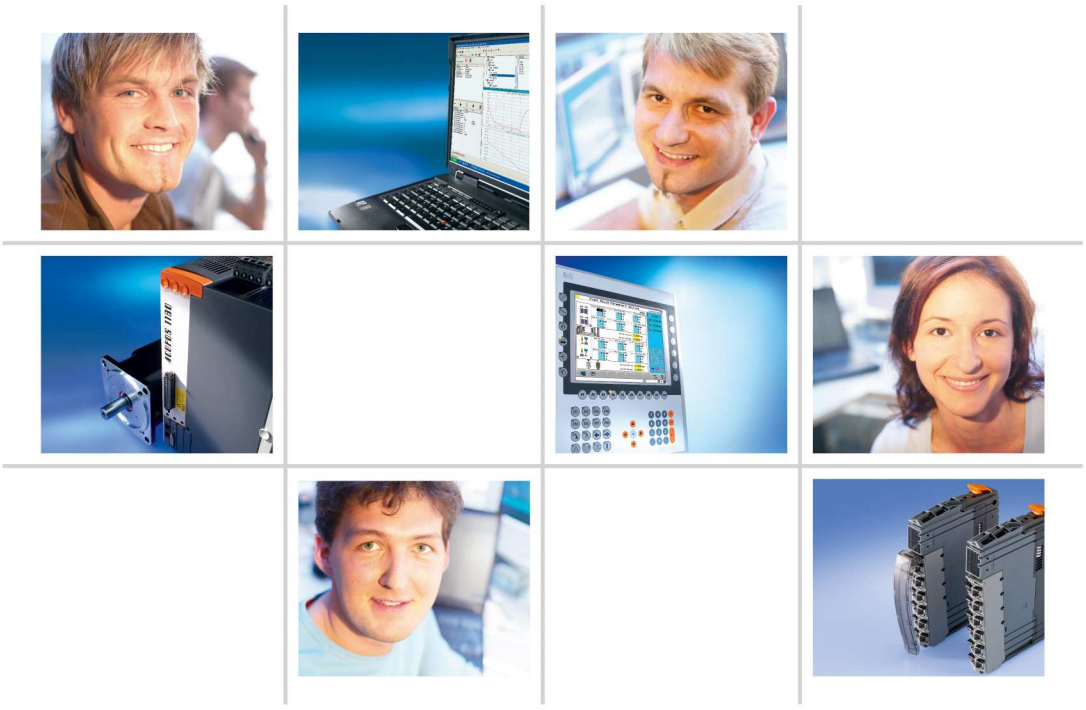
Structured Software Generation

TM230



Perfection in Automation

www.br-automation.com

Prerequisites

Training modules:	TM210 – The Basics of Automation Studio
	TM211 – Automation Studio Online Communication
	TM223 – Automation Studio Diagnostics
	At least one programming language.
Software:	Automation Studio
Hardware:	none

Table of Contents

1. INTRODUCTION

This training module is all about generating application software in the field of automation. If you are (or are going to be) a programmer of machines or plants, please ask yourself a few questions:

- Is software generation more than just coding, coding, coding?
- How can I improve the quality of the software I produce?
- By the way, what is software quality?
- What about costs to fix defects in software?
- How do I create well structured software?
- Is there a way to analyze, describe and discuss machine logic in a formal and exact way?
- How can I write better source code?
- How should I test and document the code I create?

Interested? Head on and dive into the following sections! HAVE FUN!

Fig. 1 Buggy code

1.1 Objectives

After successfully working through this module, the course participant will

- have been introduced to software engineering concepts, steps in software generation and software quality issues
- be familiar with the method of state diagrams and finite state machines to analyze, describe and discuss the logical function of machines in an exact and formal way
- be familiar with the B&R Coding Guidelines to produce, test and document high quality source code

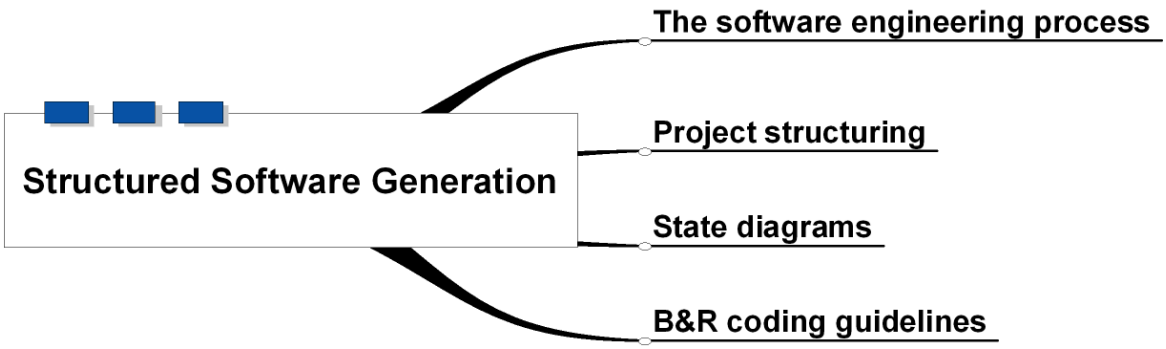


Fig. 2 Overview

2. THE SOFTWARE ENGINEERING PROCESS

What is software engineering anyway? Here are two definitions:

- Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation and maintenance of software; that is the application of engineering to software (IEEE Standard Computer Dictionary).
- Software engineering is a discipline whose aim is the production of high quality software, delivered on time, within budget and satisfying users' needs (S. R. Schach: Software Engineering).

2.1 Steps in Software Generation

Typical steps in software generation are:

- Requirements analysis
- Software specification
- Software design (or software architecture)
- Coding (implementation)
- Testing
- Documentation
- Maintenance

The first task in creating a desired software product is **analyzing** and extracting its **requirements**. It may require skill and experience to recognize incomplete, ambiguous or contradictory requirements.

The result should be a precise description of the software to be written - the **software specification**. Typically a specification is a written **agreement with the customer**.

The software **design** (software architecture) is developed **based** on the **specification**. It determines how the software is to function in a general way without being involved in details.

Software architecture may be defined as 'the internal conceptual design of software, enabling the software to exhibit a certain set of attributes'. It is wise to discuss the desired software architecture with the customer and get an agreement on it.

In the following **implementation** step the software design is coded in a specific programming language (specified program behavior is converted into operational code).

Subsequent steps are **testing**, **documentation** and **maintenance**.

Issues that must already be considered in the requirements analysis and software specification/design phases include:

- machine operating modes (such as manual, semi-automatic, setup, automatic, stand alone, linked, ...) and conditions for changing from one mode to another
- emergency and safety
- error handling
- providing diagnostic information
- remote access (via HMI, VNC, HTTP, etc.)

2.2 Software Quality

Important quality issues of application software in automation are:

- clean architecture and design
- conformance to requirements and specifications
- absence of bugs
- source code quality: the way a program is written can have some important consequences for the human maintainers, such as readability, logical structuring of the code into manageable sections low resources consumption (memory, CPU time)
- ease of maintenance

2.3 Cost of Fixing Defects

Building a software system is like any other project that takes people and money. People introduce errors and errors cost (additional) money.

In software engineering it pays to do things right the first time, because the expense of fixing defects rises dramatically as the time from when it is introduced to when it is detected increases (as shown in the picture, which is taken from S. McConnell: Code Complete).

Therefore the general principle is to find an error as close as possible to the time at which it was introduced.

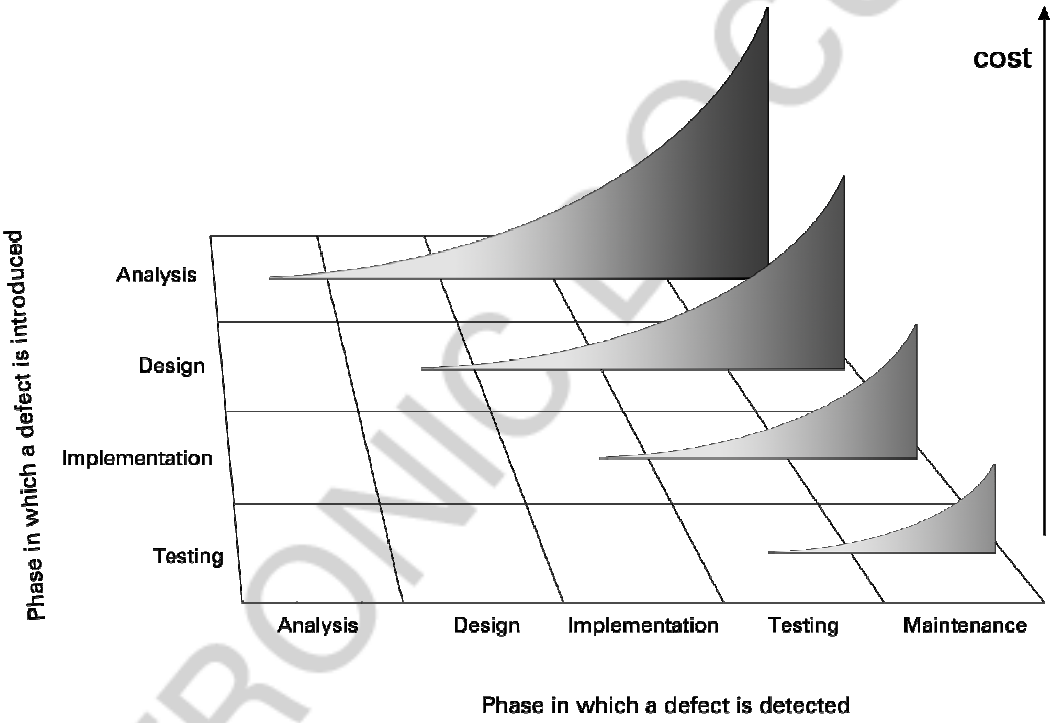


Fig. 3 Cost of fixing a defect as a function of when it is introduced and detected

3. PROJECT STRUCTURING

3.1 Structured Software Design

Clean architecture and design are the foundation of any good software.

