MASTERARBEIT

Automatisierungsbibliothek für VB.NET

Dipl.-Ing. (FH) Johannes Glaser

masterarbeit@johannes-glaser.de

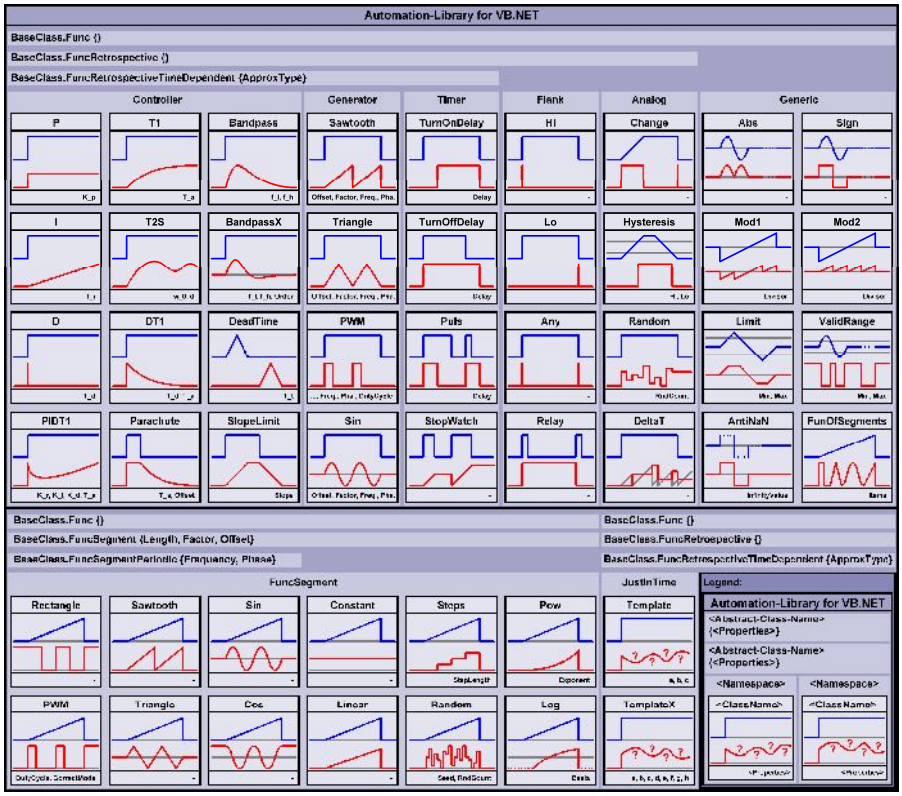
Matrikel-Nr. 3614719

Fakultät Elektrotechnik

Masterstudiengang Elektro- und Informationstechnik

SS 2015

01.10.2014 – 30.09.2015



FHWS

Hochschule

für angewandte Wissenschaften Würzburg-Schweinfurt

Prof. Dr. Jochen Seufert

jochen.seufert@fhws.de

+ 49 9721 940 723

Inhaltsverzeichnis

Kurzfassung **6**

1 Einleitung **7**

1.1 Programmiersprachen 8

1.1.1 Klassische Programmiersprachen 8

1.1.2 Hardwarenahe Programmiersprachen 9

1.1.3 Grafische Programmiersprachen 10

1.1.4 Objektorientierte Programmiersprachen 11

1.2 Steuerungen 12

1.2.1 Klassische Automatisierungssteuerungen 12

1.2.2 Moderne Automatisierungssteuerungen 13

1.3 Idee 14

1.4 Erfahrungen 15

1.4.1 Projekt Wickelmaschine 15

1.4.2 Kugelwaage 19

1.5 Problem 21

1.6 Lösung 21

1.7 Vorstellung eines Funktionsblocks der Bibliothek 22

1.7.1 Mathematische Beschreibung 23

1.7.2 Approximation 24

1.7.3 Stabilität 26

1.7.4 Implementierung 27

1.7.5 Anwendungsbeispiele 28

2 Aufbau der Bibliothek **29**

2.1 Basisklassen (BaseClass) 31

2.1.1 Funktionsblock 31

2.1.2 Rückblickender Funktionsblock 32

2.1.3 Zeitabhängiger Funktionsblock 33

2.1.4 Funktionssegment 34

2.1.5 Periodisches Funktionssegment ... 35
2.2 Implementierung eines Funktionsblocks ... 36
2.3 Hinweise zur Implementierung ... 38
2.4 Datentyp „Double“ ... 40
3 Dokumentation der Funktionsblöcke ... 42
3.1 Regelglieder (Controller) ... 45
3.1.1 Proportionalglied (Controller.P) ... 49
3.1.2 Integrationsglied (Controller.I) ... 51
3.1.3 Differenzierer (Controller.D) ... 53
3.1.4 PIDT1-Übertragungsglied (Controller.PIDT1) ... 55
3.1.5 Verzögerungsglied 1. Ordnung (Controller.T1) ... 59
3.1.6 Verzögerungsglied 2. Ordnung (Controller.T2S) ... 61
3.1.7 Verzögernder Differenzierer (Controller.DT1) ... 63
3.1.8 Fallschirm (Controller.Parachute) ... 65
3.1.9 Bandpass (Controller.Bandpass) ... 67
3.1.10 Bandpass höherer Ordnung (Controller.BandpassX) ... 69
3.1.11 Totzeitglied (Controller.DeadTime) ... 71
3.1.12 Steigungsbegrenzer (Controller.SlopeLimit) ... 73
3.2 Signalgeneratoren (Generator) ... 75
3.2.1 Sägezahn-Generator (Generator.Sawtooth) ... 77
3.2.2 Dreieck-Generator (Generator.Triangle) ... 79
3.2.3 PWM-Generator (Generator.PWM) ... 81
3.2.4 Sinus-Generator (Generator.Sin) ... 83
3.3 Zeitglieder (Timer) ... 85
3.3.1 Anzugsverzögerung (Timer.TurnOnDelay) ... 87
3.3.2 Abfallverzögerung (Timer.TurnOffDelay) ... 89
3.3.3 Impuls (Timer.Puls) ... 91
3.3.4 Stoppuhr (Timer.StopWatch) ... 93
3.4 Flankenerkennungen (Flank) ... 95

3.4.1 Positive Flankenerkennung (Flank.Hi) ... 97
3.4.2 Negative Flankenerkennung (Flank.Lo) ... 99
3.4.3 Flankenerkennung (Flank.Any) ... 101
3.4.4 Toggle-Flipflop (Flank.Relay) ... 103
3.5 Analoge Übertragungsglieder (Analog) ... 105
3.5.1 Wertänderung (Analog.Change) ... 107
3.5.2 Hysterese (Analog.Hysteresis) ... 109
3.5.3 Zufallsgenerator (Analog.Random) ... 111
3.5.4 Zykluszeit (Analog.DeltaT) ... 113
3.6 Grundlegende Funktionen (Generic) ... 115
3.6.1 Betrag (Generic.Abs) ... 117
3.6.2 Vorzeichen (Generic.Sign) ... 119
3.6.3 Modulo1 (Generic.Mod1) ... 121
3.6.4 Modulo2 (Generic.Mod2) ... 123
3.6.5 Begrenzer (Generic.Limit) ... 125
3.6.6 Gültigkeitsbereich (Generic.ValidRange) ... 127
3.6.7 AntiNaN (Generic.AntiNaN) ... 129
3.6.8 Abschnittsweise definierte Funktion (FuncOfSegments) ... 131
3.7 Funktions-Segmente (FuncSegment) ... 133
3.7.1 Konstante Funktion (FuncSegment.Constant) ... 135
3.7.2 Lineare Funktion (FuncSegment.Linear) ... 137
3.7.3 Stufenfunktion (FuncSegment.Steps) ... 139
3.7.4 Zufallsfunktion (FuncSegment.Random) ... 141
3.7.5 Potenzfunktion (FuncSegment.Pow) ... 143
3.7.6 Logarithmische Funktion (FuncSegment.Log) ... 145
3.7.7 Rechteck-Funktion (FuncSegment.Rectangle) ... 147
3.7.8 PWM-Funktion (FuncSegment.PWM) ... 149
3.7.9 Sägezahn-Funktion (FuncSegment.Sawtooth) ... 151
3.7.10 Dreieck-Funktion (FuncSegment.Triangle) ... 153

3.7.11 Sinus-Funktion (FuncSegment.Sin) ... 155
3.7.12 Cosinus-Funktion (FuncSegment.Cos) ... 157
3.8 Kompilierung zur Laufzeit (JustInTime) ... 159
3.8.1 Funktionsblock-Vorlage (JustInTime.Template) ... 161
3.8.2 Erweiterte Funktionsblock-Vorlage (JustInTime.TemplateX) ... 163
4 Testumgebung ... 165
4.1 Funktionsweise ... 165
4.2 Testsignale ... 168
4.3 Erstellen eines Funktionsblock-Tests ... 171
4.4 Schaubild ... 177
4.5 Wertetabelle ... 182
4.6 Integrierte Entwicklungsumgebung ... 183
4.7 Fehlerbehebung ... 184
5 Beispielprojekt Audio-Analyse ... 185
5.1 Programm „DJ JOE Genius“ ... 186
5.1.1 Funktionsweise ... 187
5.1.2 Programmierung ... 187
5.1.3 Einsatz der Automatisierungsbibliothek ... 187
5.2 Programm „Audio Streamer“ ... 188
5.2.1 Funktionsweise ... 188
5.2.2 Programmierung ... 189
5.2.3 Einsatz der Automatisierungsbibliothek ... 190
5.3 Programm „Audio Analyzer“ ... 191
5.3.1 Funktionsweise ... 191
5.3.2 Programmierung ... 192
5.3.3 Einsatz der Automatisierungsbibliothek ... 197
6 Ergebnis ... 198
Literaturverzeichnis ... 199
Abbildungsverzeichnis ... 200

Selbstständigkeitserklärung **205**

Anhang **206**

Programmcode-Verzeichnis 206

Programmcode der Automatisierungsbibliothek 208

Namensbereich-Verzeichnis 231

Funktionsblock-Verzeichnis (alphabetisch sortiert) 232

Funktionsblock-Verzeichnis (nach Namensbereich sortiert) 234

Kurzfassung

In Zeiten der Industrie 4.0 bieten die auf dem Markt erhältlichen Anlagensteuerungen und Softwarepakete Lösungen für alle gängigen Problemstellungen der Automatisierungstechnik. Für sehr spezielle Anwendungen kann es jedoch erforderlich sein, eine individuell auf die Anforderung zugeschnittene Steuerungssoftware zu entwickeln. Ein Beispiel hierfür liefert das vorangegangene Diplomarbeitsprojekt „Automatisiertes Bewickeln von Gummikompensatoren“, bei dem eine eigens in VB.NET geschriebene Steuerungssoftware das Automatisieren eines sehr komplexen Arbeitsschritts ermöglichte.

