mathbf{u}(k \cdot h) \tag{D.9}$$

Now use the method given in Fig. D.4 and calculate the estimated value $\mathbf{x}^p((k+1) \cdot h)$ using the step size h and the derivative $\dot{\mathbf{x}}(k \cdot h)$:

$$\mathbf{x}^p((k+1) \cdot h) = \mathbf{x}(k \cdot h) + h \cdot \dot{\mathbf{x}}(k \cdot h) \tag{D.10}$$

The derivative $\mathbf{x}^p((k+1) \cdot h)$ is calculated using the state equation:

$$\dot{\mathbf{x}}^p((k+1) \cdot h) = \mathbf{A} \cdot \mathbf{x}^p((k+1) \cdot h) + \mathbf{B} \cdot \mathbf{u}((k+1) \cdot h) \tag{D.11}$$

Now we rewrite:

$$\mathbf{x}((k+1) \cdot h) = \mathbf{x}(k \cdot h) + \frac{h}{2} \cdot [\dot{\mathbf{x}}(k \cdot h) + \dot{\mathbf{x}}^p((k+1) \cdot h)] \tag{D.12}$$

Finally, we introduce the expressions into the original simulation equation:

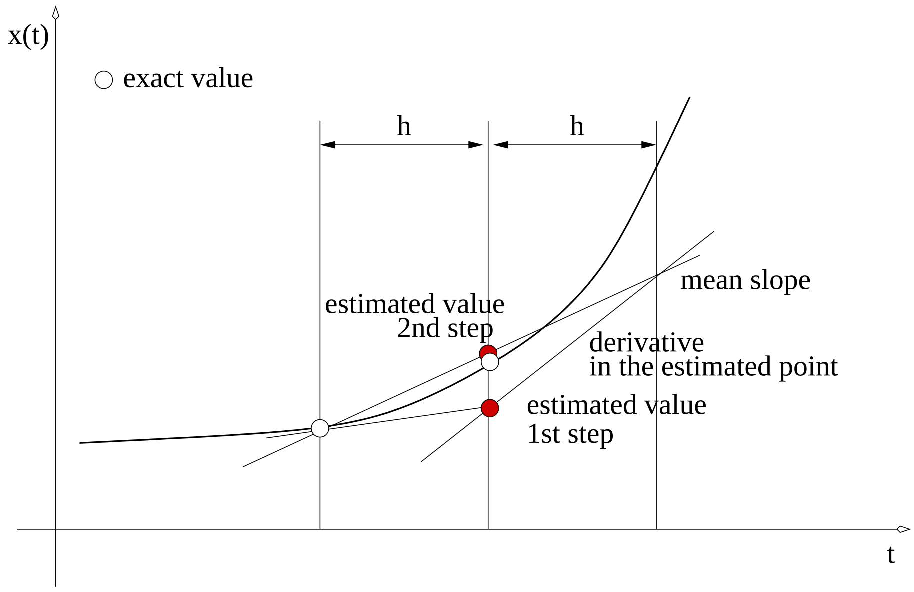


Figure D.3: Heun integration als an extended Euler method

$$
\begin{aligned}
\mathbf{x}((k+1)\cdot h) &= \mathbf{x}(k\cdot h) + \frac{h}{2}\cdot[\mathbf{A}\cdot\mathbf{x}(k\cdot h) + \mathbf{B}\cdot\mathbf{u}(k\cdot h) \\
&\quad + \mathbf{A}\cdot\mathbf{x}^p((k+1)\cdot h) + \mathbf{B}\cdot\mathbf{u}((k+1)\cdot h)]
\end{aligned} \tag{D.13}
$$

The expression can be further expanded:

$$
\begin{aligned}
\mathbf{x}((k+1)\cdot h) &= \mathbf{x}(k\cdot h) + \frac{h}{2}\cdot[\mathbf{A}\cdot\mathbf{x}(k\cdot h) + \mathbf{B}\cdot\mathbf{u}(k\cdot h) \\
&\quad + \mathbf{A}\cdot\mathbf{x}(k\cdot h) + \mathbf{A}h\cdot\dot{\mathbf{x}}(k\cdot h) + \mathbf{B}\cdot\mathbf{u}((k+1)\cdot h)]
\end{aligned} \tag{D.14}
$$

Finally we obtain:

$$
\boxed{\mathbf{x}((k+1)\cdot h) = \left(\mathbf{I} + \mathbf{A}h + \frac{\mathbf{A}^2h^2}{2}\right)\cdot\mathbf{x}(k\cdot h) + \left(\frac{h}{2}\mathbf{B} + \frac{\mathbf{AB}h^2}{2}\right)\cdot\mathbf{u}(k\cdot h) + \frac{h}{2}\mathbf{Bu}((k+1)h)}
$$