The capabilities and limitations of all target platforms on which the software will be run must be taken into account.

Take your time in the conception phase and work on the design until you are satisfied with your concept. Usually the time invested in a good software design is well spent because it is saved many times later in the implementation phase.

Studies have shown that programmers who rush to coding generally take longer to finish their programs than programmers who plan first.

One approach to software design is the structured design approach. It is also known as top-down design, stepwise or successive refinement technique and decomposition approach. In this approach a problem is analyzed by repeatedly dividing it into smaller functional parts (modules), which are easier to handle (successive refinement technique).

In human history this problem solving strategy has been known for at least two thousand years ('divide et impera' = 'divide and conquer' = 'teile und herrsche' was the motto of ancient Roman emperor Marcus Aurelius).

The structured design approach is characterized by moving from a general statement of what the program does to detailed statements about specific tasks that are performed.

In a first step sub-systems are identified, the program is partitioned into major components and interfaces in between are defined. In the following decomposition steps more details are introduced successively.

An especially important issue of software design is the design of data and information flow, which defines:

- where data resides in your software modules
- data exchange between software modules
- how data is organized into data structures

In general, software design is not a deterministic process but requires creativity and is often an iterative process.

Software design with the structured design approach can be summarized as follows:

- Design top level first
- Postpone details to lower levels
- Formalize each level
- Verify each level (if it complies with the specification)
- Move on to the next level
- Stop decomposing, when it is easier to code the next level than to decompose it. The design should seem obvious and easy then.

3.2 Case Study: Injection Molding Machine

Let's have a look now at an injection molding machine. The top level of the software design is the machine in total. Every detail is postponed to lower levels.

Fig. 4 Software architecture at top level

At level 2, the machine is decomposed into its basic functional (in this case mechanical) parts. Injection Unit, Hydraulic Clamp Cylinder, Mold, Transport, Feeder, Dosage and a General module, which is responsible for the basic logical function of the machine.

The next pciture shows the level 2 software architecture including data dependencies and data interfaces.

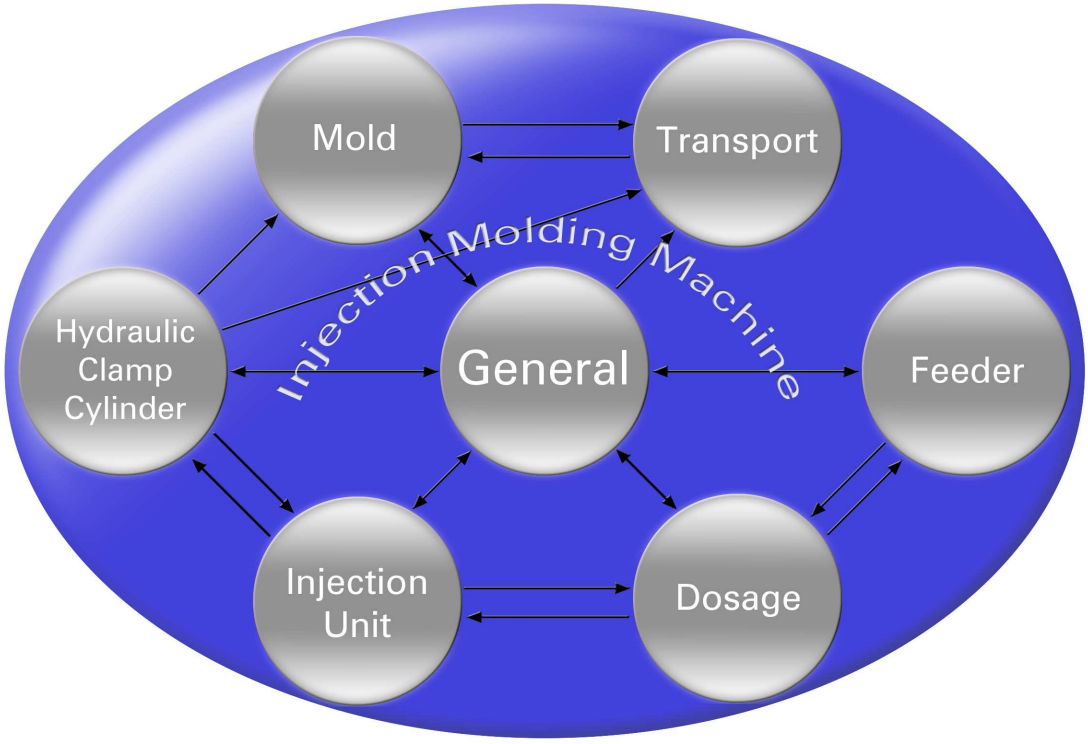


Fig. 5 Level 2 software architecture

At level 3 the basic functional parts are further refined: the functional part `Transport` is decomposed into `Scales` and `Conveyor belt`.

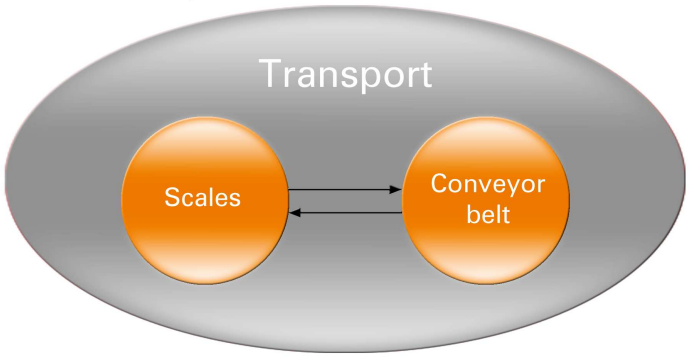


Fig. 6 Decomposition of functional part Transport

The `Scales` module performs weight checking of the product and signals the `Conveyor belt` module what to do with the product (further processing if product weight is ok or throw away otherwise). `Conveyor belt` gives feedback if it is ready to process the next piece.

The functional part `Mold` is decomposed into `Hydraulic ejector`, `Cores` and `Mold Heating`. `Hydraulic ejector` and `Cores` must not be active at the same time. Both have no functional connection to `Heating`, which exchanges data with the `General` module.

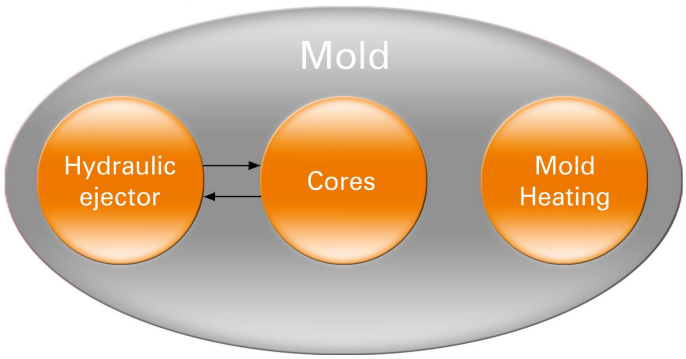


Fig. 7 Decomposition of functional part Mold

Injection Unit is decomposed into Injection Piston and Barrel Heating. There is no data connection and dependency between Injection Piston and Barrel Heating, these modules get their commands from General.

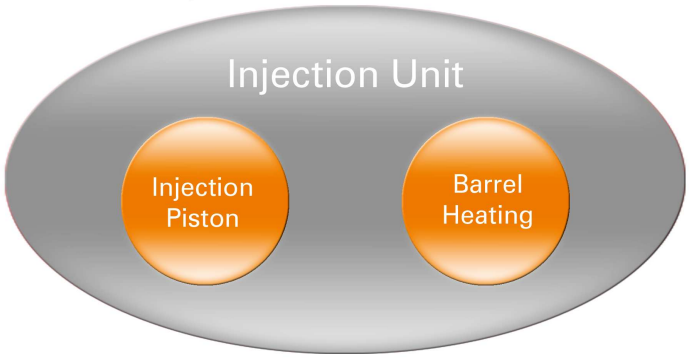


Fig. 8 Decomposition of functional part Injection Unit

The next picture outlines the data dependency and connection between the modules General, Mold and Transport.