Ziel dieser Masterarbeit ist es, auf Basis der im Diplomarbeitsprojekt gewonnenen Erfahrungen eine Automatisierungsbibliothek zu entwickeln, um die sehr umfangreiche Entwicklungsumgebung Visual Studio und die mächtige Programmiersprache VB.NET für weitere Projekte der Automatisierungstechnik noch effektiver einsetzen zu können. Diese Bibliothek soll alle wichtigen Elemente bereitstellen, die häufig bei der Programmierung von computerbasierten Anlagensteuerungen zum Einsatz kommen. Hierzu gehören in erster Linie signalverarbeitende Funktionsblöcke wie z. B. Übertragungsglieder aus der Regelungstechnik, Signalgeneratoren und Zeitglieder, die für nicht-zeitäquidistante Zykluszeiten ausgelegt sind.

Die Masterarbeit setzt sich im ersten Kapitel intensiv mit Steuerungen von Anlagen und den Programmiersprachen der Automatisierungstechnik auseinander. Dazu werden Erfahrungen aus vorhergehenden Projekten diskutiert und der Einsatz der Programmiersprache VB.NET zum Erstellen von Steuerungssoftware kritisch bewertet.

Das zweite Kapitel erklärt den Aufbau der Automatisierungsbibliothek und gibt wichtige Hinweise für deren Nutzung in späteren Softwareprojekten. Hierzu ist im darauf folgenden Kapitel eine Dokumentation zu finden, die alle 50 Elemente der Bibliothek detailliert erklärt und deren Funktionsweise mit Schaubildern verdeutlicht.

Die im Rahmen der Masterarbeit entwickelte Testumgebung wird im vierten Kapitel vorgestellt. Eine ausführliche Anleitung zeigt zusätzlich, wie Funktionsbausteine mit der Software getestet werden können und die Bibliothek mit der integrierten Entwicklungsumgebung erweitert werden kann.

Das letzte Kapitel dieser Arbeit präsentiert ein Beispielprojekt aus dem Bereich Audio- und Lichttechnik, zu dessen Umsetzung die Automatisierungsbibliothek umfassend eingesetzt wurde. Durch das Projekt werden nochmals die Vorteile und Möglichkeiten der entwickelten Bibliothek veranschaulicht und dabei Anregungen für den Einsatz der Bibliothek in späteren Projekten gegeben.

1 Einleitung

Das Herzstück einer jeden Fertigungs- oder Prozessanlage ist ihre Anlagensteuerung. Sie plant, steuert, überwacht und protokolliert den Prozess. Hierbei tauscht die Steuerung kontinuierlich Prozessdaten mit Sensoren und Aktoren aus, verarbeitet diese Daten und steht mit den Anlagenführern über Bedien- und Anzeigeelemente in Verbindung. Die verschiedenen Aufgaben einer Anlagensteuerung erfordern zur Realisierung Programmiersprachen und Zielsysteme mit unterschiedlichen Eigenschaften. Oftmals stehen hierbei Abstraktionsgrad und Hardwarenähe sowie Rechenleistung und Echtzeitfähigkeit im Widerspruch. Deshalb kommen bei der Realisierung einer Steuerung im Allgemeinen mehrere unterschiedliche Programmiersprachen und Zielsysteme zum Einsatz, die auf die Anforderungen der jeweiligen Aufgaben zugeschnitten sind.

In dieser Masterarbeit wird unter dem Begriff Anlagensteuerung nicht nur die Steuerelektronik und die dazugehörige Steuerungssoftware von Fertigungs- oder Prozessanlagen verstanden, sondern auch die von Prüfständen, „Hardware in the Loop“ - Simulatoren, Versuchsaufbauten sowie Messeinrichtungen.

Abb. 1-1: Verschiedene Anlagentypen

In den folgenden Unterkapiteln werden zunächst die in der Automatisierungstechnik eingesetzten Programmiersprachen vorgestellt und der Aufbau von klassischen sowie modernen Anlagensteuerungen erklärt. Anschließend wird die Idee diskutiert, computerbasierte Anlagensteuerungen mit VB.NET umzusetzen und dabei von Erfahrungen aus früheren Projekten berichtet. Am Ende des Einführungskapitels wird die Automatisierungsbibliothek vorgestellt, die den Einsatz der Programmiersprache noch effektiver macht.

1.1 Programmiersprachen

Im Folgenden werden die Eigenschaften gängiger Programmiersprachen der Automatisierungstechnik vorgestellt und ihre Anwendungsbereiche erklärt.

1.1.1 Klassische Programmiersprachen

Die klassischen Programmiersprachen FUP, KOP, AWL und ST wurden entwickelt, um Programme für speicherprogrammierbare Steuerungen (SPS) zu schreiben, die den Prozessablauf einer Anlage steuern. In allen vier Sprachen werden durch Verknüpfen von Eingangssignalen die Ausgänge einer SPS beschaltet. Somit können die Zustände von Sensoren (z. B. Taster) direkt eingelesen und Aktoren (z. B. Signalleuchten) direkt angesteuert werden. Zum Verarbeiten von digitalen Signalen stehen neben logischen Verknüpfungen grundlegende Funktionsbausteine wie beispielsweise Merker, Zeitglieder und Zähler zur Verfügung. Analoge Signale hingegen werden mit mathematischen Funktionen und Reglern verarbeitet. Die Visualisierung von Prozessdaten ist mit den klassischen Programmiersprachen nur bedingt über kleine Displays möglich. Für aufwändigere grafische Benutzeroberflächen muss auf Prozessvisualisierungssysteme wie zum Beispiel WinCC zurückgegriffen werden.

Die Programmierung kann grafisch durch das Verschalten von Funktionsblöcken in einem Funktionsplan (FUP) oder durch Erstellen eines Kontaktplans (KOP) erfolgen, der sich an die Stromlaufpläne anlehnt, mit denen Elektroniker vertraut sind. Eine textuelle Programmierung ist mit einer Anweisungsliste (AWL) aus CPU-nahen Befehlen oder mittels hochsprachenähnlichem strukturierten Text (ST) möglich. Die folgende Grafik zeigt eine einfache Selbsthalteschaltung, die in die jeweiligen Programmiersprachen umgesetzt ist.

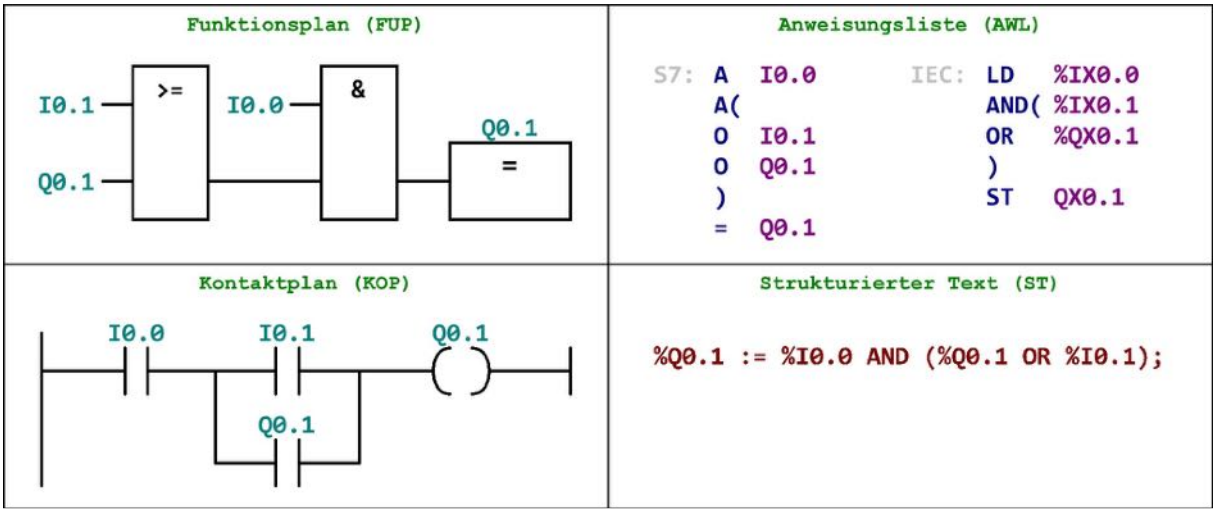


Abb. 1-2: Selbsthalteschaltung, umgesetzt in FUP, KOP, AWL und ST

Aufgrund der verschiedenen Darstellungsmöglichkeiten des Programmcodes sind die klassischen Programmiersprachen auch für Elektroniker ohne Programmierkenntnisse leicht verständlich. Deshalb werden die Sprachen auch noch heute in modernen Industrie-PCs eingesetzt.

1.1.2 Hardwarenahe Programmiersprachen

Hardwarenahe Programmiersprachen wie C, C++ und Java werden in verschiedenen Bereichen der Automatisierungstechnik zur zeitkritischen Verarbeitung von Daten und Signalen eingesetzt. Ein Einsatzgebiet stellen eingebettete Echtzeitsysteme dar, in denen Firmware-Programme auf Mikrocontrollern ausgeführt werden, um die Funktionalitäten von Sensor- und Aktor-Elektronik übergeordneten Steuerungen zugänglich zu machen. Solche Firmware ist zum Beispiel in Industrierobotern, Antriebsreglern und intelligenten Distanzsensoren zu finden und wird vom jeweiligen Hersteller bereitgestellt. Hardwarenahe Programmiersprachen sind aber auch für Ingenieure von großer Bedeutung, die die Steuerung einer Anlage umsetzen, um auf Industrie-PCs sehr zeitkritische und komplexe Aufgaben zu lösen, welche weit über das Steuern des Prozessablaufs einer Anlage hinausgehen.