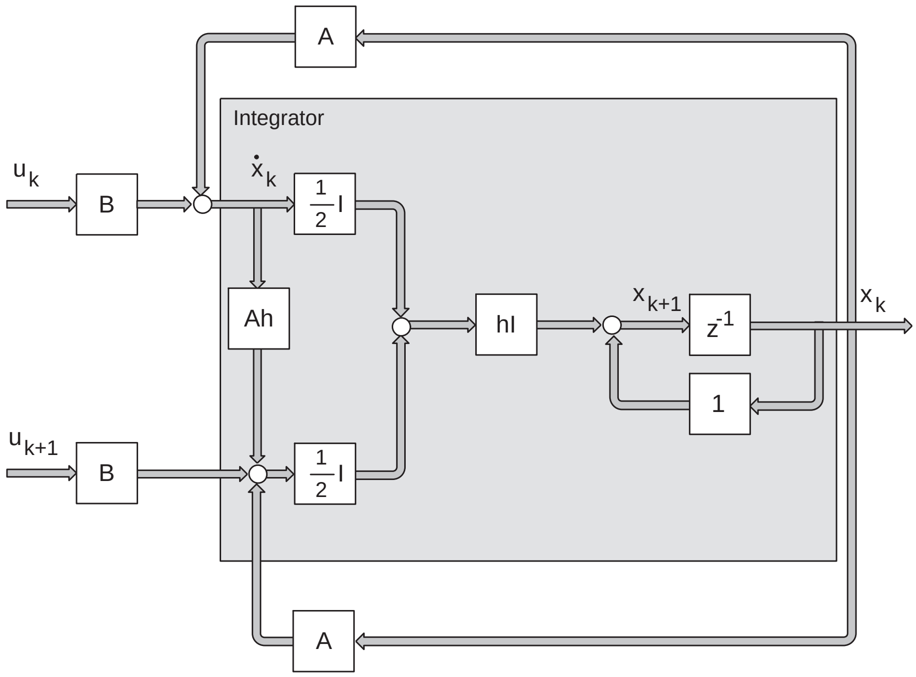


Figure D.4: Block diagram describing the Heun integration algorithm

D.2 Discretization of continuous time control algorithms

In order to convert continuous time control algorithms into their discrete time equivalents, different methods are applicable. In the following sections, we apply direct discretization using simple numerical euler integration and backward differentiation algorithms.

D.2.1 Digital PI- Controller

The digital PI algorithm is derived outgoing from the continuous time description:

$$u(t) = K_p \cdot x_d(t) + K_i \cdot \int_0^t x_d(t)dt \tag{D.16}$$

The integral is approximated using the rectangular rule (Euler integration). Thus we obtain:

$$u(k) = K_p \cdot x_d(k) + K_i \cdot \sum_{n=0}^{k} x_d(n) \cdot T_a \tag{D.17}$$

To obtain an applicable and implementable difference equation, one has to shift the whole equation :

$$u(k+1) = K_p \cdot x_d(k+1) + K_i \cdot \sum_{n=0}^{k} x_d(n) \cdot T_a + x_d(k+1) \cdot T_a \qquad \text{(D.18)}$$

The unshifted version of the difference equation is now reorganized as follows:

$$u(k) - K_p \cdot x_d(k) = +K_i \cdot \sum_{n=0}^{k} x_d(n) \cdot T_a \qquad \text{(D.19)}$$

and then inserted into the shifted equation. This yields:

$$u(k+1) = K_p \cdot x_d(k+1) - K_p \cdot x_d(k) + u(k) + K_i \cdot x_d(k+1) \cdot T_a \qquad \text{(D.20)}$$

Sorting the variables and performing $\mathcal{Z}$-transform yields:

$$\begin{aligned} u(k+1) - u(k) &= K_p \cdot (x_d(k+1) - x_d(k)) + K_i \cdot x_d(k+1) \cdot T_a \\ U(z) \cdot (z-1) &= K_p \cdot X_d(z)(z-1) + X_d(z) \cdot z \cdot T_a \cdot K_i \qquad \text{(D.21)} \end{aligned}$$

The transfer function of the controller is then given by:

$$\boxed{G_R(z) = \frac{U(z)}{X_d(z)} = K_p + K_i \cdot \frac{z}{z-1} \cdot T_a}$$

D.2.2 Digital PD Controller

The derivation of the digital PD controller is similar to the previously shown derivation of the PI controller. The control algorithm is in the continuous time domain given by:

$$u(t) = K_p \cdot x_d(t) + K_d \cdot \dot{x}_d(t) \qquad \text{(D.23)}$$

The differentiation with respect to time is replaced by the backward difference quotient:

$$u(k) = K_p \cdot x_d(k) + K_d \cdot \frac{x_d(k) - x_d(k-1)}{T_a} \qquad \text{(D.24)}$$

The transfer function of the controller is then again derived using $\mathcal{Z}$-transform:

$$U(z) = K_p \cdot X_d(z) + K_d \cdot X_d(z) \cdot \frac{1 - z^{-1}}{T_a} \qquad \text{(D.25)}$$

$$\boxed{G_R(z) = \frac{U(z)}{X_d(z)} = K_p + K_d \frac{z-1}{z} \cdot \frac{1}{T_a}}$$

D.2.3 Digital PID Algorithm

In the continuous time domain, the PID control algorithm is given by:

$$u(t) = K_p \cdot x_d(t) + K_d \cdot \dot{x}_d(t) + K_i \cdot \int\limits_0^t x_d(t)dt \tag{D.27}$$

again we approximate this differential equation using backward difference quotients and Euler integration.