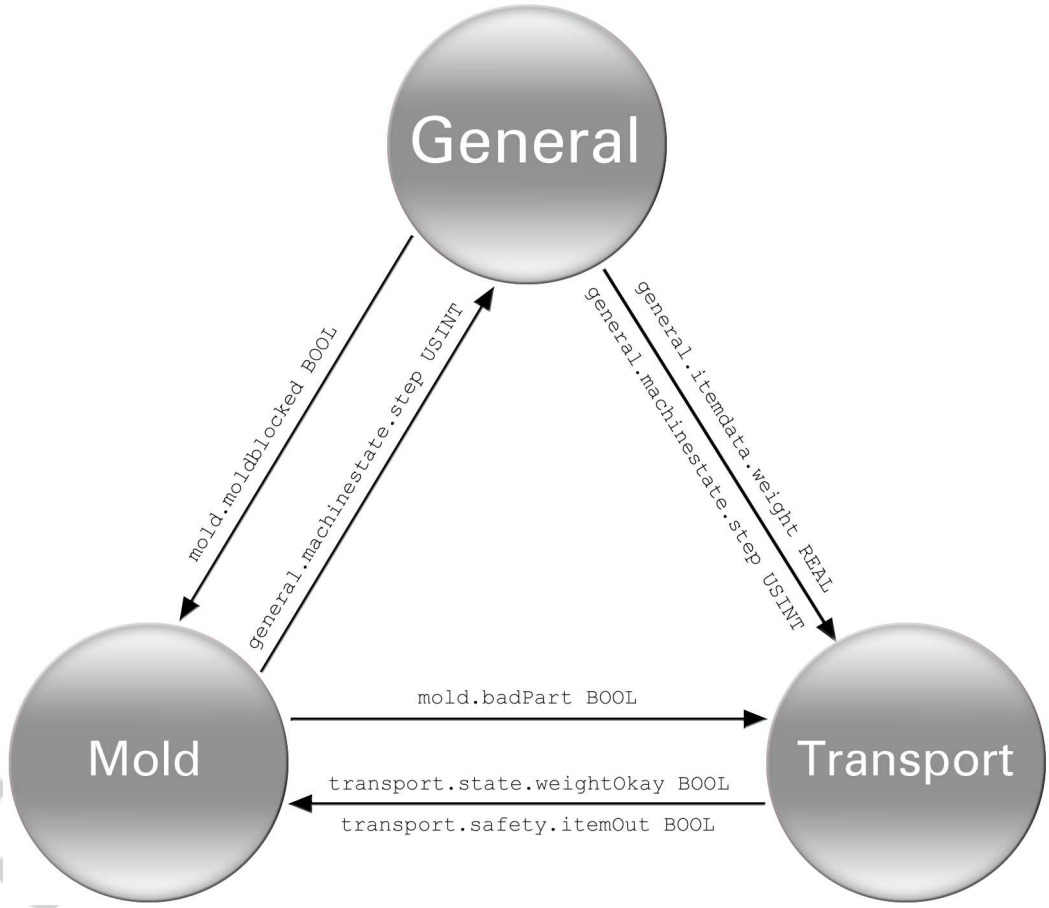


Fig. 9 Inter module data dependencies

4. STATE DIAGRAMS

One of the most common tasks for software engineers in the field of automation is the programming of machine or plant logic, which determines the general function of a machine or plant.

Usually the machine or plant logic is specified by a textual description, which is often inaccurate. With the help of state diagrams you can describe machine or plant logic in an exact and formalized way. State diagrams are also a convenient way of discussing machine logic with customers on a formal level.

Once you have developed a state diagram for your desired logic, the implementation (coding) is a straight forward and easy task. This section should familiarize you with state diagrams as a powerful method to analyze and describe machine sequences and plant procedures.

In this section all code fragments are given in the ANSI C programming language. For those who are not familiar with C, here is a short table with a description of operators in ANSI C:

`!`	`logical negation operator`
`=`	`assignment operator`
`==`	`comparison operator`
`&&`	`logical AND operator`
`++`	`increment operator`

4.1 Types of Logic

4.1.1 Combinatorial Logic

Let's have a look now at a water tank:

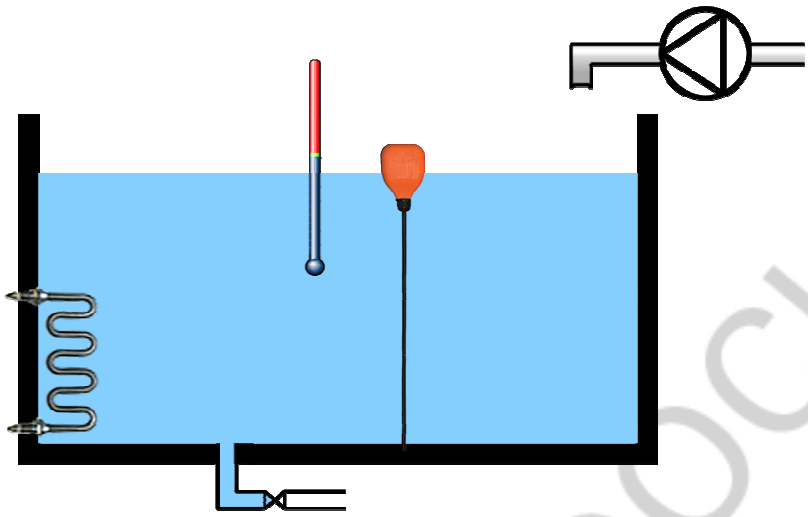


Fig. 10 Water tank temperature control

water is taken from the tank if an external valve is opened. The water level in the tank is held constant with a pump, which is switched on if the water level falls below a certain level, detected by a level switch:

```
pumpOn = (!levelSwitch)
```

Furthermore, the water should be conditioned to temperature `Tset` with a heating element. The actual water temperature `Tact` is measured with a temperature sensor. The heating element is switched on, if the water level in the tank is ok and if the actual water temperature is below the set temperature:

```
heaterOn = ((levelSwitch == 1) && (Tact < Tset))
```

In this simple example we have three inputs,

- set temperature `Tset` (of data type `INT`)
- actual temperature `Tact` (`INT`)
- level switch senor `levelSwitch` (`BOOL`)

and two outputs:

- pumpOn (BOOL)
- heaterOn (BOOL)

The outputs are a function of, and only of, the present inputs. There is a static relation between inputs and outputs. Such type of logic is called a combinatorial logic. It does not have memory (or storage).

Because of this static input to output relation there is no inherent system dynamic with possible associated stability problems (e.g. deadlocks) in a combinatorial logic.

Because of this missing ability to remember what happened in the past, combinatorial logic is too simple for the desired functionality of common machines or plants.

This is a severe limitation and some drawbacks of this approach can be seen easily even in our simple example:

- If the water level falls below the level switch, our simple logic will switch on the pump. The water level will increase causing the logic to switch off the pump. The result is a continuous switching with negative effects on the lifetime of the pump.

- The same thing happens with the heater: when the heater is on, the water temperature will increase causing the heater to shut down. Then the water will get cooler causing the heater to start and so on. The same effect may be caused by a flickering temperature sensor. Both will lead to a decreasing contactor lifetime.

In the next section we will make our simple logic a little more intelligent to overcome these disadvantages.

4.1.2 Sequential Logic

First, we do not switch off the pump immediately when `levelSwitch` changes to 1, but after a time delay of `tDelay` seconds.

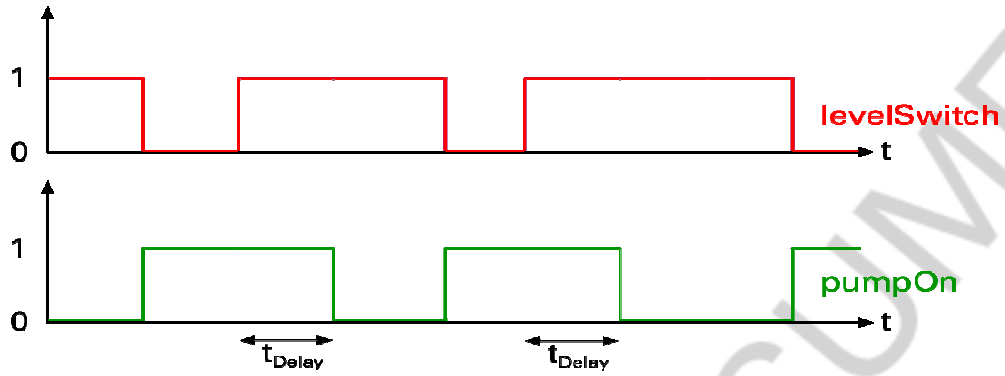


Fig. 11 Sequential water level control logic: time delay

Our second improvement is to add a hysteresis to the threshold value for switching off the heater: we do not switch off the heater when `Tact` reaches `Tset`, but when `Tact > (Tset+deltaT)`.

Now the outputs (`heaterOn` and `pumpOn`) are no longer determined by the present inputs:

- When `Tset < Tact < (Tset+deltaT)` it depends on the history of `Tact`, if the heater is on or off: The heater will now be on if less time has passed since the last occurrence of event '`Tact == Tset`' than time passed since the last occurrence of event '`Tact == (Tset+deltaT)`', otherwise the heater will be off.