Die hardwarenahe Hochsprache C bietet die Möglichkeit, mit Daten- und Kontrollstrukturen wie zum Beispiel Feldern, Zeichenketten, Bedingungen, Schleifen sowie Funktionen zu programmieren und ermöglicht das Umsetzen hardwarenaher Programme für einfachere Anwendungen. Mit der ebenso hardwarenahen, aber zudem objektorientierten Sprache C++ können abstrakte Programmstrukturen erstellt und somit wesentlich komplexere Aufgaben gelöst werden. Java unterscheidet sich von C++ hauptsächlich wie folgt: Zum einen ist Java wesentlich robuster ausgelegt und verfügt über eine automatische Speicherbereinigung. Zum anderen werden in Java geschriebene Programme in Laufzeitumgebungen ausgeführt, die das einfache Portieren auf verschiedene Zielsysteme ermöglichen, aber zudem auch eine gewisse Hardwaredistanz zur Folge haben.

```c
// Programmiersprache: C
// Zielsystem: ATMEGA8

#include <util/delay.h>
#include <avr/io.h>

int main (void) {

  DDRB  |= (1<<5);

  while(1) {
    PORTB |= (1<<5);
    _delay_ms(500);
    PORTB &=~(1<<5);
    _delay_ms(500);
  }

  return 0;
}
```

```c
// Programmiersprache: C++
// Zielsystem: Arduino Uno

int led = 13;
int time = 500;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(time);
  digitalWrite(led, LOW);
  delay(time);
}
```

```java
// Programmiersprache: Java
// Zielsystem: Raspberry Pi

public class ControlGpioExample {
  public static void main() {

    final GpioController gpio =
    GpioFactory.getInstance();

    final GpioPinDigitalOutput pin =
    gpio.provisionDigitalOutputPin(
    RaspiPin.GPIO_01);

    while(true) {
      pin.toggle();
      Thread.sleep(500);
    }
  }
}
```

Abb. 1-3: Blinkende LED, umgesetzt in C, C++ und Java

Der in C, C++ und Java geschriebene Programmcode zeigt, wie mit der jeweiligen Sprache direkt auf die physikalisch greifbaren Bits eines Hardwareregisters zugegriffen wird, um eine am Zielsystem angeschlossene LED blinken zu lassen.

1.1.3 Grafische Programmiersprachen

Im Gegensatz zur textuellen Programmierung steht bei grafischen Programmiersprachen wie LabVIEW und Simulink der Datenfluss im Vordergrund. Die Programmierung erfolgt durch das Erstellen eines Blockdiagramms per Drag & Drop. Hierbei werden Funktionsblöcke eingefügt und deren Anschlüsse mittels Daten- und Signallinien verbunden. Umfangreiche Bibliotheken bieten hierzu alle wichtigen Elemente zur Signalverarbeitung und hochsprachigen Programmierung. Benutzeroberflächen können ebenfalls grafisch erstellt werden.

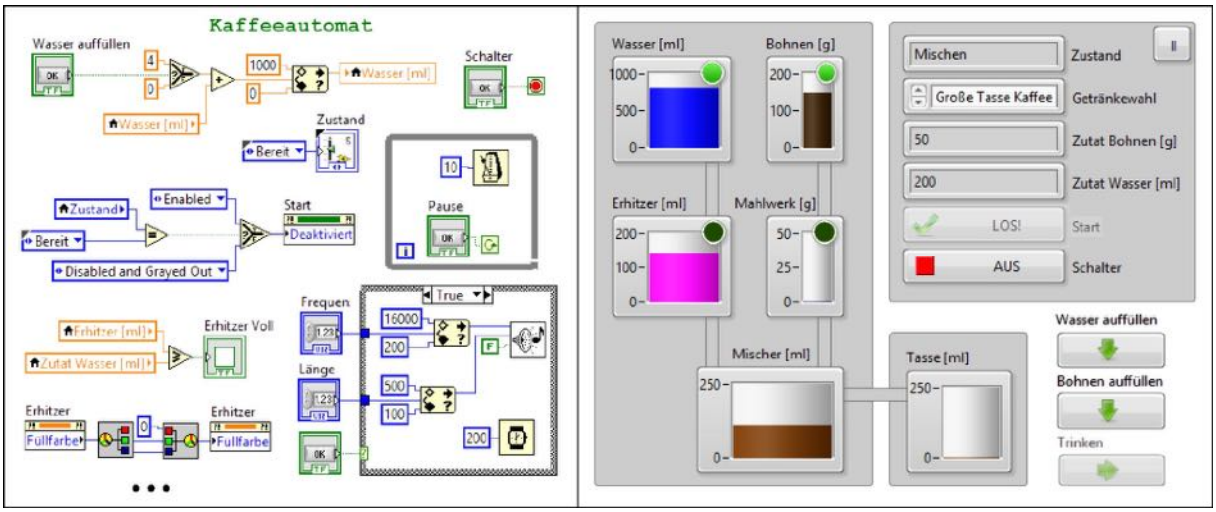


Abb. 1-4: Kaffeeautomat, umgesetzt in LabVIEW

Programme, die mittels LabVIEW oder Simulink erstellt wurden, können auf verschiedenen Plattformen ausgeführt werden. Laufzeitumgebungen für Windows-, Mac- oder Linux-Betriebssysteme ermöglichen die Ausführung mit grafischen Benutzeroberflächen auf einem PC oder Mac. Für zeitkritische Anwendungen lassen sich die Programme aber auch für Industrie-PCs, Mikrocontroller und FPGAs mit DSPs sowie CPUs in eingebetteten Systemen kompilieren und in Echtzeit ausführen.

Für die Anbindung an eine Hardware stellen namhafte Messgerätehersteller Gerätetreiber zur Verfügung, mit denen IO-Karten, Multimeter, Oszilloskope und Frequenzgeneratoren einfach als Funktionsblock in ein LabVIEW-Programm eingefügt und darüber ferngesteuert werden können. Auch die üblichen Hardwareschnittstellen eines PCs wie Soundkarte und serielle Schnittstelle lassen sich auch direkt über Funktionsblöcke ansprechen.

Aufgrund der visuellen Darstellung von Datenströmen und Signalen in Blockschaltbildern sind grafische Programmiersprachen besonders gut für Aufgaben aus der Mess- und Regelungstechnik geeignet und im Vergleich zu textuellen Sprachen auch für Ingenieure aller Fachrichtungen leicht zu verstehen. Deshalb werden zum Beispiel LabVIEW und Simulink häufig im Prototypenbau an Prüfständen und Simulatoren eingesetzt, an denen Ingenieure mit sehr unterschiedlichen Programmierkenntnissen arbeiten.

1.1.4 Objektorientierte Programmiersprachen

Mit objektorientierten Programmiersprachen ist es möglich, die Architektur einer Software an den Grundstrukturen der Wirklichkeit auszurichten. Dies ermöglicht die Umsetzung sehr komplexer Problemstellungen ohne die Einschränkung auf einen bestimmten Problemtyp. Deshalb stellt die objektorientierte Programmierung für die Automatisierungstechnik ein großes Potenzial an Möglichkeiten dar. Jedoch sind für die Programmierung mit solchen Sprachen höhere Kenntnisse von Softwareentwicklern erforderlich, die bei Automatisierungsprojekten nicht immer zur Verfügung stehen. Zudem sind für die meisten Automatisierungsanwendungen die Möglichkeiten einfacherer Sprachen ausreichend. Aus diesen Gründen werden für einfache Ablaufsteuerungen bevorzugt klassische Automatisierungssprachen eingesetzt. So werden nur spezielle Erweiterungsmodule oder grafische Benutzeroberflächen in objektorientierte Hochsprachen umgesetzt, für die fertige Softwarepakete keine Lösung bieten.

Bei der objektorientierten Programmierung beschäftigt sich der Entwickler mit den Elementen der Problemstellung und bildet diese in gleicher Struktur durch das Erstellen von Klassen im Programmcode ab. Eine Klasse beschreibt, ähnlich wie ein Konstruktionsplan, die Eigenschaften (Attribute) und Funktionen (Methoden) beliebiger Objekte, die untereinander kooperieren können. Das folgende Beispiel zeigt eine einfache Klasse, die in die Programmiersprachen VB.NET und C# umgesetzt wurde und den Kraftstoffverbrauch von Autos simuliert.

```vb
'Programmiersprache: VB.NET

Public Class Auto

    Property Kilometerstand As Double
    Property Tankfüllstand As Double
    Property Verbrauch As Double

    Sub New(ByVal Verbrauch As Double)
        Me.Verbrauch = Verbrauch
        Me.Tankfüllstand = 80
    End Sub

    Public Sub Fahren(ByVal Distanz As Double)
        Kilometerstand += Distanz
        Tankfüllstand -= Verbrauch * Distanz
    End Sub

End Class
```

```cs
//Programmiersprache: C#

public class Auto {

  public double Kilometerstand { get; set; }
  public double Tankfüllstand { get; set; }
  public double Verbrauch { get; set; }

  public Auto(double Verbrauch) {
    this.Verbrauch = Verbrauch;
    this.Tankfüllstand = 80;
  }

  public void Fahren(double Distanz) {
    Kilometerstand += Distanz;
    Tankfüllstand -= Verbrauch * Distanz;
  }

}
```

Abb. 1-5: Einfache Autosimulation, umgesetzt in VB.NET und C#

Die objektorientierten Hochsprachen VB.NET und C# von Microsoft stellen zusammen mit der dazugehörigen Entwicklungsumgebung Visual Studio und dem .NET-Framework ein sehr mächtiges Werkzeug dar, mit dem aufwändige und komplexe Softwareprojekte umgesetzt werden können. Das Programmieren von 64-Bit-Anwendungen, grafischen Benutzeroberflächen, Visualisierungen mit Direct3D, Anbindung an Datenbanken und Multithreading sind nur wenige Beispiele für Möglichkeiten, die VB.NET und C# bieten.

1.2 Steuerungen

Im Folgenden wird die Bedeutung der bereits vorgestellten Programmiersprachen in klassischen sowie in modernen Automatisierungssteuerungen erklärt.

1.2.1 Klassische Automatisierungssteuerungen

In klassischen Automatisierungslösungen spiegelt sich die funktionelle Gliederung der Automatisierungspyramide in den Steuerungskomponenten und den dafür eingesetzten Programmiersprachen wider.