$$u(k) = K_p \cdot x_d(k) + K_d \cdot \frac{x_d(k) - x_d(k-1)}{T_a} + K_i \cdot \sum_{n=0}^{k} x_d(k) \cdot T_a \tag{D.28}$$

We again shift the difference equation one step ahead:

$$u(k+1) = K_p \cdot x_d(k+1) + K_d \cdot \frac{x_d(k+1) - x_d(k)}{T_a} + K_i \cdot \sum_{n=0}^{k} x_d(k) \cdot T_a + K_i \cdot T_a \cdot x_d(k+1) \tag{D.29}$$

Now the equation for $u(k)$ is solved for the sum:

$$u(k) - K_p \cdot x_d(k) - K_d \cdot \frac{x_d(k) - x_d(k-1)}{T_a} = K_i \cdot \sum_{n=0}^{k} x_d(k) \cdot T_a \tag{D.30}$$

the result is then introduced in the equation for $u(k+1)$:

$$\begin{aligned} u(k+1) \quad = \quad & K_p \cdot x_d(k+1) + K_d \cdot \frac{x_d(k+1) - x_d(k)}{T_a} + u(k) - K_p \cdot x_d(k) - \\ & K_d \cdot \frac{x_d(k) - x_d(k-1)}{T_a} + K_i \cdot T_a \cdot x_d(k+1) \end{aligned} \tag{D.31}$$

und vereinfacht ergibt sich:

$$\begin{aligned} u(k+1) \quad = \quad & K_p \cdot (x_d(k+1) - x_d(k)) + K_d \cdot \frac{x_d(k+1) - 2x_d(k) + x_d(k-1)}{T_a} + \\ & u(k) + K_i \cdot T_a \cdot x_d(k+1) \end{aligned} \tag{D.32}$$

$\mathcal{Z}$-Transform yields the transfer function:

$$\begin{aligned} u(k+1) - u(k) &= K_p \cdot (x_d(k+1) - x_d(k)) + K_d \cdot \frac{x_d(k+1) - 2x_d(k) + x_d(k-1)}{T_a} + \\ & K_i \cdot T_a \cdot x_d(k+1) \end{aligned} \tag{D.33}$$

we obtain:

$$U(z)\cdot(z-1) = K_p \cdot X_d(z)\cdot(z-1) + \frac{K_d}{T_a}\cdot X_d(z)\cdot(z-2+z^{-1}) + K_i \cdot T_a \cdot z \cdot X_d(z) \tag{D.34}$$

Finally we get:

$$G_R(z) = \frac{U(z)}{X_d(z)} = K_p \cdot X_d(z) + \frac{K_d}{T_a} \cdot X_d(z) \cdot \frac{z-2+z^{-1}}{z-1} + K_i \cdot T_a \cdot \frac{z}{z-1} \cdot X_d(z) \tag{D.35}$$

The same result can be obtained easier if a linear combination of P, D and I parts is formed.

$$\boxed{\begin{aligned} G_R(z) &= K_p \cdot X_d(z) + \frac{K_d}{T_a} \cdot X_d(z) \cdot \frac{z^2-2z+1}{z(z-1)} + K_i \cdot T_a \cdot \frac{z}{z-1} \cdot X_d(z) \\ &= K_p \cdot X_d(z) + \frac{K_d}{T_a} \cdot X_d(z) \cdot \frac{z-1}{z} + K_i \cdot T_a \cdot \frac{z}{z-1} \cdot X_d(z) \end{aligned}}$$

D.2.4 Tustin-Transformation

The Tustin transform is another method to approximate continuous time systems in the z-domain. The tustin approximation relates a trapezoid integration algorithm with the term $\frac{1}{s}$ in the Laplace domain:

$$y(k) = \sum_{n=0}^{k} \frac{T_a}{2} \cdot (x(k+1) + x(k)) \tag{D.36}$$

The summation of trapezoid areas yield the value of the integral:

$$y(k+1) = \sum_{n=0}^{k} \frac{T_a}{2} \cdot (x(k+1) + x(k)) + \frac{T_a}{2} \cdot (x(k+1) + x(k)) \tag{D.37}$$

Again, equation (D.36) is shifted which yields equation (D.37). Then the unshifted equation is inserted into the shifted equation:

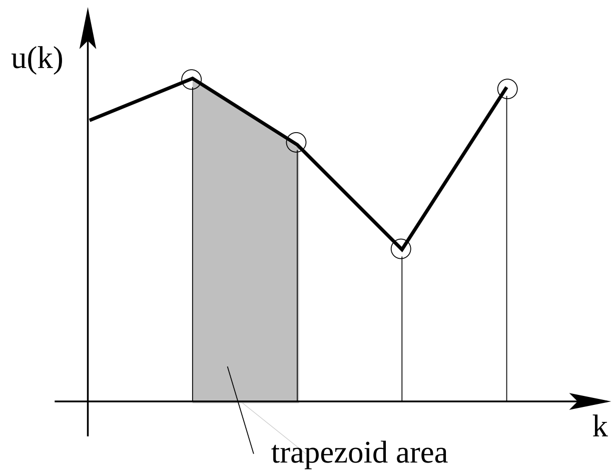


Figure D.5: Tustin tranformation

$$y(k+1) = y(k) + \frac{T_a}{2}\left(x(k+1) + x(k)\right) \tag{D.38}$$

Now we apply $\mathcal{Z}$-transform:

$$z \cdot Y(z) = Y(z) + \frac{T_a}{2}\left(zX(z) + X(z)\right) \tag{D.39}$$

We obtain:

$$\boxed{\frac{Y(z)}{X(z)} = G(z) = \frac{T_a}{2} \cdot \frac{z+1}{z-1}}$$

now we compare $\frac{1}{s}$ with our result:

$$\frac{1}{s} = \frac{T_a}{2} \cdot \frac{z+1}{z-1} \tag{D.41}$$

D.3 Sampled data systems in state space

The continuous time state equations can be expressed in the sampled data domain by assuming a special hold device (zero order hold). This assumption holds for most of the realizations of A/D and D/A converters in sampling or reconstruction devices. The derivation is based on a linear continuous time time invariant state space system. First we have to solve the state equations:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\cdot\mathbf{x}+\mathbf{B}\cdot\mathbf{u} && \text{(D.42)}\\ \mathbf{y} &= \mathbf{C}\cdot\mathbf{x}+\mathbf{D}\cdot\mathbf{u} && \text{(D.43)} \end{aligned}$$

Using a series expansion of $\mathbf{x}(t)$, we get:

$$\mathbf{x}(t) = \mathbf{b}_0 + \mathbf{b}_1\cdot t + \mathbf{b}_2\cdot t^2 + \cdots \tag{D.44}$$

Derivation with respect to time yields:

$$\dot{\mathbf{x}}(t) = \mathbf{b}_1 + 2\cdot\mathbf{b}_2\cdot t + 3\cdot\mathbf{b}_3\cdot t^2 + \cdots \tag{D.45}$$

These equations are inserted into the homogenious system equation:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\cdot\mathbf{x} && \text{(D.46)}\\ \mathbf{b}_1 + 2\cdot\mathbf{b}_2\cdot t + 3\cdot\mathbf{b}_3\cdot t^2 + \cdots &= \mathbf{b}_0 + \mathbf{b}_1\cdot t + \mathbf{b}_2\cdot t^2 + \cdots && \text{(D.47)} \end{aligned}$$

Equating coefficients yields:

$$\begin{aligned} \mathbf{b}_1 &= \mathbf{A}\cdot\mathbf{b_0} && \text{(D.48)}\\ 2\cdot\mathbf{b}_2 &= \mathbf{A}\cdot\mathbf{b_1} && \text{(D.49)}\\ \mathbf{b}_2 &= \frac{1}{2}\cdot\mathbf{A}\cdot\mathbf{b_1} && \text{(D.50)}\\ &= \frac{1}{2}\cdot\mathbf{A}^2\cdot\mathbf{b_0} && \text{(D.51)}\\ 3\cdot\mathbf{b}_3 &= \mathbf{A}\cdot\mathbf{b_2} && \text{(D.52)}\\ \mathbf{b}_3 &= \frac{1}{3}\cdot\mathbf{A}\cdot\mathbf{b_2} && \text{(D.53)}\\ &= \frac{1}{6}\cdot\mathbf{A}^3\cdot\mathbf{b_0} && \text{(D.54)}\\ &\vdots && \text{(D.55)} \end{aligned}$$

We then obtain the matrix e-function

$$\mathbf{x}(t) = \left(\sum_{k=0}^{\infty}\frac{(\mathbf{A}\cdot t)^k}{k!}\right)\cdot\mathbf{b}_0 \tag{D.56}$$

f"ur $t = 0$ bleibt nur:

$$\mathbf{x}(0) = \mathbf{b}_0 \tag{D.57}$$

The solution for the homogenious differential equation is given by:

$$\boxed{\begin{aligned} \mathbf{x}(t) &= \left(\sum_{k=0}^{\infty} \frac{(\mathbf{A}\cdot t)^k}{k!}\right)\cdot \mathbf{x}(0) \\ \mathbf{x}(t) &= e^{\mathbf{A}\cdot t}\cdot \mathbf{x}(0) \end{aligned}}$$

In order to determine the inhomogenious solution, we reformulate the equation

$$\dot{\mathbf{x}} = \mathbf{A}\cdot\mathbf{x} + \mathbf{B}\cdot\mathbf{u} \tag{D.58}$$

$$\dot{\mathbf{x}} - \mathbf{A}\cdot\mathbf{x} = \mathbf{B}\cdot\mathbf{u} \tag{D.59}$$

Now we recognize, that

$$\frac{d}{dt}\left(e^{-\mathbf{A}\cdot t}\cdot\mathbf{x}\right) = -e^{-\mathbf{A}\cdot t}\cdot\mathbf{A}\cdot\mathbf{x} + e^{-\mathbf{A}\cdot t}\cdot\dot{\mathbf{x}} \tag{D.60}$$

by pre multiplying of

$$e^{-\mathbf{A}\cdot t} \tag{D.61}$$

we obtain

$$e^{-\mathbf{A}\cdot t}\left(\dot{\mathbf{x}} - \mathbf{A}\cdot\mathbf{x}\right) = e^{-\mathbf{A}\cdot t}\cdot\mathbf{B}\cdot\mathbf{u} \tag{D.62}$$