- The pump can now be on, even if (`levelSwitch == 1`). Again in this case it depends on the history of `levelSwitch` if the pump is on or off. The pump will now be on if less than `tDelay` seconds have passed since the last occurrence of event '`levelSwitch` changes from 0 to 1'.

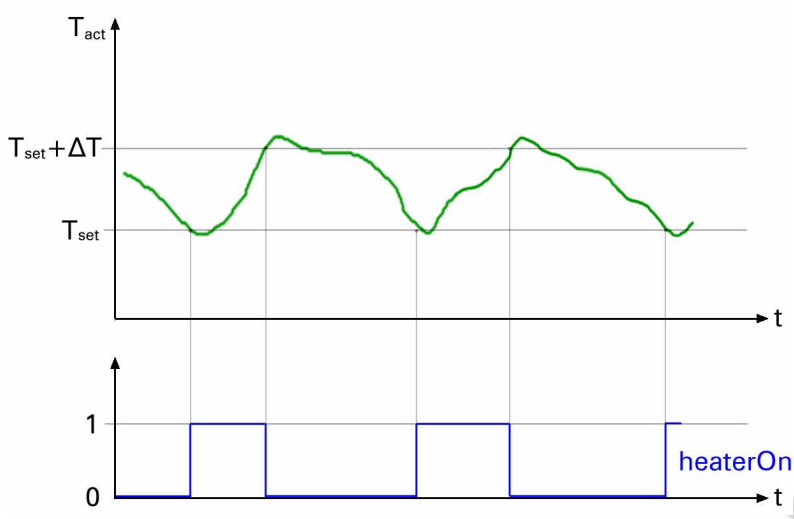


Fig. 12 Sequential water temperature control logic: hysteresis

The outputs of our improved logic depend not only on present inputs but also on past inputs. Such type of logic is called a sequential logic. Sequential logic is capable of storing and remembering information.

To describe the behavior of a sequential logic we need additional variables to represent the information the logic remembers. Such variables are called state variables because they uniquely define the state the logic is in.

Remark: In the last picture there are some overshoots (`Tact > (Tset+deltaT)`) and undershoots (`Tact < Tset`) in the temperature signal. This is a consequence of some dead time in the control loop.

4.2 Finite State Machines

The first step in the analysis of a sequential logic is to determine the number of state variables and all their possible values.

In our example we can choose the following state variables:

- `pumpState` with three possible values of:

```
PUMP_OFF
PUMP_ON_LEVELTOOLOW
PUMP_ON_TIMEDELAY
```

- `heaterState` with two possible values of:

```
HEATER_OFF
HEATER_ON
```

Our sequential logic has two state variables, where state variable 1 can have three possible values and state variable 2 can have two possible values.

A finite state machine is the description of a sequential logic with a finite number of states. A finite state machine consists of

- **States**: one state is the init or default state, in which the machine is, when turned on
- **Events**: they trigger transitions between states and must be prioritized, if more than one event can occur at the same time in a specific state (see following example)
- **Actions**: there are two types of actions:
  - actions, which are taken at state transitions (e.g. setting outputs)
  - actions, which are taken while being in a state (e.g. increasing counters)
- **Transitions** between states, which are triggered by events and cause actions ('if we are in state x1 and event y occurs take action z and transit to state x2')

In our example a transition between states can be triggered by the following events:

- `TooCold: (Tact < Tset)` changes from FALSE to TRUE
- `TooHot:(Tact > ((Tset+deltaT))` changes from FALSE to TRUE
- `LevelOk: levelSwitch` changes from FALSE to TRUE
- `LevelNotOk: levelSwitch` changes from TRUE to FALSE
- `TimerElapsed: tDelay` seconds have passed since last occurrence of `LevelOk`

We have to take the following actions:

- `SwitchPumpOn`
- `SwitchPumpOff`
- `SwitchHeaterOn`
- `SwitchHeaterOff`
- `ResetTimer`
- `IncrementTimer` (this action is not taken at a state transition, but is performed while being in state `TIMER_ON`)

4.2.1 State Transition Table

A state transition table is the description of transitions in a tabular form:

State	Event	Action	Transit to state
HEATER_OFF	TooCold	SwitchHeaterOn	HEATER_ON
HEATER_ON	TooHot	SwitchHeaterOff	HEATER_OFF

State	Event	Action	Transit to state
PUMP_OFF	LevelNotOk	SwitchPumpOn	PUMP_ON
PUMP_ON	LevelOk	ResetTimer	TIMER_ON
TIMER_ON	TimerElapsed	SwitchPumpOff	PUMP_OFF

4.2.2 State Diagrams

State diagrams are graphical representations of state machines.

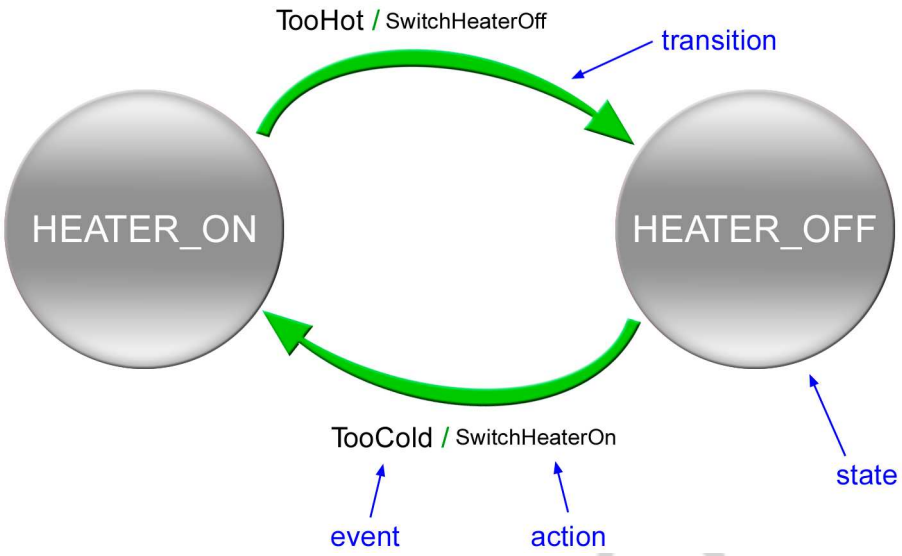


Fig. 13 State diagram for the heater

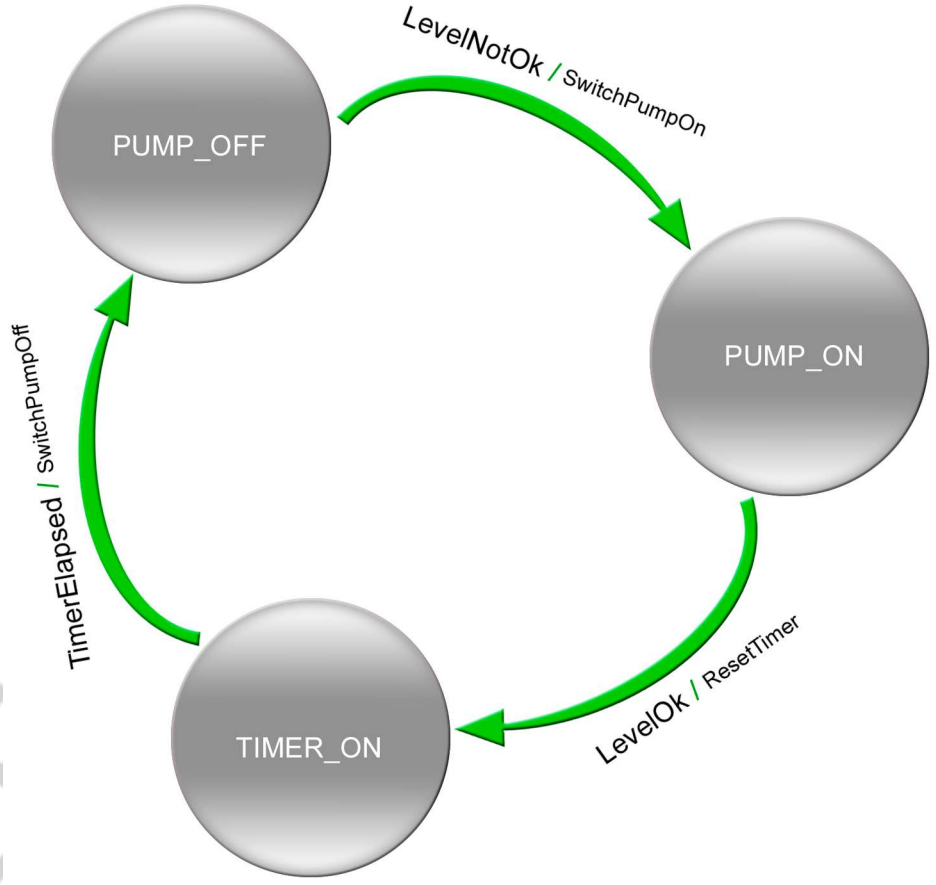


Fig. 14 State diagram for the pump

4.3 Implementation

If you have successfully described and formalized the sequential machine logic in a state diagram, implementation is an easy and straight forward task.