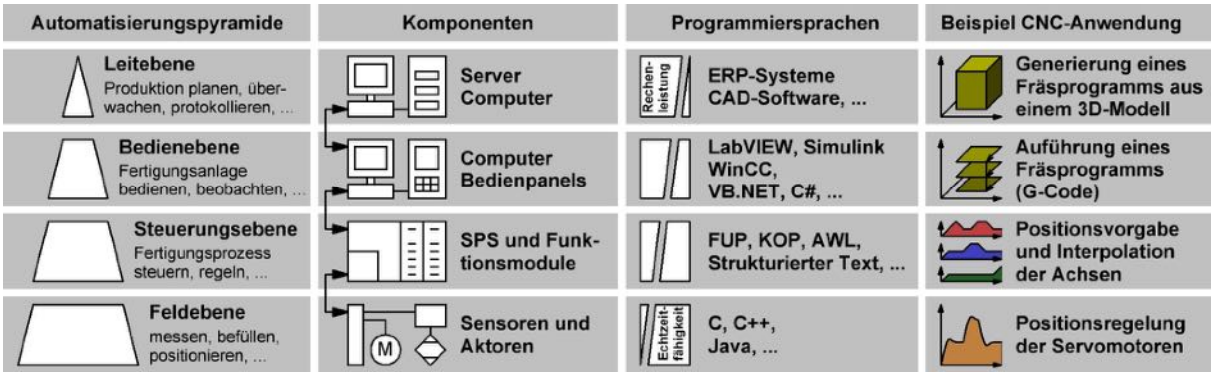


Abb. 1-6: Aufbauschema einer klassischen Automatisierungssteuerung

Der Aufbau einer klassischen Automatisierungssteuerung lässt sich gut anhand des CNC-Beispiels aus der letzten Spalte des Schemas (Abb. 1-6) erklären. Die sehr rechenaufwändigen Aufgaben, wie zum Beispiel das Generieren von Fräsprogrammen aus 3D-Modellen, werden von Servern der Leitebene übernommen. Computer der Bedienebene führen die Fräsprogramme aus, indem sie einzelne Steuerbefehle an die Steuerung der CNC-Fräse weitergeben. Ein solcher Befehl könnte zum Beispiel ein G-Code-Befehl sein, der eine Fräserbewegung im dreidimensionalen Raum beschreibt. Die SPS-Steuerung der Steuerungsebene nimmt diese Befehle entgegen und berechnet daraus in kurzen Zeitintervallen die Sollpositionen der einzelnen Achsen. Für die zeitkritische Positionsregelung der Stellantriebe kommen Antriebsregler, die mit kleineren Prozessoren ausgestattet sind, in der Feldebene zum Einsatz.

Je nach Automatisierungsebene werden für die Programmierung der eingesetzten Komponenten unterschiedliche Programmiersprachen eingesetzt. In den oberen Ebenen kommen objektorientierte Hochsprachen zum Einsatz, um abstrakte und komplexe Programme zu erstellen, die große Datenmengen verarbeiten können und auf Computern oder Servern ausgeführt werden. In den unteren Ebenen hingegen werden auf echtzeitfähigen Systemen zeitkritische Steuerungen und Regelungen mit hardwarenahen Programmiersprachen realisiert.

In klassischen Automatisierungslösungen findet der Datenaustauch nur zwischen Komponenten benachbarter Ebenen statt. Die hierfür eingesetzten Bus-Systeme sind in den oberen Ebenen für hohe Datenraten und in den unteren Ebenen für eine zeitkritische Übertragung ausgelegt.

1.2.2 Moderne Automatisierungssteuerungen

In modernen Automatisierungslösungen kommen Netzwerke aus Multifunktions-Komponenten zum Einsatz, die Aufgaben mehrerer Automatisierungsebenen übernehmen können. Die Verarbeitung von Informationen ist somit nicht an eine bestimmte Komponente gebunden. Funktionen der Steuerungsebene müssen nicht klassisch in einer SPS abgearbeitet werden, sondern können auch von intelligenten Geräten der Bedien- oder Feldebene übernommen werden. Beispielsweise stellt ein moderner Antriebsregler neben der Motorregelung in der Feldebene auch erweiterte SPS-Funktionalitäten zur Verfügung, mit denen Aufgaben aus der Steuerungsebene und Bedienebene realisiert werden können. Aufgrund der engen Vernetzung der Komponenten ist es auch möglich, alle Steuerungsprozesse der Anlage in einem zentralen Industrie-PC abzuarbeiten, der mit allen Komponenten in Verbindung steht.

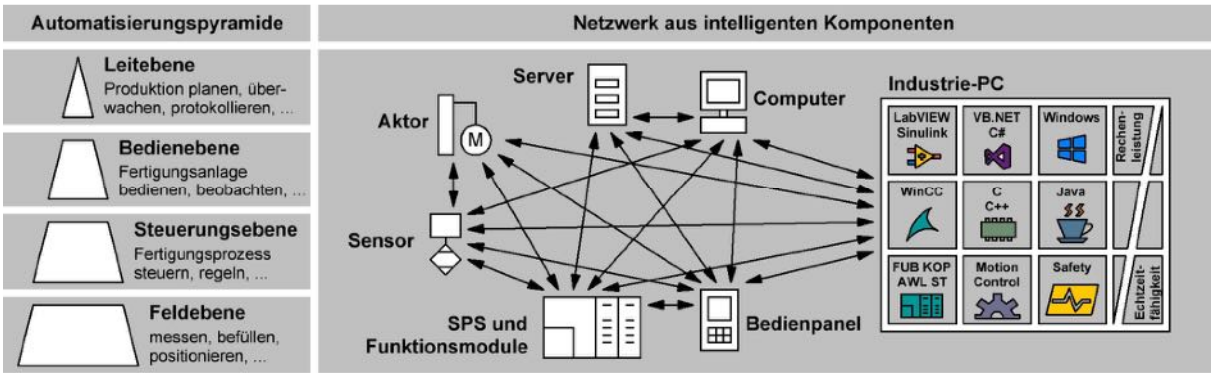


Abb. 1-7: Aufbauschema einer modernen Automatisierungssteuerung

Ein Industrie-PC ist im Vergleich zu einer herkömmlichen speicherprogrammierbaren Steuerung wesentlich leistungsfähiger und flexibler. Beliebige Automatisierungskomponenten können über Bussysteme oder Netzwerke angebunden und die Funktionalitäten eines Industrie-PCs mit Softwarepaketen wie zum Beispiel TwinCAT, CodeSys und Simatic beliebig erweitert werden. Im Folgenden wird das umfassende Softwarepaket TwinCAT der Firma Beckhoff kurz vorgestellt, um die Möglichkeiten einer PC-basierten Steuerung zu verdeutlichen.

TwinCAT setzt sich aus einer Entwicklungsumgebung, welche auf dem Visual Studio von Microsoft basiert, und einer Laufzeitumgebung zusammen. Das Softwarepaket ermöglicht mit nur einer Entwicklungsumgebung die Programmierung in FUP, KOP, AWL, ST, C und C++ und führt den Code auf einem Industrie-PC in einer Laufzeitumgebung, die eine Ebene unter dem Betriebssystem läuft, in Echtzeit aus. Die Programmierung in VB.NET und C# ist ebenfalls möglich. Zudem können LabVIEW- und Simulink-Programme und TwinCAT-Bibliotheken für MotionControl-, Robotik- sowie Safety-Anwendungen eingebunden werden. Zusammengefasst kann auf einem Industrie-PC die komplette Software zum Steuern, Bedienen, Protokollieren und Überwachen einer Anlage ausgeführt werden.

1.3 Idee

Wie im vorherigen Kapitel dargelegt, sind die Anforderungen, die an eine Automatisierungssteuerung gestellt werden, sehr vielfältig und unterschiedlich. Deshalb kommen bei der Umsetzung einer Steuerung viele verschiedene Automatisierungswerkzeuge zum Einsatz, die auf die unterschiedlichen Aufgabenbereiche zugeschnitten sind. Somit können die einzelnen Aufgabenstellungen schnell und einfach gelöst werden. Der Einsatz vieler verschiedener Komponenten, Entwicklungsumgebungen und Programmiersprachen hat aber auch eine große Anzahl an Schnittstellen auf Hardware- und Softwareebene zur Folge, die den Informationsfluss zwischen den Einzelsystemen einschränken. Des Weiteren liefern die auf den Anwendungsfall zugeschnittenen Softwarepakete und Entwicklungsumgebungen zwar schnell eine Lösung für gängige Problemstellungen, sind für sehr spezielle Anwendungen aber oftmals nur schwer oder gar nicht anpassbar oder erweiterbar.

Möchte man einen sehr speziellen und komplexen Fertigungsprozess automatisieren, der sehr hohe Anforderungen an die Steuerung stellt, sucht man nach einer Automatisierungssteuerung, die theoretisch alles kann und beliebig skalierbar ist. Fündig wird man dann zum Beispiel bei der Firma Beckhoff mit der bereits vorgestellten PC-basierten Steuerung TwinCAT. Jedoch kommt es vor, dass die Anwendung so speziell ist, dass nur wenige vorgefertigte Elemente der umfangreichen Automatisierungssoftware genutzt werden können. In solchen Fällen macht es Sinn, eine speziell auf die Anwendung zugeschnittene Steuerungssoftware zu entwickeln.

Im nächsten Kapitel werden zwei solche sehr spezielle Automatisierungsaufgaben vorgestellt, bei denen die Entwicklung einer auf den Anwendungsfall angepassten Steuerungssoftware erforderlich war. Bei der Realisierung dieser Projekte kam die Idee auf, eine der mächtigsten Programmiersprachen der Softwaretechnik für das Entwickeln einer Steuerungssoftware in der Automatisierungstechnik einzusetzen. Genauer wurde das Visual Studio als einziges Entwicklungstool eingesetzt, um die komplette Steuerungssoftware in VB.NET zu schreiben. Der Gedanke hinter dieser Idee ist, alle Module der Steuerung mit nur einer Entwicklungsumgebung, einer Programmiersprache und einem Zielsystem umzusetzen. Somit sollten möglichst wenige Schnittstellen auf Hardware- und Softwareebene entstehen und hierdurch eine maximale Flexibilität sichergestellt werden, um die Realisierbarkeit der automatisierungstechnischen Herausforderungen zu erreichen.