Which can be rewritten as:

$$\frac{d}{dt}\left(e^{-\mathbf{A}\cdot t}\cdot\mathbf{x}\right) = e^{-\mathbf{A}\cdot t}\cdot\mathbf{B}\cdot\mathbf{u} \tag{D.63}$$

Integration of this equation yields:

$$\left[e^{-\mathbf{A}\cdot\tau}\cdot\mathbf{x}\right]_0^t = \int_0^t e^{-\mathbf{A}\cdot\tau}\cdot\mathbf{B}\cdot\mathbf{u}d\tau \tag{D.64}$$

an by solving further:

$$e^{-\mathbf{A}\cdot t}\cdot\mathbf{x}(t) - \mathbf{x}(0) = \int_0^t e^{-\mathbf{A}\cdot\tau}\cdot\mathbf{B}\cdot\mathbf{u}d\tau \tag{D.65}$$

The equation is now multiplied with $e^{\mathbf{A}\cdot t}$ and we obtain the result:

$$\mathbf{x}(t) = e^{\mathbf{A}\cdot t} \cdot \mathbf{x}(0) + \int_0^t e^{-\mathbf{A}\cdot(t-\tau)} \cdot \mathbf{B} \cdot \mathbf{u} d\tau \tag{D.66}$$

Substituting $\mathbf{\Phi}(t) = e^{\mathbf{A}\cdot t}$:

$$\mathbf{x}(t) = \mathbf{\Phi}(t) \cdot \mathbf{x}(0) + \int_0^t \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u} d\tau \tag{D.67}$$

For linear time invariant systems we get further:

$$\mathbf{x}(t) = \mathbf{\Phi}(t-t_a) \cdot \mathbf{x}(t_a) + \int_0^t \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u} d\tau \tag{D.68}$$

We assume that the system input is the output of a D/A converter with zero order hold. Thus $\mathbf{u}(t)$ is assumed to be a staircase function:

$$\mathbf{u}(t) = \sum_{k=0}^{t/T_a} \mathbf{u}(k) \cdot (\sigma(t - k \cdot T_a) - \sigma(t-(k+1) \cdot T_a) \tag{D.69}$$

Now we take a closer look upon the reaction of the system for one step with step height $\mathbf{u}(k)$ at the point of time $k \cdot T_a$. Therefor we obtain:

$$\mathbf{x}(t) = \mathbf{\Phi}(t - k \cdot T_a) \cdot \mathbf{x}(k \cdot T_a) + \int_{k\cdot T_a}^t \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u}(\mathbf{k}) d\tau \tag{D.70}$$

Using the fact, that $\mathbf{u}(k)$ is not a function of t, one can take it out of the integral

$$\mathbf{x}(t) = \mathbf{\Phi}(t - k \cdot T_a) \cdot \mathbf{x}(k \cdot T_a) + \int_{k\cdot T_a}^t \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u}(\mathbf{k}) d\tau \tag{D.71}$$

First we calculate the integral

$$\int_{k \cdot T_a}^{t} \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u}(k) d\tau \tag{D.72}$$

substituting

$$\begin{aligned} \theta &= t - \tau && \text{(D.73)} \\ \frac{d\theta}{d\tau} &= -1 && \text{(D.74)} \\ d\theta &= -d\tau && \text{(D.75)} \end{aligned}$$

yields:

$$\int_{k \cdot T_a}^{t} \mathbf{\Phi}(t-\tau) \cdot \mathbf{B} \cdot \mathbf{u}(k) d\tau = -\int_{\theta(k \cdot T_a)}^{\theta(t)} \mathbf{\Phi}(\theta) \cdot \mathbf{B} \cdot \mathbf{u}(k) d\theta \tag{D.76}$$

Exchanging the limits changes the sign:

$$-\int_{\theta(k \cdot T_a)}^{\theta(t)} \mathbf{\Phi}(\theta) \cdot \mathbf{B} \cdot \mathbf{u}(\mathbf{k}) d\theta = \int_{\theta(t)}^{\theta(k \cdot T_a)} \mathbf{\Phi}(\theta) \cdot \mathbf{B} \cdot \mathbf{u}(k) d\theta \tag{D.77}$$

Now we introduce a Matrix $\mathbf{H}(t)$:

$$\mathbf{H}(t) = \int_{0}^{t} \mathbf{\Phi}(\theta) \cdot \mathbf{B} d\theta \tag{D.78}$$

Integration yields:

$$\mathbf{H}(t) = \left[\mathbf{A}^{-1} \cdot e^{\mathbf{A} \cdot t} \cdot \mathbf{B}\right]_0^t \tag{D.79}$$

This gives:

$$\mathbf{H}(t) = \mathbf{A}^{-1} \left[\cdot e^{\mathbf{A} \cdot t} - \mathbf{I}\right] \cdot \mathbf{B} \tag{D.80}$$