In pseudo code a simple implementation of a finite state machine could look like:

```
switch(state)
    for all states i
          case (state i)
                perform actions in state i
                for all events j
                if event j
                      then take action k
                      transit to state l
```

This is a pretty compact notation. A real implementation can get lengthy as our simple example demonstrates:

```
/********************************************************************
 * COPYRIGHT -- B&R Industrial Automation
 ********************************************************************
 * Program: TankControl
 * File: TankControl.c
 * Created: 04-March-2005
 ********************************************************************
 * Implementation of program TankControl
 ********************************************************************/

#include <bur\plctypes.h>

#ifdef _DEFAULT_INCLUDES
 #include <AsDefault.h>
#endif

#define PUMP_OFF   0
#define PUMP_ON    1
#define TIMER_ON   2

#define HEATER_OFF 0
#define HEATER_ON  1
```

```
/* ----------- events -------------- */

BOOL TooCold(void)
{
    return(Tact < Tset);
}

BOOL TooHot(void)
{
    return(Tact > (Tset + deltaT));
}

BOOL LevelOk(void)
{
    return(!levelSwitch);
}

BOOL LevelNotOk(void)
{
    return(levelSwitch);
}

BOOL TimerElapsed(void)
{
    return(timer == tDelay);
}

/* ---------- actions ---------- */

void SwitchPumpOn(void)
{
    pumpOn = 1;
}

void SwitchPumpOff(void)
{
    pumpOn = 0;
}

void SwitchHeaterOn(void)
{
    heaterOn = 1;
}

void SwitchHeaterOff(void)
{
    heaterOn = 0;
}

void ResetTimer(void)
{
    timer = 0;
}

void IncrementTimer(void)
{
    timer++;
}
```

```
/* ---------- transitions -------- */

void _CYCLIC WaterTankCYCLIC(void)
{

    /* ---------- pump ----------- */

    switch (pumpState)
    {
        case PUMP_OFF:
            if (LevelNotOk())
            {
                SwitchPumpOn();
                pumpState= PUMP_ON;
            }
            break;

        case PUMP_ON:
            if (LevelOk())
            {
                pumpState = TIMER_ON;
                 ResetTimer();
            }
            break;

        case TIMER_ON:
            IncrementTimer();
            if (TimerElapsed())
            {
                SwitchPumpOff();
                pumpState = PUMP_OFF;
            }
            break;

    }  /* end switch */

    /* ---------- heater ---------- */

    switch (heaterState)
    {
        case HEATER_OFF:
            if (TooCold())
            {
                SwitchHeaterOn();
                heaterState= HEATER_ON;
            }
            break;

        case HEATER_ON:
            if (TooHot())
            {
                SwitchHeaterOff();
                heaterState= HEATER_OFF;
            }
            break;

    }  /* end switch */

} /* end WaterTankCYCLIC */
```

4.4 Case Study: Sorting Material on a Conveyor Belt

Now let's take a look at a conveyor belt transporting pieces with different lengths. A control logic should sort out pieces, which length $L \leq A1$ or $L \geq A2$. The length is determined with photoelectric barrier sensors with digital input signals B1, B2 and B3. A signal is high if a piece passes through a barrier.

Sorting is done by setting the digital signal `lengthOk = 1` for one clock period which causes the pneumatic actuator to push a piece of correct length onto another conveyer belt. We do not have to worry about the actuator. It will return to its original position automatically.

Distances between the photoelectric barriers are A1 and A2, where $A1 > A2/2$. The distance between different pieces of material is much larger than A2.

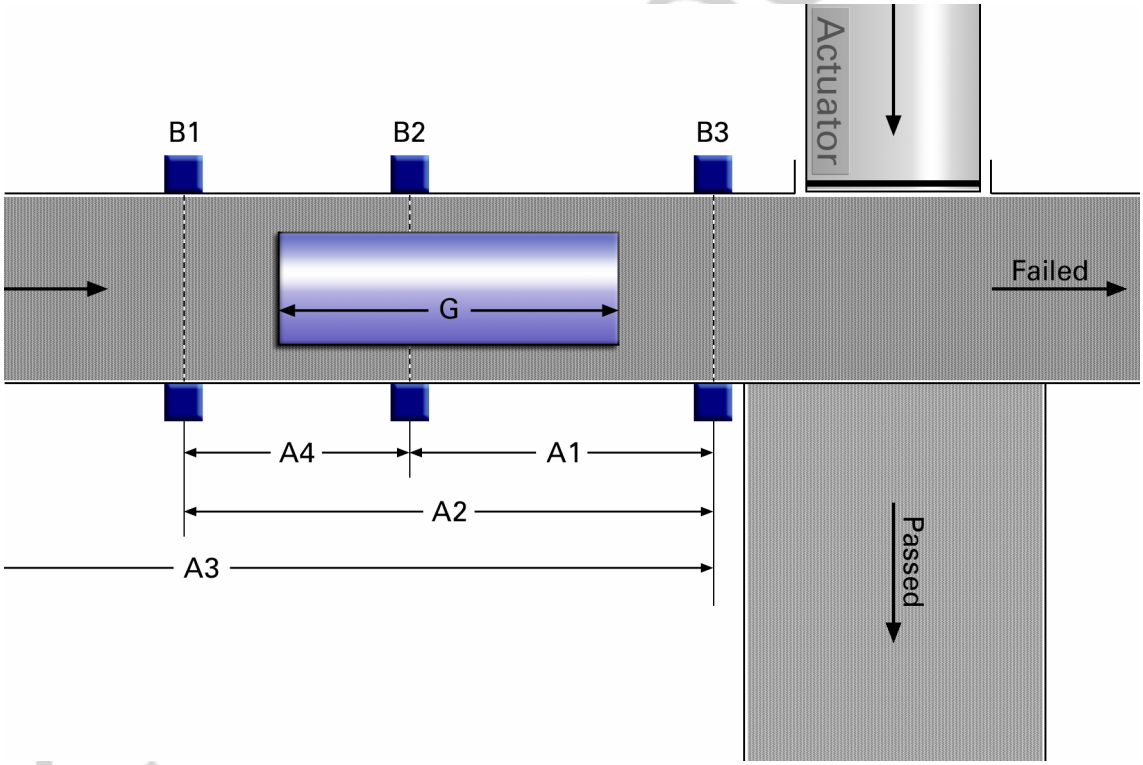


Fig. 15 Sorting arrangement

Our job is to design a control logic for the pneumatic actuator. In this example we focus on the process logic and explicitly neglect error handling (which is an important part in real life applications).

In the following, $(B_1B_2B_3)$ denotes the actual input signals of all photo-electric barriers, e.g. (010) means `B1 = 0`, `B2 = 1` and `B3 = 0`.

The input sequence for a piece with the correct length is:

`A1 < L < A2: (000),(100),(110),(010),(011),(001),(000)` → `lengthOk = 1`

Input sequences for incorrect lengths are:

```
L > A2:       (000),(100),(110),(111),(011),(001),(000)
L = A2:       (000),(100),(110),(011),(001),(000)
L = A1:       (000),(100),(110),(010),(001),(000)
A4 < L < A1: (000),(100),(110),(010),(000),(001),(000)
L = A4:       (000),(100),(010),(000),(001),(000)
L < A4:       (000),(100),(000),(010),(000),(001),(000)
```

Events:

- `Enter1`: B1 changes from 0 to 1 (piece enters photoelectric barrier 1)
- `Enter2`: B2 changes from 0 to 1 (piece enters photoelectric barrier 2)
- `Enter3`: B3 changes from 0 to 1 (piece enters photoelectric barrier 3)
- `Pass1`: B1 changes from 1 to 0 (piece leaves photoelectric barrier 1)
- `Pass2`: B2 changes from 1 to 0 (piece leaves photoelectric barrier 2)
- `Pass3`: B3 changes from 1 to 0 (piece leaves photoelectric barrier 3)

Actions:

- Push: set `lengthOk = 1`
- Reset: set `lengthOk = 0`

A solution with 13 states and 19 transitions is depicted in the state diagram in Fig. 23.

In the state transition table, events with highest priority are marked with (P). This means, that e.g. if we are in state `OK1` and the event `Pass1` happens, we will transit to state `TOOSHORT1` - **but** if `Pass1` **and** `Enter2` happen at the same clock cycle, we will transit to state TOOSHORT2.

state	event	action	transit to state
INIT	Enter1	-	OK1
OK1	Pass1 & Enter2 (P)	-	TOOSHORT2
OK1	Pass1	-	TOOSHORT1
OK1	Enter2	-	OK2
OK2	Pass1 & Enter3 (P)	-	TOOLONG2
OK2	Enter3	-	TOOLONG1
OK2	Pass1	-	OK3
OK3	Pass2 & Enter3 (P)	-	FAILED
OK3	Pass2	-	TOOSHORT3
OK3	Enter3	-	OK4
OK4	Pass2	-	OK5
OK5	Pass3	Push	FINALOK
FINALOK	-	Reset	INIT
TOOSHORT1	Enter2	-	TOOSHORT2
TOOSHORT2	Pass2	-	TOOSHORT3
TOOSHORT3	Enter3	-	FAILED
TOOLONG1	Pass1	-	TOOLONG2
TOOLONG2	Pass2	-	FAILED
FAILED	Pass3	-	INIT

Please note that actions are to be taken only in 2 out of 19 transitions. The transition from FINALOK to INIT need not be triggered by an event. This happens by default at the next clock cycle.