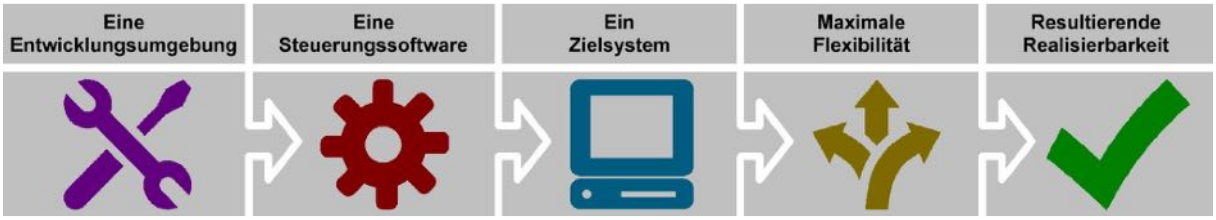


Abb. 1-8: Visuelle Darstellung der Idee

1.4 Erfahrungen

In diesem Kapitel werden zwei ambitionierte Projekte vorgestellt, bei denen der Einsatz von vorgefertigten Softwarepaketen zur Realisierung der Steuerung nicht möglich bzw. aus oben genannten Gründen nicht sinnvoll war. Deshalb wurde jeweils die komplette Steuerungssoftware eigenständig mit der objektorientierten Programmiersprache VB.NET in der Entwicklungsumgebung Visual Studio umgesetzt. Die beiden Beispiel-Projekte aus der Praxis sollen die Möglichkeiten, die durch den Einsatz von VB.NET in der Automatisierungstechnik entstehen, anschaulich darlegen.

1.4.1 Projekt Wickelmaschine

Das erste Projekt stellt das vorhergegangene Diplomarbeitsprojekt „Automatisiertes Bewickeln von Gummikompensatoren“ dar, in dessen Rahmen ein Wickelvorgang automatisiert wurde, bei dem Gewebeband auf einen wellenförmigen Rotationskörper aufgewickelt wird. Hierzu wurden insgesamt drei Antriebe verbaut: ein Getriebemotor, der den Rotationskörper dreht, ein Linearführungssystem, das das Gewebeband positioniert und auf dem Schlitten der Linearführung eine elektronische Bremse, die das Gewebeband geregelt auf Spannung hält.

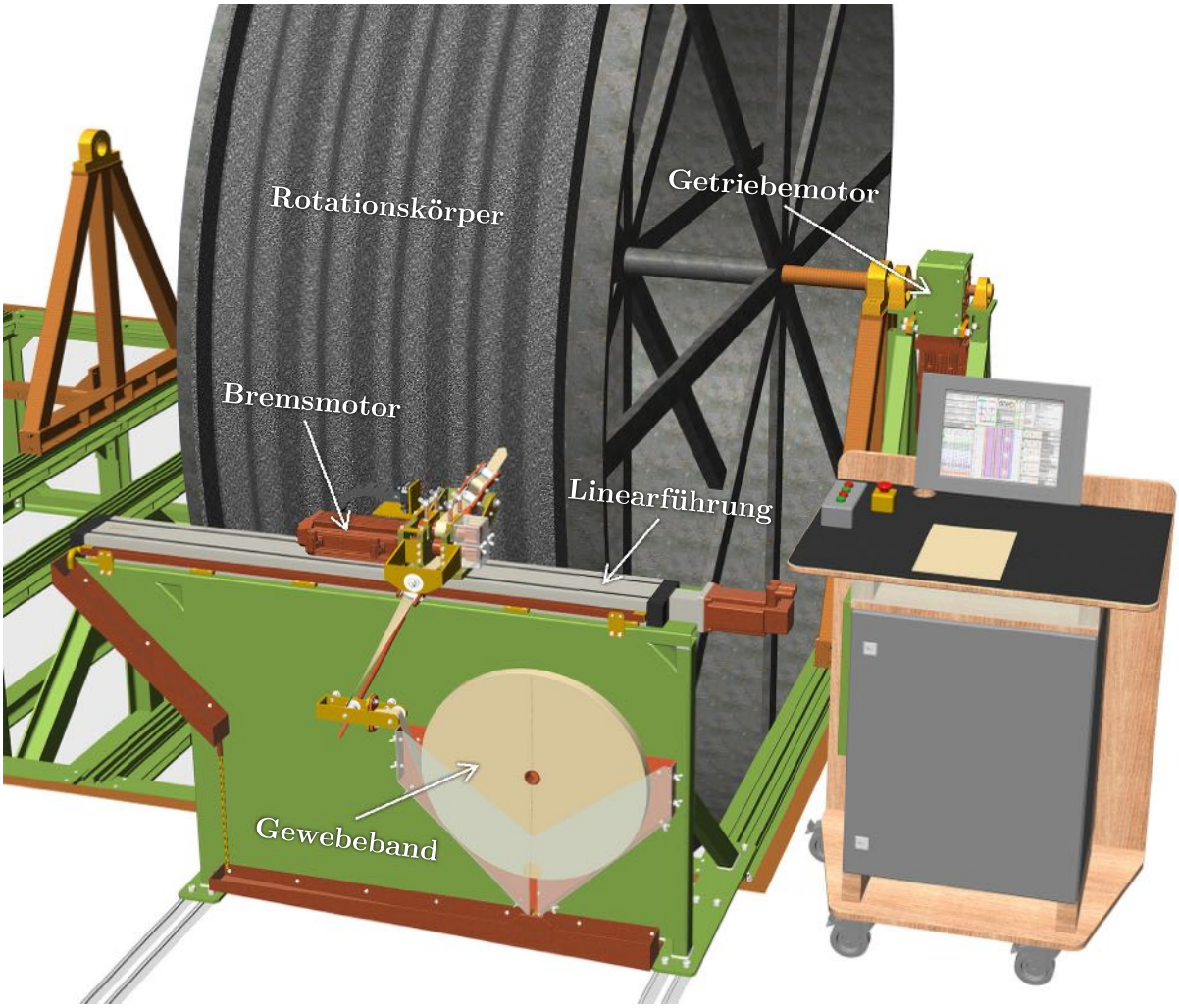


Abb. 1-9: 3D-Modell der Wickelmaschine

Da es sich bei diesem Wickelvorgang um einen Arbeitsschritt aus der Einzelanfertigung handelt und sich somit die zu bewickelnden Rotationskörper in Form und Größe unterscheiden, sind die Anforderungen an die Anlagensteuerung sehr speziell und hoch. Bevor zum Beispiel ein Wickelvorgang ausgeführt werden kann, muss die Anlagensteuerung zunächst die Form des Rotationskörpers bestimmen und anhand eines vorgegebenen Wickelschemas ein Wickelbild generieren, das die Linie beschreibt, entlang der das Gewebeband auf den Rotationskörper gewickelt werden soll. Während des eigentlichen Wickelvorgangs muss die Steuerung den Schlitten der Linearführung in Abhängigkeit der Winkelposition des Getriebemotors nach dem zuvor berechneten Wickelbild positionieren und die Zugkraft des Gewebebands über eine elektronische Bremse regeln. Zudem überwacht die Steuerung über Sensoren die verbleibende Länge des Gewebebandes und steht mit dem Warenwirtschaftssystem über Netzwerk und mit dem Anlagenführer über eine aufwändige Benutzerschnittstelle in Verbindung.

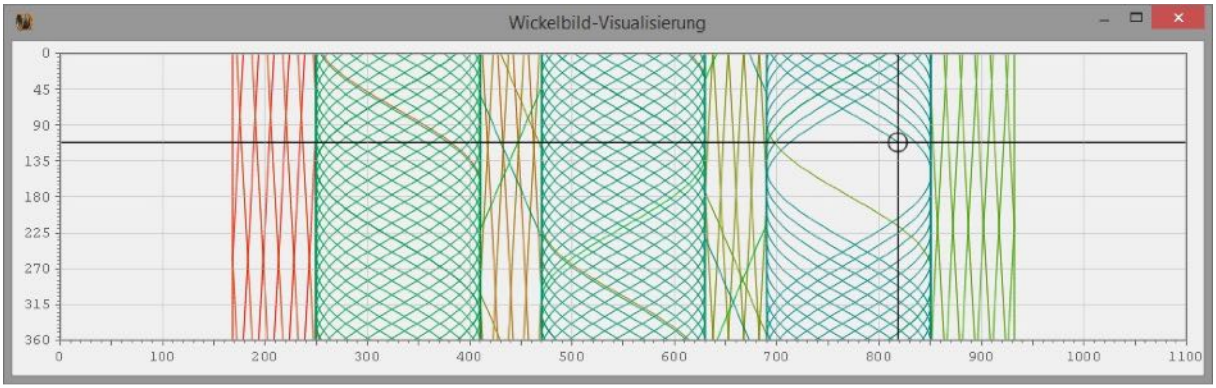


Abb. 1-10: Berechnetes Wickelbild

Die genannten Aufgaben der Anlagensteuerung erfordern eine enge Zusammenarbeit aller Hardwarekomponenten und Softwaremodule. Um dies zu ermöglichen, wurden zunächst möglichst viele Schnittstellen auf Hardwareebene vermieden, indem die komplette Anlage von nur einem zentralen Windows-PC aus gesteuert wird. Die unterschiedlichen Hardware- und Benutzerschnittstellen des PCs ermöglichen es der Steuerungssoftware, auf kürzestem und schnellstem Wege mit externen Automatisierungskomponenten, dem Warenwirtschaftssystem und dem Anlagenführer zu kommunizieren.