Inserting this result into the integrated state equation, we get:

$$\mathbf{x}(t) = \mathbf{\Phi}(t - k \cdot T_a) \cdot \mathbf{x}(k \cdot T_a) + \mathbf{H}(t - k \cdot T_a) \cdot \mathbf{u}(k) d\tau \tag{D.81}$$

In order to find out the response for a staircase function, we sample $\mathbf{x}(t)$:

$$\mathbf{x}(t)|_{t=\nu \cdot T_a} = \mathbf{\Phi}(t - k \cdot T_a)|_{t=\nu \cdot T_a} \cdot \mathbf{x}(k \cdot T_a) + \mathbf{H}(t - k \cdot T_a)|_{t=\nu \cdot T_a} \cdot \mathbf{u}(\mathbf{k}) d\tau \tag{D.82}$$

using this, we obtain:

$$\mathbf{x}(\nu \cdot T_a) = \mathbf{\Phi}((\nu - k) \cdot T_a) \cdot \mathbf{x}(k \cdot T_a) + \mathbf{H}((\nu - k) \cdot T_a) \cdot \mathbf{u}(k) \tag{D.83}$$

The system's answer for the staircase input is then obtained by:

$$\mathbf{x}((\nu+1) \cdot T_a) = [\mathbf{\Phi}(((\nu + 1) - k) \cdot T_a) \cdot \mathbf{x}(k \cdot T_a)]_{k=\nu} + [\mathbf{H}(((\nu + 1) - k) \cdot T_a) \cdot \mathbf{u}(\mathbf{k})]_{k=\nu} \tag{D.84}$$

This yields:

$$\mathbf{x}((\nu + 1) \cdot T_a) = \mathbf{\Phi}(T_a) \cdot \mathbf{x}(\nu) + \mathbf{H}(T_a) \cdot \mathbf{u}(\nu) \tag{D.85}$$

Finally, we obtain the state space representation of the sampled data system:

$$\boxed{\begin{aligned} \mathbf{x}(k + 1) &= \mathbf{\Phi}(T_A) \cdot \mathbf{x}(k) + \mathbf{H}(T_a) \cdot \mathbf{u}(k) && \text{(D.86)} \\ \mathbf{y}(k) &= \mathbf{C} \cdot \mathbf{x}(k) + \mathbf{D} \cdot \mathbf{u}(k) && \text{(D.87)} \end{aligned}}$$

Appendix E

Case Study: a development environment for image processing applications

In image processing we deal with a special signal processing setup, beginning with the image acquisition using e.g. a radiation sensor in a computer tomograph or a CCD sensor in a matrix camera for optical inspection. The next step is the denoising, the so called signal restoration which removes noise and unwanted channel distortion from the signal.
The signal is then segmented, analyzed and from the performed analysis, we can draw conclusions - e.g. in quality control whether a mechanical part is in the given tolerance. The whole process is depicted in Fig. E.1.

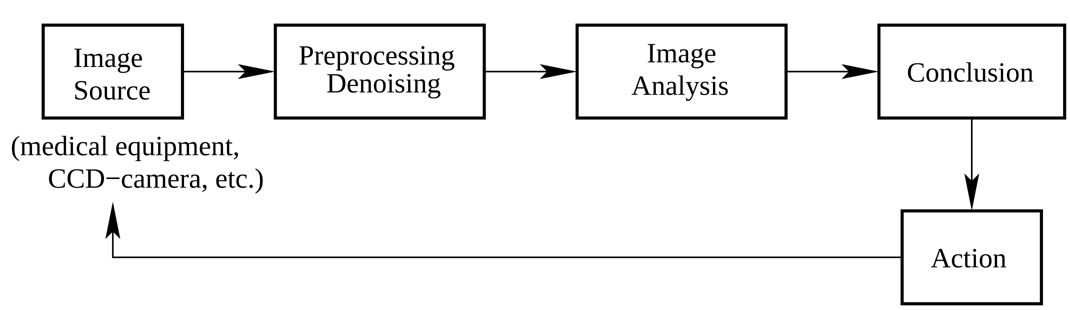


Figure E.1: The image processing pipeline

The image processing pipeline provides at the same time very specific and also very different operators. This makes it necessary to introduce a certain structure and also to be as open as possible in order to provide a platform for a wide range of operators. An intuitive user interface for configuration of the operators and their interconnection as given in Fig. E.2 has to be provided.