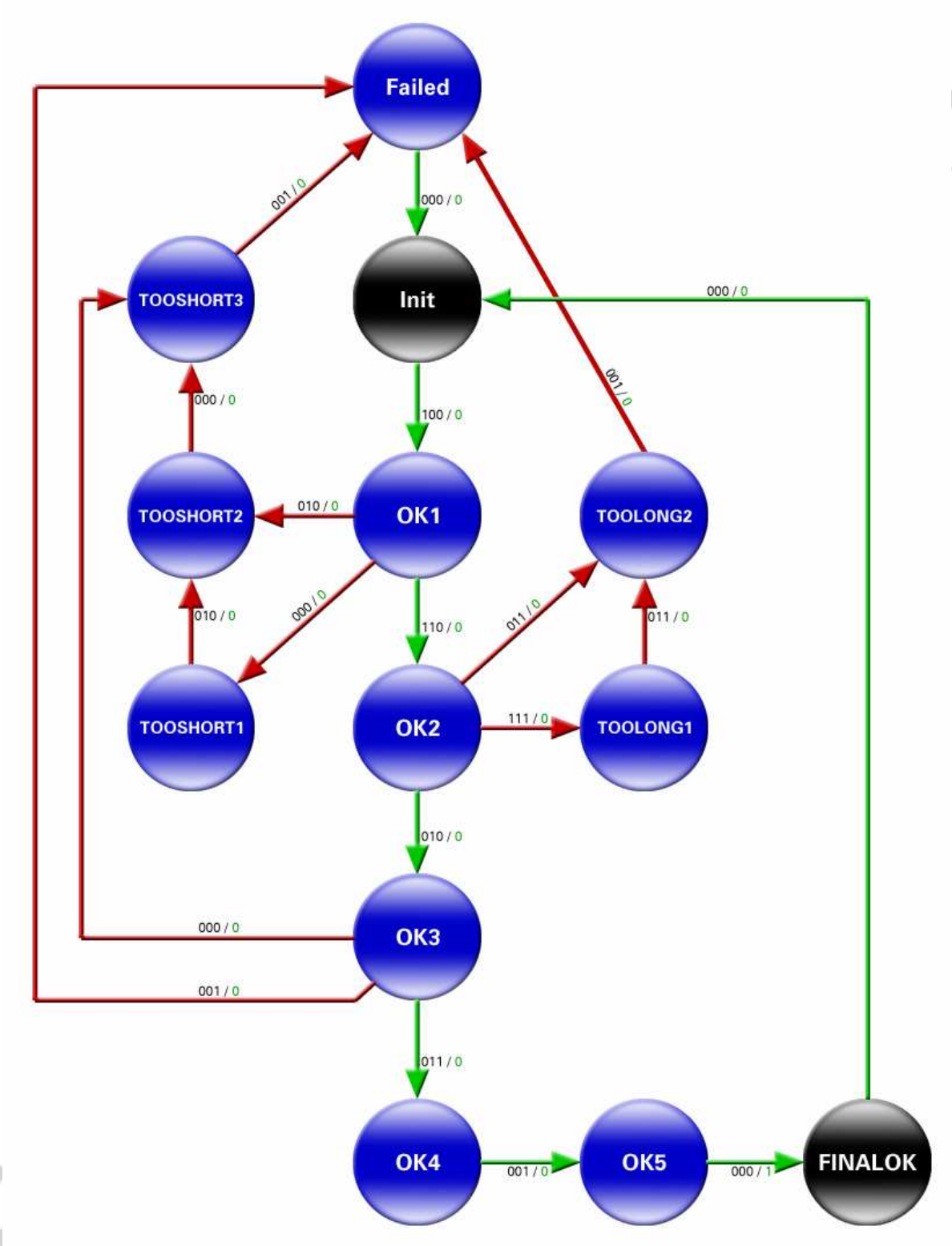


Fig. 16 State diagram for the conveyor belt logic

5. B&R CODING GUIDELINES

Computer programming is an engineering discipline (software engineering) and as usual in engineering there **is** an absolute truth ... whether a program **does** work or it **does not** work.

But computer programming also is an art (see the famous book 'The Art of Computer Programming' by Donald E. Knuth which has been named among the best twelve scientific monographs of the century) as sometimes it is more a question of aesthetics how a program does it's job and if the code looks appealing. Without question programming is a creative process.

Software production costs money - and - earns you money. It is the B&R philosophy to produce high quality products, and software is no exception here. So let's produce high quality software code!

Attributes of high quality code are (among others):

- clean architecture and design
- easy to read and understand
- easy to maintain
- re-usable
- well commented
- bug free

This document should assist you in improving your code quality. If you follow the guidelines outlined here your code should be of reasonable quality.

You are working in a team so please be considerate of your colleagues, who maybe won't appreciate dealing with those quick'n'dirty completely undocumented routines you hacked at 2:00 am Saturday night.

In the end **you** (the author) are responsible for the code you create. Do it well and then be proud of what you have created and achieved!

5.1 Before You Code

The foundations of any good software are a clean architecture and design. Take your time in the conception phase and work on the design until you are happy with your software concept.

Before you actually code please mind the concepts and methods discussed in sections of this document.

5.2 Naming Conventions

Good variable and data type names are a key element of program readability. All names should be descriptive and easy to read. Use either underscores or capital letters (don't mix them) in composite names to enhance readability, like

```
actPressure = actForce / pistonArea;
cmdCount++;
```

or

```
act_pressure = act_force / piston_area;
cmd_count++;
```

An identifier may contain letters and numbers and must start with a letter. You cannot use reserved keywords as identifiers. A complete list of reserved key words for each programming language can be found in the Automation Studio online help.

5.2.1 Language

If no different specification is given by the customer it is strongly recommended to code and comment in English for trouble-free international usage of software. Within B&R this recommendation is compulsory.

5.2.2 User Data Types (Structures)

User data types start with an upper case letter and end with the postfix '_type', in between lower and upper case letters may be mixed.

```
TYPE
    Recipe_type:  STRUCT
        base:     UINT;
        binder:   UINT;
        additive: USINT;
    END_STRUCT;
    MachineParams_type: STRUCT
        speed:       REAL;
        pressure:    REAL;
        temperature: INT;
        pRecipe:     REFERENCE TO Recipe_type;
    END_STRUCT;
END_TYPE
```

5.2.3 Constants

Constants are all upper case. Use underscores '_' to enhance readability.

```
VAR CONSTANT
    STEP_CONDITIONING: USINT := 23;       (* [-] *)
    HEATING_TIME_OUT:  UINT  := 5000;     (* [s] *)
    MAX_PRESSURE:      REAL  := 6.7e+006; (* [Pa]*)
END_VAR
```

These rules also apply to constants defined by the `#define` pre-processor directive and the enum statement in C source code. Please note that declarations via the `#define` directive and enum statement are local to the scope of your C code!

5.2.4 Local Variables

Local variables start with a lower case letter. Upper case-letters (or underscores '_') are only used to enhance readability.

```
VAR
    machineStep:    USINT; (* [-]     *)
    actPressure:    REAL;  (* [bar]   *)
    avgTemperature: INT;   (* [0.1°C] *)
END_VAR

VAR
    machine_step:    USINT; (* [-]     *)
    act_pressure:    REAL;  (* [bar]   *)
    avg_temperature: INT;   (* [0.1°C] *)
END_VAR
```

In the following, only examples without underscores are included. If you prefer naming with underscores you will be able to figure it out.

5.2.5 Global Variables

Global variables start with the pre-fix 'g' followed by an upper-case letter or '_':

```
VAR
    gHeaterOn: BOOL;
     gActCmd:   Cmd_typ;
     gCmdCount: UINT;
END_VAR
```

This convention is reserved for global variables - do not use it for non-global variables.

5.2.6 Pointers

Local pointers start with the pre-fix 'p' followed by an upper-case letter or '_'. Global pointers start with the pre-fix 'gp' followed by an upper-case letter or '_':

```
VAR
    pActRecipe: REFERENCE TO Recipe_type;
END_VAR

VAR
    gpActRecipe: UDINT;
END_VAR
```

The above convention is reserved for pointers - do not use it for other variables.