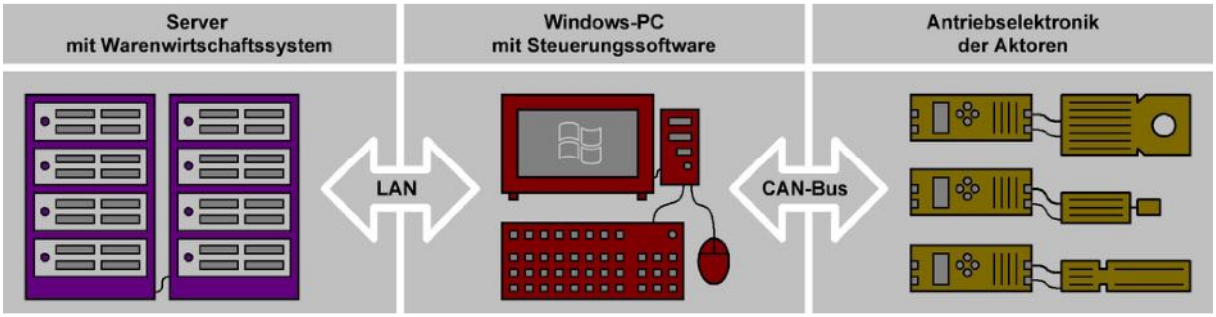


Abb. 1-11: Schematischer Aufbau der Anlagensteuerung

Die Steuerungssoftware der Anlage wurde mit dem Visual Studio in VB.NET geschrieben. Der Einsatz der umfangreichen Entwicklungsumgebung ermöglichte das parallele Entwickeln und Testen der Steuerungssoftware direkt an der Anlage. Beispielsweise konnten Prozessdaten aufgenommen, Regelparameter optimiert und sogar der Programmcode verändert werden, während die Steuerungssoftware ausgeführt wurde.

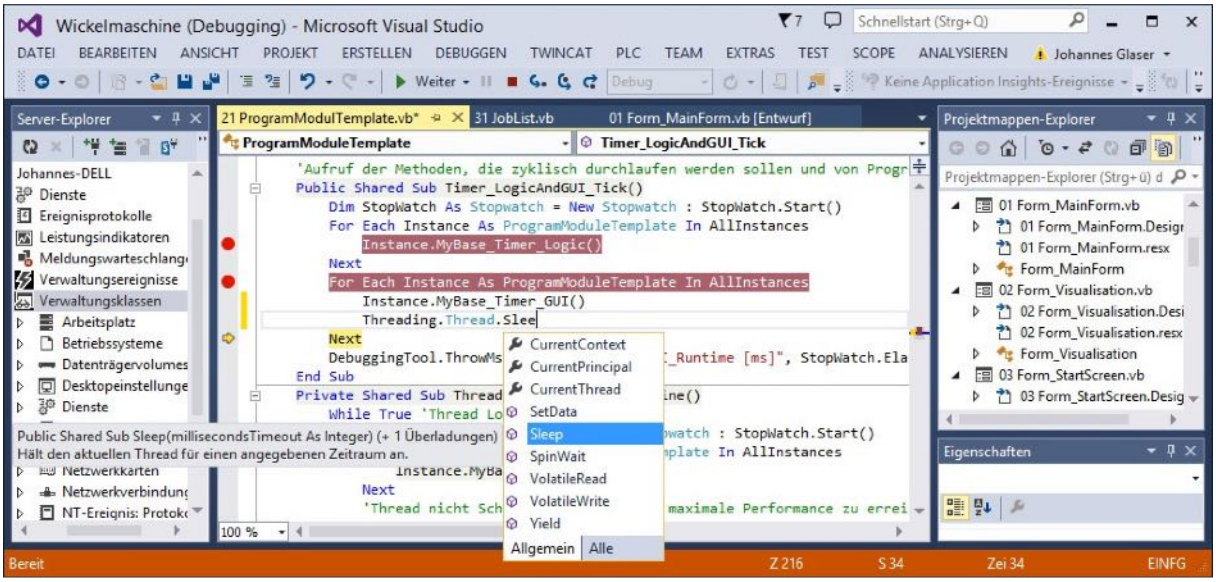


Abb. 1-12: Editieren von Programmcode während der Laufzeit

Die objektorientierte Programmiersprache VB.NET machte es möglich, die Programmstruktur an die realen Hardwarekomponenten und Aufgaben der Anlagensteuerung anzulehnen und das Steuerungsprogramm modular aufzubauen. Dies hat mehrere Vorteile: Zum einen wird der Programmcode überschaubar gehalten, kann gut dokumentiert und einfach erweitert werden. Zum anderen können zeitkritische und rechenintensive Programmteile voneinander getrennt und auf unterschiedlichen CPU-Kernen ausgeführt werden, um die je nach Programmmodul geforderte Rechenleistung und Echtzeitfähigkeit zu erreichen. Die aufgrund der Kapselung zwischen den Programmmodulen entstehenden Schnittstellen sind ausschließlich softwarebasiert und stellen somit keine größeren Zeitverzögerungen oder Engpässe für große Datenmengen dar. Die folgende Abbildung zeigt die prozentuale Auslastung des im Rahmen des Projekts Wickelmaschine eingesetzten Intel Core i7-3740QM Prozessors, während die Steuerungssoftware ausgeführt wurde. Es ist gut zu erkennen, dass der Scheduler die verschiedenen Threads der Software verteilt auf alle CPU-Kerne abarbeitet.

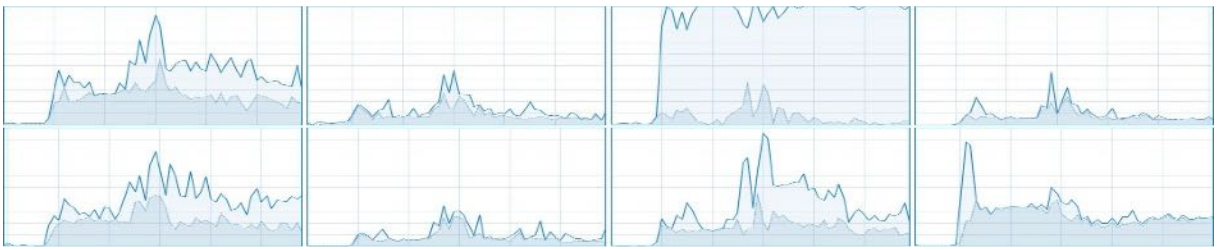


Abb. 1-13: Auslastung der CPU-Kerne in Abhängigkeit der Zeit

Die folgende Abbildung zeigt einen Screenshot der Steuerungssoftware. Im oberen linken Bereich gibt der Anlagenführer Kenndaten ein und wird über den Maschinenzustand informiert. Im oberen rechten Bereich stehen Werkzeuge zur Simulation des Wickelvorgangs und ein Logbuch zur Verfügung, das alle Anlagenereignisse und Benutzereingaben protokolliert. Die Tabellenansicht links und die Grafik in der Mitte visualisieren das Muster, in dem die Wickelmaschine das Gewebeband auf den Rotationskörper legt. Über die Steuerelemente rechts unten kann die Anlage manuell verfahren und es können Einstellungen wie zum Beispiel die Zugkraft des Gewebebandes vorgenommen werden. Alle Steuerelemente wurden speziell für das Projekt Wickelmaschine programmiert und sind für die Bedienung per Touchscreen ausgelegt.

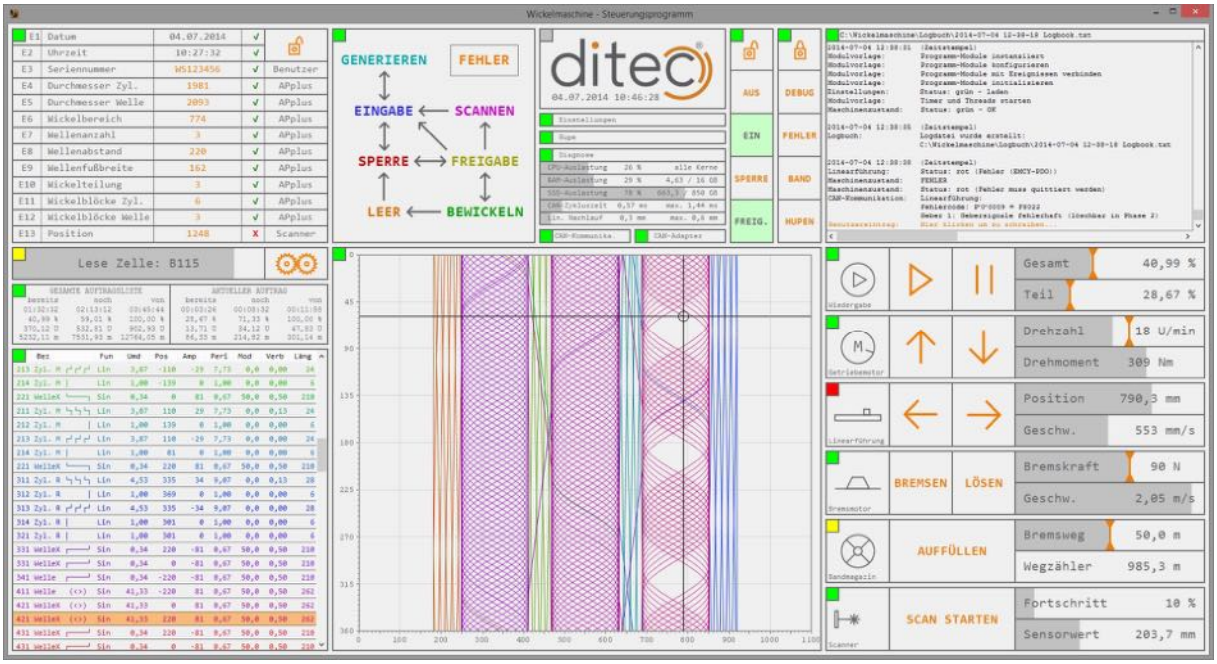


Abb. 1-14: Benutzeroberfläche der Steuerungssoftware

Die Programmierung einer speziell auf den Anwendungsfall ausgelegten Steuerungssoftware ermöglichte die Automatisierung eines komplexen Arbeitsschritts, der bisher nur händisch ausgeführt werden konnte. Aufgrund der Flexibilität der objektorientierten Programmiersprache VB.NET konnten alle Aufgaben zur Steuerung, Überwachung, Protokollierung und Visualisierung des Anlagenprozesses gelöst und in einer Anwendung umgesetzt werden. Dies ermöglichte die erforderliche enge Zusammenarbeit aller rechenintensiven und zeitkritischen Programmmodule. Zusammengefasst konnte durch den Einsatz von VB.NET die Realisierbarkeit einer Anlagensteuerung für einen sehr komplexen Arbeitsschritt hergestellt und erfolgreich umgesetzt werden.