Although the complexity of the problem is quite higher than in our hard wired

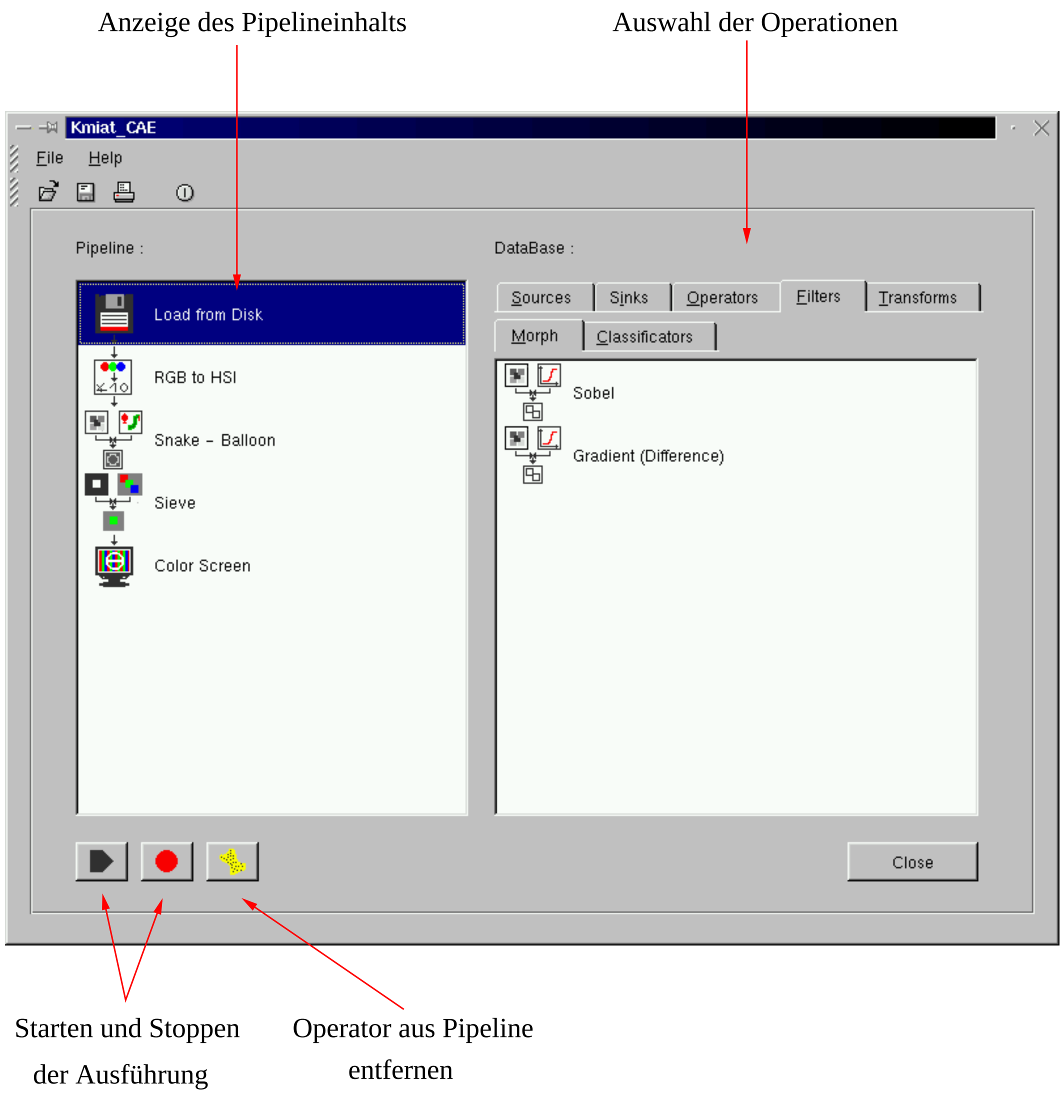


Figure E.2: The dynamic user interface for the application

control loop simulator - which is due to fully dynamic outputs - we find the same underlying structure. A basic operator as given in Fig. E.3 with a basic functionality is provided from which all other operators are derived.
The meta system is also similar to the one used in the control loop simulator - the dynamic list structure.