Global pointers have the data type 'UDINT' because IEC doesn't support pointers to generic data types. Cast the global pointer to a generic local pointer to access structure members:

```
pActRecipe = (Recipe_type*)gpActRecipe;
actSpeed   = pActRecipe->speed;
```

5.2.7 Hardware-Connected Variables

Variables assigned to hardware I/O points start with a pre-fix defining the I/O point type:

Prefix	Type
di	digital input
do	digital output
ai	analog input
ao	analog output

The pre-fix is followed by an upper-case letter or '_':

```
VAR
    diEmergencyOff: BOOL;
    doSolidStateOn: BOOL;
    aiActTemp:      INT;  (* [0.1°C]                  *)
    aoValvePos:     INT;  (* 0=closed, 32767=open     *)
END_VAR
```

This convention is reserved for HW connected variables - do not use it for other variables.

5.2.8 C-local Variables

Variables defined in C source code are not visible outside their definition scope – you cannot see them e.g. in a Watch or Trace window.

```
Recipe_type* pPrevRecipe = 0;
USINT        cmdCount    = 0;
REAL         actSpeed    = 0; /* [m/s] */
```

If not declared ‘static’ they are allocated from stack each time the C routine is executed (and therefore non-remanent) and are not initialized! It is therefore wise to initialize them in the declaration (as done above).

5.2.9 Instances of Function Blocks

Instances of function blocks should be named to contain the name of the function block:

```
VAR
    valveSwitchTON: TON_type;
    solidStateTOF:  TOF_type;
    pressureLCPID:  LCPID_type;
END_VAR
```

5.3 Code Format

Visual layout of the code should accurately represent the logical structure of a computer program. Thus visual information acquisition of the human brain can support the reader in code understanding.

5.3.1 Indentation

Proper indentation is a key element for the readability of a code and is a **must** in all programs!

The whole idea behind indentation is to clearly visualize where a block of control starts and ends.

A large indentation size (6 or 8 characters) makes the code structure easier to see, while a smaller indentation size (2 or 4 characters) saves space on the right hand side of your screen.

We suggest an indentation size of 4 characters. If you have good reasons choose another indentation size that you prefer and stick to it.

5.3.2 File Header

Every file must have a header, which includes:

- Information about author and copyright
- Short description (summary comments) with a focus on purpose of the code, input and output variables, global effects of the routine, limitations and interface assumptions
- Timing behavior and memory requirements (if critical)
- Revision number, history and date (in an international unmistakable format, e.g. 04-March-2005 instead of 04-03-05 or 03/04/05)

A template header is automatically included when you create a program in Automation Studio™.

Revision number format is Vxx.yy, where xx is incremented with every major code update (e.g. when new features are added or incompatibilities to the previous version are introduced) and yy is incremented with minor improvements and bug fixes.

5.3.3 Placing Braces

There are a lot of brace placement strategies around. The preferred method is putting each brace on a line by itself combined with proper indentation:

```
if (inst.request > 0)
{
    inst.ok2jump = 1;
    inst.status  = 0;
}
else
{
    inst.ok2jump = 0;
    inst.status  = 5;
}

UINT CheckStatus(REAL xDeviation, REAL yDeviation)
{
    function body
}
```

If this doesn't look visually appealing to you, choose another consistent style, e.g. as suggested by Kernighan and Ritchie:

```
if (inst.request > 0) {
    inst.ok2jump = 1;
    inst.status  = 0;
}
else {
    inst.ok2jump = 0;
    inst.status  = 5;
}
```

5.3.4 Spaces

For readability reasons add a space before and after each operator:

```
xAxisPos = x0 + deltaX;
if (machineState == STATE_RUN)
   ...
```

with exception of:

.	member selection operator
->	member selection operator
[]	subscription operator
()	function call and function declaration operator
(type)	unary casting operator
++	pre- and post increment operator
--	pre- and post decrement operator
!	unary negation operator
~	unary one's complement

If the assignment operator '=' is placed directly behind the variable, a search (or search and replace) in the editor for e.g. 'someVariable=' will only find assignments to this variable in the code. If this is important for you, format assignments this way:

```
xAxisPos= x0 + deltaX;
```

We recommend placing the reference '&' and dereference operators '*' near the type in declarations:

```
void GetCtrlParams(REAL deadTime, REAL dXmax, Params_type*
pCtrlParams)
```

5.3.5 Visual Alignment

Visual alignment of elements that belong together reinforces the visual binding of these elements:

```
stPar.X0     = pIntern->X0;
stPar.deltaX = stPar.dir * abs(inst.options.deltaX);
stPar.t1set  = 0; /* [ms] */
stPar.t2set  = 0;
```

5.4 Programming Techniques

5.4.1 GOTO statement

You should **not** use the GOTO statement because it will prevent you from clearly structuring your code.

Should you feel tempted to include a GOTO into your code think of Edsger W. Dijkstra's famous classic paper 'Go To Statement Considered Harmful' published in 1968 (http://www.acm.org/classics/oct95/):

"For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of GOTO statements in the programs they produce."

We have nothing to add to Dijkstra.

5.4.2 Usage of Standard Algorithms

If you need to include a standard algorithm (e.g. for ring buffers, sorting, searching, etc.) don't implement it yourself. Most likely your implementation will not be bug free without some time invested in testing and debugging.

The better way is to copy it in electronic form from a trusted source (e.g. CDs that come with standard text books).

5.4.3 Usage of IEC Data Types

For consistency and target independent code, use the IEC data types in C source code. They are automatically defined with the following statement:

```
#include <bur\plctypes.h>
```

5.4.4 Handling Hardware-Connected (I/0) Variables

It is a good idea to copy (and scale or negate if desired) hardware-connected variables to/from data structures of your software modules in a special task which does just that and nothing else.

You can easily disable this task and disconnect all I/Os for testing purposes. As another benefit you will only have to make minor changes at one single place in the code if external sensor or actor logic changes (which happens quite frequently).

5.4.5 Dynamic Memory Management

Please take care when using dynamically allocated memory. Access to memory which you have not properly allocated leads to errors which are **really hard** to discover!

Don't allocate and free memory frequently in cyclic code because it will lead to memory fragmentation. As a consequence the system will sometimes run out of memory.

5.4.6 Communication between Software Modules

Inter-module communication has to be implemented with global data structures. Therefore it needs to be designed with special care. Please mind appropriate naming of your communication data structures.

5.4.7 Compiler Warnings

Compiler warnings may indicate some unexpected program behavior. Be sure to understand the warning message and correct your code to avoid the warning. If this is not possible or not intended, document the warning at the corresponding line of code.

5.4.8 Determining Array Size

If you need to determine the size of an array (e.g. if you need to check the last array element or need to loop over all elements) use the sizeof function:

```
for (i = 0; i < (sizeof(array)/sizeof(array[0])); i++)
{
  loop body
}
```

5.4.9 Data Alignment

When defining user-defined data types you should note data alignment: in general the compiler has to add empty storage (typically 1 – 3 bytes) between structure members to place (or 'align') the members to specific (e.g. even) memory addresses for memory access.

The actual compiled data size is then larger than the sum of individual member sizes. You can easily check the compiled size with the sizeof function. It may be different for different target hardware architectures.

If you have to write platform-independent code take data alignment into consideration (especially when using data modules). You may place unused alignment bytes into your data structure by yourself to force identical data layout on all your target hardware:

```
TYPE
    Cutter_type: STRUCT
        speed:     REAL;     (* 4 bytes                *)
        cmdcount:  USINT;    (* 1 byte                 *)
        reserve1:  USINT;    (* alignment              *)
        xPosition: UINT;     (* 2 bytes                *)
        yPosition: UINT;     (* 2 bytes                *)
        reserve2:  USINT;    (* alignment              *)
        reserve3:  USINT;    (* alignment              *)
        cutterTON: TON_type; (* align like 4 byte type *)
    END_STRUCT;
```

Please see the Automation Studio online help for details about compiler data alignment.

5.5 Testing

Software testing is a crucial issue for software quality issue. It ensures that the behavior of the code is compliant to the specifications.

Usually the first task in testing is the definition of test cases on the basis of software specification. Testing of special situations and functionalities (special and corner cases) requires special care, e.g. what happens if an incorrect value is passed to a function (‘An effective way to test code is to exercise it at its natural boundaries.’ – Brian Kernighan, one of the creators of the C language).

If you are working in a project team consider testing your code mutually (the code you create is tested by one of your colleagues as an independent tester and vice versa).

Automation Studio™ provides excellent features for software testing: watching, tracing and forcing of variables. These diagnostic methods are extensively discussed in B&R Training Module ‘Automation Studio Diagnostics’.

5.5.1 Unit Testing

The goal of unit testing is to show that isolated individual parts (libraries, modules, functions, ...) are correct.

5.5.2 Integration Testing

In integration testing, individual software modules are combined and tested as a group to verify if they properly work together.

5.5.3 System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements (IEEE Standard Computer Dictionary). System testing is typically performed at machine or plant commissioning.

5.5.4 Usability Testing

Usability testing measures how well people can handle the machine or plant you have programmed. Usability testing typically focuses on the HMI.

5.6 Documentation

Documenting the software you have created is an important task. On the one hand it supports the users in working with all the functionalities you have provided for him and on the other hand it provides valuable information for other programmers who have to fix a bug or implement some additional functionality into your code.

Documentation on a software project typically consists of information both inside the source-code listings (the code itself and 'comments') and outside them (typically in the form of separate documents).