1.4.2 Kugelwaage

Das zweite Projekt ist aus einer Regelungstechnik-Vorlesung heraus entstanden. Das Projektziel war die Konstruktion und Umsetzung eines Testaufbaus, an dem Studenten für Übungszwecke durch das Entwerfen eines Reglers eine sehr instabile Regelstrecke stabilisieren können. Für die Masterarbeit stellt das Projekt ein ideales Beispiel dar, um die Qualität der Echtzeitfähigkeit von VB.NET aufzuzeigen und zu diskutieren.

Die Regelstrecke des Übungsaufbaus besteht aus einem drehbar gelagerten V-Profil, dessen Neigung durch Ändern der Drehzahl und Pitch eines Propellerantriebs verstellt werden kann. Zu regeln ist die Position einer Kugel, die in diesem V-Profil rollt. Hierzu wird die Lage der Aluminium-Kugel über Karbonstäbe nach dem Prinzip eines Spannungsteilers ermittelt und der Drehwinkel des V-Profils mit einem Beschleunigungssensor erfasst.

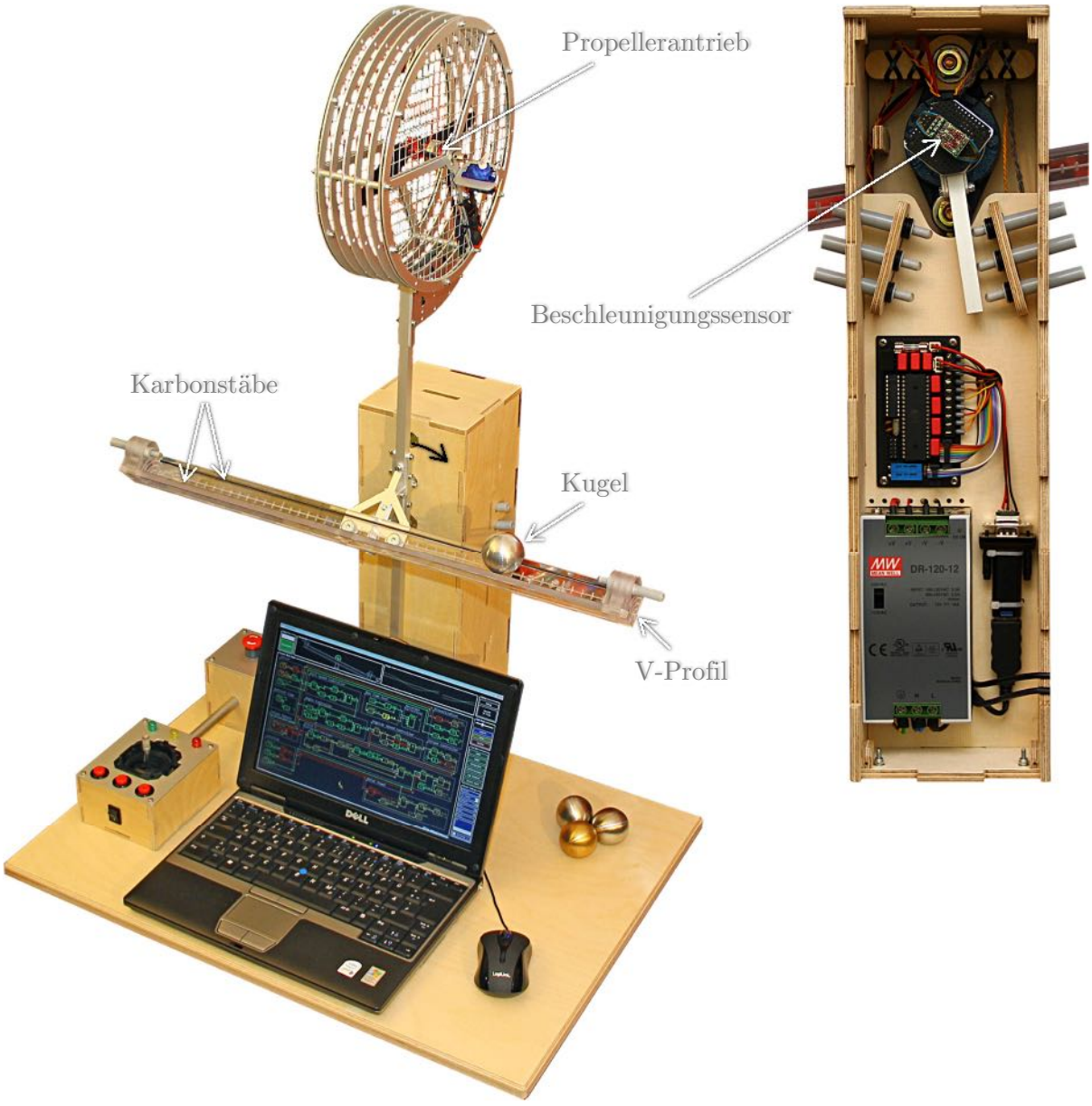


Abb. 1-15: Aufbau der Kugelwaage

Die Regelung des Systems wird von einem in VB.NET geschriebenen Programm übernommen, das auf dem Notebook ausgeführt wird. Hierbei handelt es sich nicht um ein statisches Steuerungsprogramm, sondern um eine Entwicklungsumgebung, in der ein Regler grafisch mit einem Funktionsblock-Editor aufgebaut werden kann. Grundlegende Regler und Zeitglieder können per „Drag and Drop“ eingefügt und mit Signallinien verbunden werden. Zusätzlich steht ein Parser zur Verfügung, mit dem die Funktion der Funktionsblöcke durch mathematische Ausdrücke frei definiert werden kann.

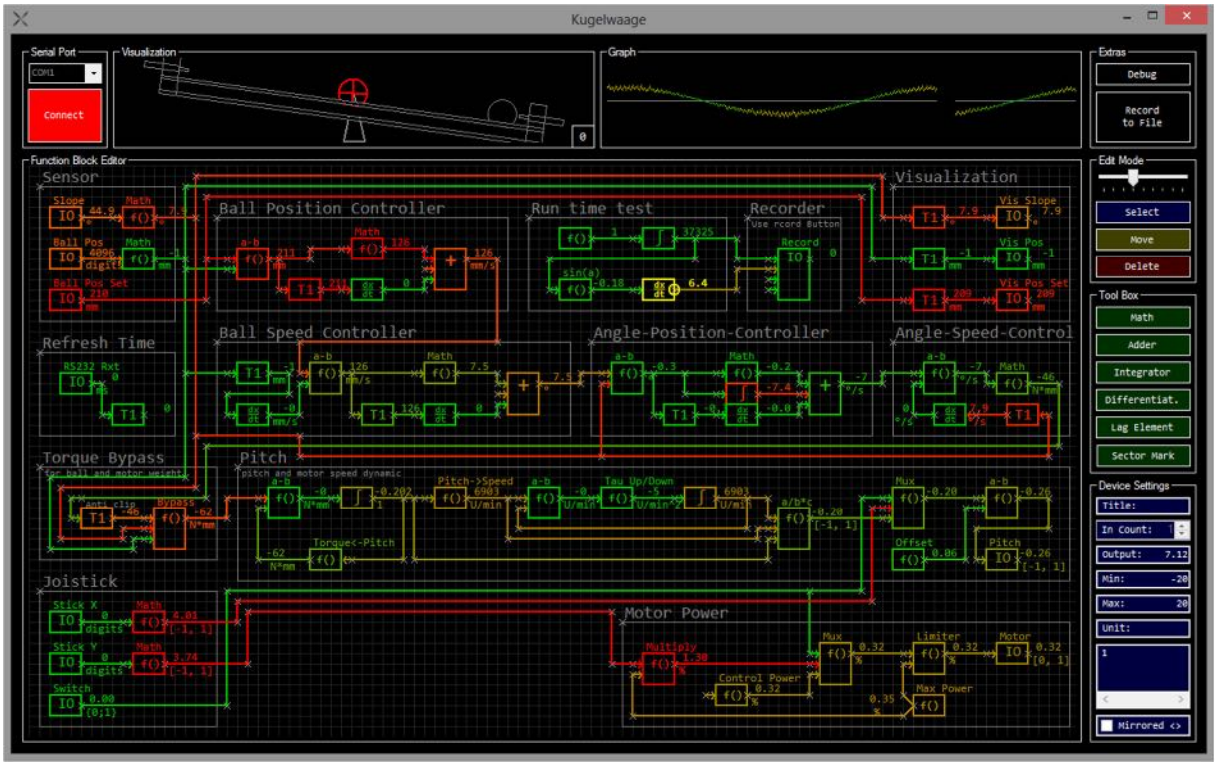


Abb. 1-16: Funktionsblockeditor zum Regeln der Kugelwaage

Eigentlich hätte man für diese Anwendung auch auf Programme wie LabVIEW oder Simulink zurückgreifen können und keine eigene Entwicklungsumgebung programmieren müssen. Der große Vorteil gegenüber LabVIEW und Simulink besteht aber darin, dass die komplette Funktionsblockstruktur während der Laufzeit ohne Unterbrechung verändert werden kann. Zudem werden alle Signallinien zur Laufzeit abhängig von deren Signalwerten farbig dargestellt, um visuell auf Bereiche des Reglers hinzuweisen, in denen Grenzwerte gefährlich nahe erreicht oder überschritten werden. Aufgrund dieser und einiger weiterer Funktionen ist die selbst geschriebene Entwicklungsumgebung ideal für Reglerentwurf-Übungen geeignet.

Kritisch zu betrachten ist die Qualität der Echtzeitfähigkeit. Weil die Software auf einem nicht-echtzeitfähigen Windows-PC ausgeführt wird, schwanken die Zykluszeiten der Regelung zwischen wenigen Mikrosekunden und einer Millisekunde. Für die Realisierung der sehr zeitkritischen Regelung der Kugelwaage war dies mehr als ausreichend; jedoch könnte die unzuverlässige schwankende Zykluszeit für noch zeitkritischere Systeme ein Problem darstellen.

1.5 Problem

Die Entwicklungsumgebung Visual Studio und die Programmiersprache VB.NET bilden zusammen ein mächtiges Werkzeug der Softwaretechnik, mit dem sehr aufwändige und komplexe Windows-Anwendungen geschrieben werden können. Die Projekte aus dem letzten Kapitel zeigen, dass durch den Einsatz von VB.NET in der Automatisierungstechnik neue Möglichkeiten entstehen, die für die Realisierbarkeit von sehr speziellen Anwendungen entscheidend sein können. Dies zeigte das Projekt Wickelmaschine, bei dem mit VB.NET eine Steuerungssoftware umgesetzt wurde, die mit gängigen Automatisierungswerkzeugen in dieser Form nicht zu realisieren gewesen wäre. Das Projekt Kugelwaage zeigte zudem, dass es sogar möglich ist, eine ganze Entwicklungsumgebung zu programmieren, mit der Regler für zeitkritische Systeme erstellt werden können.