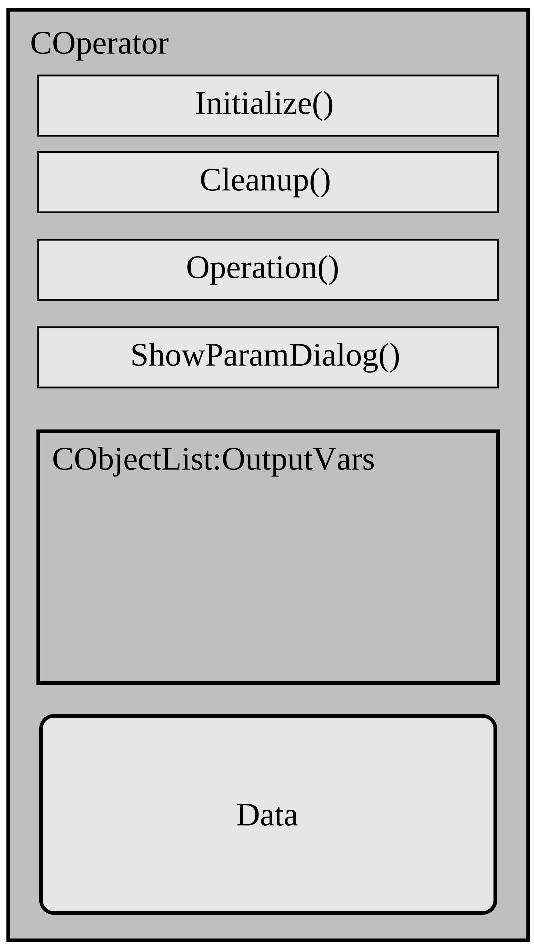


Figure E.3: Object oriented entity diagram of the basic operator

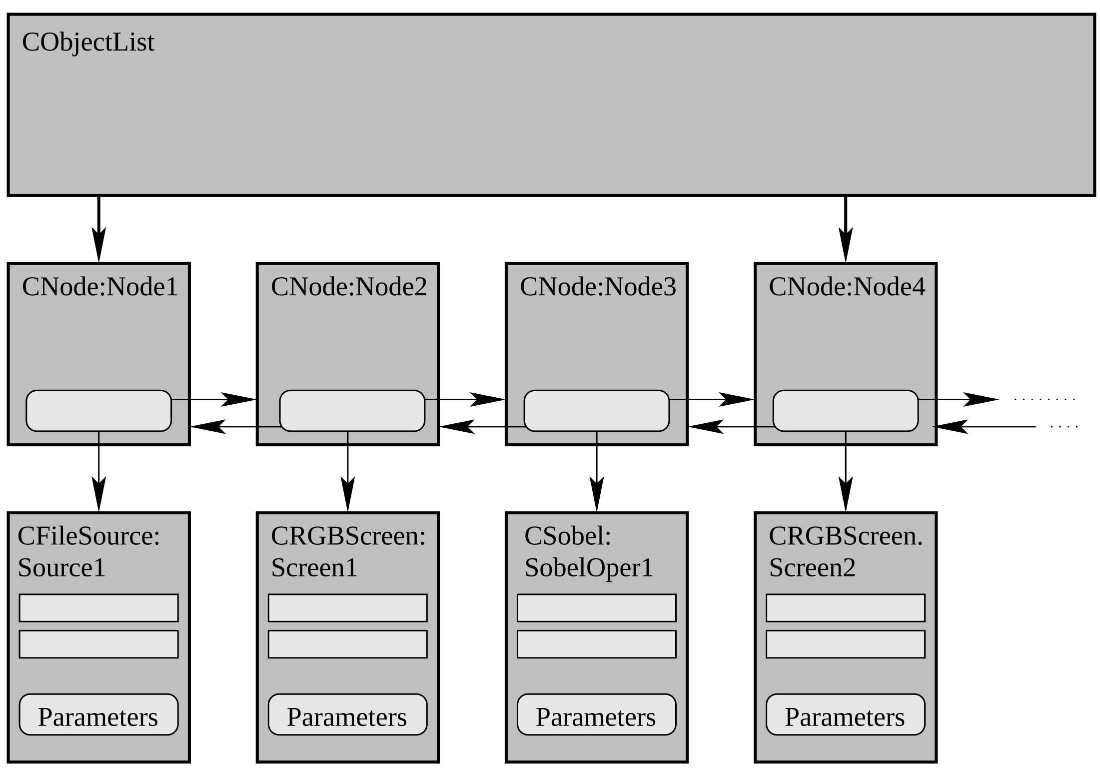


Figure E.4: Object oriented entity diagram of the meta system

Bibliography

[1] Helmut Balzert. *Handbuch der Softwaretechnik*, volume 1. Spektrum Akad. Verlag, Heidelberg, Germany, 1999.

[2] Helmut Balzert. *Handbuch der Softwaretechnik*, volume 2. Spektrum Akad. Verlag, Heidelberg, Germany, 1999.

[3] Helmut Balzert. *Lehrbuch Grundlagen der Informatik*, volume 2. Spektrum Akad. Verlag, Berlin, 1999.

[4] Otto Foellinger. *Lineare Abtastsysteme*. Oldenbourg Verlag, Munich, Germany, 5th edition, 1993.

[5] Heiko Hengen. Object oriented system analysis and synthesis for digital image processing systems, 1997. Studienarbeit.

[6] Heiko Hengen and Andreas Arnold. Verfahren der digitalen bildverarbeitung und deren einsatz in der praxis, 1998. Technical Report.

[7] Lothar Litz. Digitale prozess-steuerung, 1998. Lecture Notes.

[8] Lothar Litz. Modellbildung und identifikation, 1998. Lecture Notes.

[9] Lothar Litz. Process automation, 1999. Lecture Notes.

[10] Alan V. Oppenheim and Ronald W. Schafer. *Zeitdiskrete Signalverarbeitung*. Oldenbourg, Munich, Germany, 3rd edition, 1999.

[11] Madhukar Pandit. Abtastregelung, 1996. Lecture Notes.

[12] Madhukar Pandit. Control systems i, 2000. Lecture Notes.

[13] Madhukar Pandit. System theory, 2000. Lecture Notes.

[14] Eckehard Schnieder, editor. *Petrinetze in der Automatisierungstechnik*. Oldenbourg Verlag, München, Wien, 1992.

[15] Samuel D. Stearns and Don R. Hush. *Digitale Verarbeitung analoger Signale*. Oldenbourg, Munich, Germany, 7th edition, 1999.

[16] Christian Tuttas. Analoge und digitale simulation, 1998. Lecture Notes.

[17] Siegfried Wendt. *Grundlagen der Informationstechnik*. Springer Verlag, Berlin, 1st edition, 1990.