5.6.1 Comments

Documentation at code level is always aimed at other developers and not at users.

The main contribution to code-level documentation isn't comments, but good programming style: good code is its own **best** documentation.

However, in every program some comments are necessary to explain things about the code that the code can't say about itself (e.g. high-level and low-level organization of programs).

Types of comments:

- Summary comments: should give an overview and a summary of the program at the beginning of the code (like a preface)
- Intent comments (comments on the code's intent): should explain the purpose of a section of code and operate more at the level of the problem than at the level of the solution (explaining the **why** more than the **how**).
- Marker comments: should mark locations where you suspect a bug may exist or where code improvements are planned. They are useful in the development phase and should not appear in completed code.

What you should document:

- Data types (structures)
- Variable declarations (including physical units if applicable)
- Major steps of your routines
- Limitations of your routines
- Global effects of routines
- Interface assumptions
- Timing issues and memory requirements (if critical)
- Revision history

What not to comment:

- Do not use comments to explain things that are obvious to programmers!
- If your code is too difficult to be understood by others rewrite it!

Remember to keep comments up to date when changing the code!

5.6.2 External Documentation

There are two types of external documentation:

- User documentation: contains all information relevant to users of the software (HMI pages, alarms, errors, etc.)
- Developer documentation: contains information for software programmers (description of software design, flow charts, interfaces, etc.)

5.6.3 Documentation Standards

The American National Standards Institute (ANSI) provides ANSI/ANS 10.3-1995 standard for documentation of engineering and scientific computer software at their website http://www.ansi.org for purchase.

The military standard MIL-STD-498 defines software development and documentation standards and is is approved for use by all departments and agencies of the department of defense of the USA (http://www.pogner.demon.co.uk/mil_498/).

Both standards do not focus on industrial automation application software but may provide some valuable general information.

This is version V1.40 [19/07/05] of the B&R Coding Guidelines.

6. SUMMARY

In this module we have discussed structured software generation in the field of automation.

Section “The Software Enineering Process” presented concepts of software engineering. We have seen that computer programming is not just coding, but that typical steps in software generation are requirements analysis, software specification, software design, coding, testing, documentation and maintenance.

We have also seen that software quality is much more than a bug-free code, it involves (among others) conformance to requirements and specifications, readability, ease of maintenance and logical structuring into manageable sections.

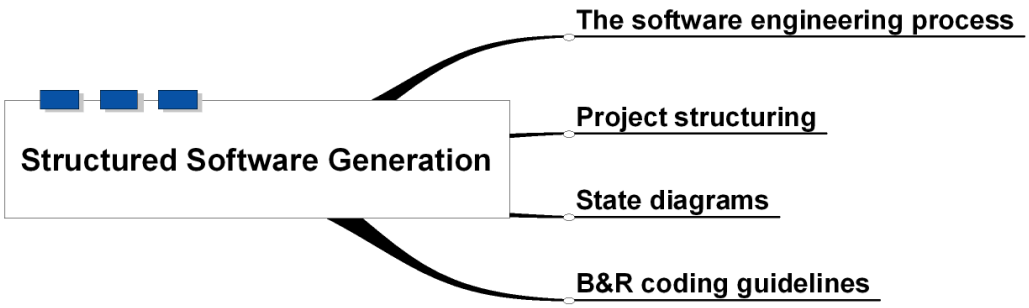


Fig. 17: Objectives

Section “Project Structuring” focused on software architecture and design as the foundations of all good software. We have seen that good programmers don’t rush into coding, but design their software architecture first.

Section “State Diagrams” presented combinatorial and sequential logics, where the latter is capable of storing and remembering information and therefore is perfectly suited to describe the logical function of machines.

Finite state machines are a formal and exact description of sequential machine logics and consist of states, events, actions and transitions. State diagrams are graphical representations of finite state machines.

The concept of state diagrams and finite state machines is not only useful to develop correct code but also to discuss machine functions on an exact and formalized level with customers and for software specification.

Section “B&R Coding Guidelines” presented the B&R Coding Guidelines for automation application software which should guide the user in developing a programming style to produce, test and document high quality source code.

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM241 – Function Block Diagram (FBD)
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB)
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I
TM261 – Closed Loop Control with LOOPCONR

TM400 – The Basics of Motion Control
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors

TM500 – The Basics of Integrated Safety Technology
TM510 – ASiST SafeDESIGNER

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 – Automation Net PVI
TM710 – PVI Communication
TM711 – PVI DLL Programming
TM712 – PVIServices
TM730 – PVI OPC

TM800 – APROL System Concept
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming
TM890 – The Basics of LINUX

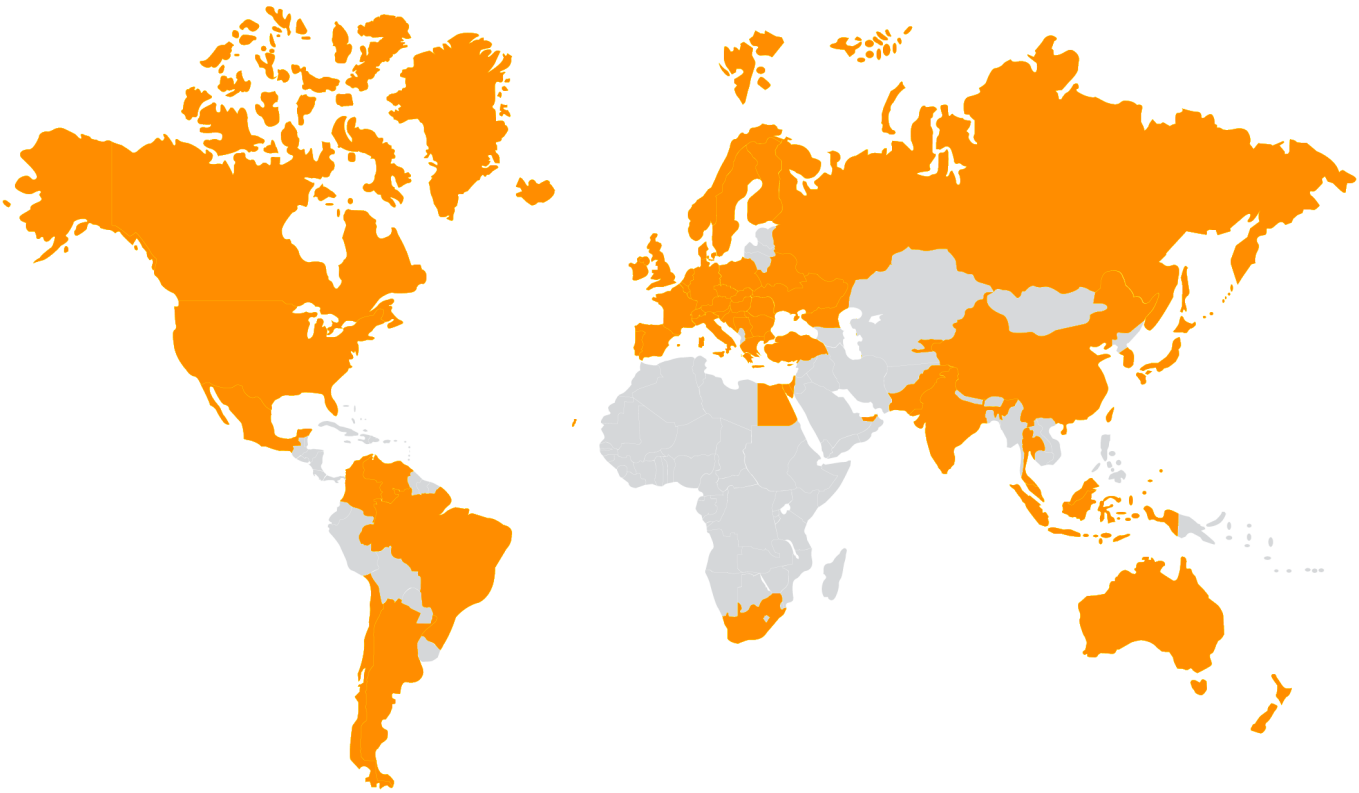
**) see Product Catalog

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Strasse 1
5142 Eggelsberg
Austria
Tel.: +43 (0) 77 48 / 65 86 - 0
Fax: +43 (0) 77 48 / 65 86 - 26
info@br-automation.com
www.br-automation.com

TM230TRE.00-ENG 0907

140 offices in more than 55 countries - **www.br-automation.com/contact**



Australia • Argentina • Austria • Belarus • Belgium • Brazil • Bulgaria • Canada • Chile • China • Colombia • Croatia • Cyprus
Czech Republic • Denmark • Egypt • Emirates • Finland • France • Germany • Greece • Hungary • India • Indonesia
Ireland • Israel • Italy • Japan • Korea • Luxemburg • Kyrgyzstan • Malaysia • Mexico • The Netherlands • New Zealand
Norway • Pakistan • Poland • Portugal • Romania • Russia • Serbia • Singapore • Slovakia • Slovenia • South Africa
Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • Ukraine • United Kingdom • USA • Venezuela • Vietnam