Jedoch trat bei den beiden Projekten während der Softwareimplementierung ein zentrales Problem auf: Die Programmiersprache VB.NET und das dazugehörige Framework sind nicht speziell für die Automatisierungstechnik ausgelegt und enthalten deshalb keine grundlegenden Automatisierungsfunktionen wie zum Beispiel Regler und Zeitglieder etc.. Vielmehr mussten solche Standardroutinen selbst implementiert werden. Dies ist sehr zeitaufwändig und hat zur Folge, dass der Programmcode extrem projektspezifisch wird und eine Übertragung von Funktionalitäten in andere Projekte nur bedingt möglich ist.

1.6 Lösung

Um für zukünftige Automatisierungsprojekte die Vorteile von VB.NET und dem Visual Studio noch effektiver nutzen zu können, ist es das Ziel dieses Masterarbeitsprojekts, eine umfangreiche Automatisierungsbibliothek für VB.NET zu erstellen. Die Bibliothek soll das .NET-Framework um grundlegende Funktionalitäten erweitern, die für das Erstellen einer Steuerungssoftware automatisierungstechnischer Anwendungen häufig zum Einsatz kommen. Hierzu gehören zum Beispiel Regler, Signalgeneratoren, Zeitglieder, Flankenerkennungen usw.. Des Weiteren soll eine speziell für diese Automatisierungsbibliothek entwickelte Testumgebung programmiert werden, mit der die Bibliotheksbausteine getestet und die Bibliothek erweitert werden kann.

Um einen ersten Eindruck von der Automatisierungsbibliothek zu bekommen, wird im nächsten Unterkapitel ein Element der Bibliothek näher vorgestellt.

1.7 Vorstellung eines Funktionsblocks der Bibliothek

Im Folgenden wird exemplarisch ein Funktionsblock der Automatisierungsbibliothek, nämlich das „T2S“-Glied aus dem Namensbereich „Controller“ vorgestellt und dessen Implementierung genauer beschrieben. In der Regelungstechnik bezeichnet man ein „T2S“-Glied als ein schwingungsfähiges Übertragungsglied, das eine Verzögerung 2. Ordnung darstellt. Das Funktionsblock-Symbol, mit dem das „T2S“-Glied in der Automatisierungsbibliothek gefunden werden kann, zeigt die Sprungantwort des Übertragungsglieds.

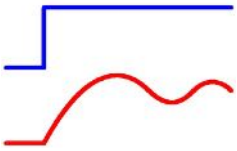


Abb. 1-17: Symbol des „T2S“-Funktionsblocks

In der Elektrotechnik ist das Standardbeispiel für ein „T2S“-Glied der elektronische Schwingkreis aus Widerstand, Spule und Kondensator. Mit der folgenden LTspice-Simulation wurde ein Reihenschwingkreis simuliert und durch Anlegen einer Gleichspannung am Eingang die Sprungantwort am Ausgang aufgezeichnet.

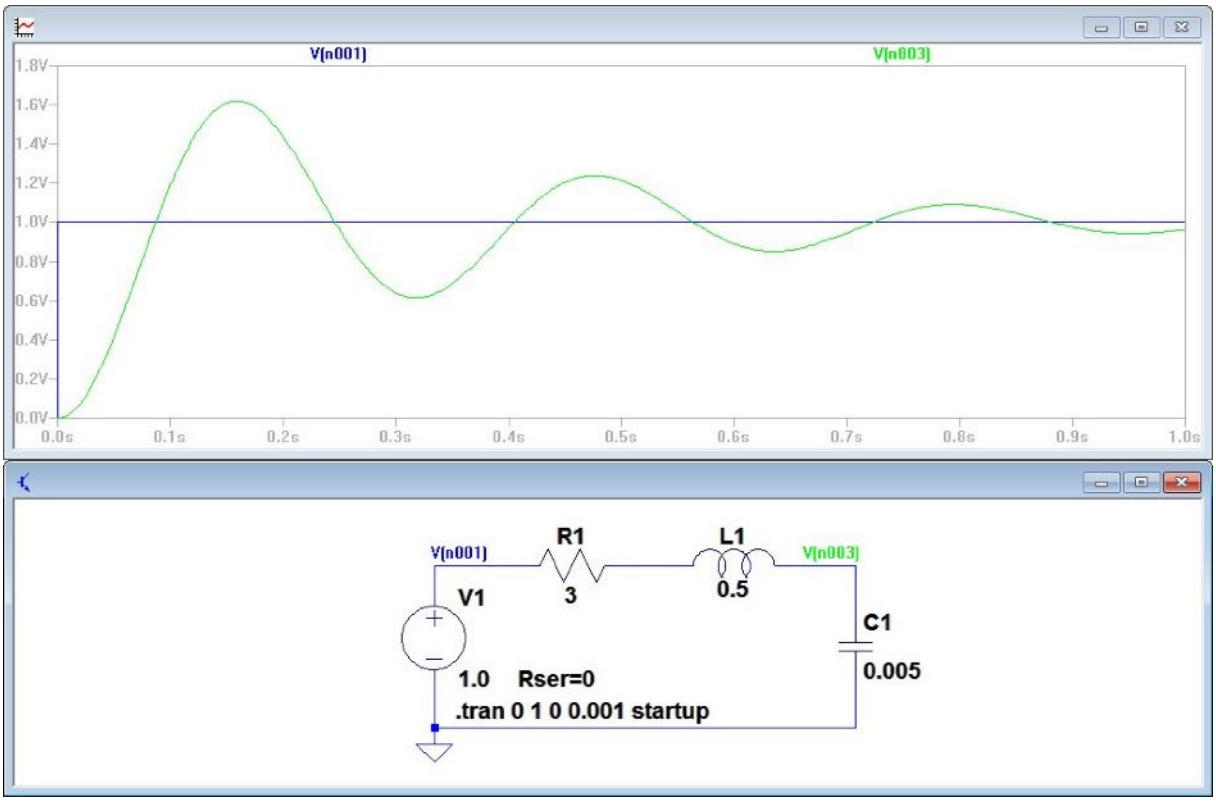


Abb. 1-18: LTspice-Simulation eines Reihenschwingkreises

Im Folgenden wird dieses „T2S“-Schwingungsverhalten zunächst mathematisch beschrieben und anschließend erklärt, wie es programmiert und als Funktionsblock in die Automatisierungsbibliothek aufgenommen wurde.

1.7.1 Mathematische Beschreibung

In der Mathematik wird eine gedämpfte harmonische Schwingung mit einer Kreisfrequenz ω_0 und einem Dämpfungsgrad d durch folgende Differentialgleichung beschrieben:

$$\frac{1}{\omega_0^2} \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) + \frac{2 \cdot d}{\omega_0} \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = u(t)$$

Am Dämpfungsgrad d lässt sich das Schwingungsverhalten eines angeregten Systems ablesen.

$d < 0$	$\Rightarrow$ Aufschwingendes System (instabil)
$d = 0$	$\Rightarrow$ Dauerschwingung (grenzstabil)
$0 < d < 1$	$\Rightarrow$ gedämpfte Schwingung (stabil)
$d = 1$	$\Rightarrow$ gerade kein Überschwingen (stabil, aperiodischer Grenzfall)
$d > 1$	$\Rightarrow$ asymptotische Annäherung (stabil, aperiodische Lösung)

Die Abklingkonstante δ, die Aussage über die Zeitdauer gibt, bis eine Schwingung abgeklungen ist, lässt sich aus der Kreisfrequenz ω_0 und dem Dämpfungsgrad d berechnen.

$$\delta = \omega_0 \cdot d$$

Nach etwa der Zeit $t_{\ddot{u}} = \frac{3}{\delta}$ ist ein System auf 5% der Ausgangsamplitude abgeklungen.

Ein ungedämpftes, frei schwingendes System schwingt in seiner Kennkreisfrequenz ω_0. Das gleiche System mit einer Dämpfung d würde mit der kleineren Eigenkreisfrequenz ω_d schwingen. Sie lässt sich aus der Kennkreisfrequenz ω_0 und der Abklingkonstante δ errechnen.

$$\omega_d = \sqrt{\omega_0^2 - \delta^2}$$

Eine Kreisfrequenz ω kann im Allgemeinen über die Frequenz f und die Periodendauer T einer Schwingung berechnet werden.

$$\omega = 2 \cdot \pi \cdot f = \frac{2 \cdot \pi}{T}$$

Nach der mathematischen Beschreibung des „T2S“-Glieds mit einer Differentialgleichung gilt es nun, das Übertragungsglied mit Programmcode darzustellen. Hierzu werden im nächsten Kapitel die notwendigen Differenzengleichungen ermittelt.

1.7.2 Approximation

Reale Systeme verhalten sich zeitkontinuierlich. Dies bedeutet, dass ein reales System zu jedem Zeitpunkt abhängig von den Eingangswerten aktuelle Ausgangswerte liefert. Ein System, das auf einem Computer simuliert wird, kann nur zu diskreten Zeitpunkten aktuelle Ausgangswerte liefern, weil ein Prozessor nur zyklisch den Programmcode abarbeiten kann. Daraus folgt, dass Systeme mit einem Computer nur näherungsweise simuliert werden können. Die folgende Grafik zeigt die Sprungantworten eines simulierten Schwingkreises. Im oberen Diagramm ist anhand der fallenden Flanken zu erkennen, an welchen Zeitpunkten der Prozessor das simulierte „T2S"-Übertragungsglied berechnet hat. Das Diagramm darunter stellt mit dem blauen Graphen das Eingangssignal dar. Die rote Sprungantwort im gleichen Diagramm zeigt, wie sich der Schwingkreis in der Realität verhalten würde. Der violette, grüne und ockerfarbene Graph stellen die Ergebnisse der drei verschiedenen Simulationen dar, bei denen das „T2S"-Glied mit unterschiedlichen Approximationsverfahren implementiert wurde.

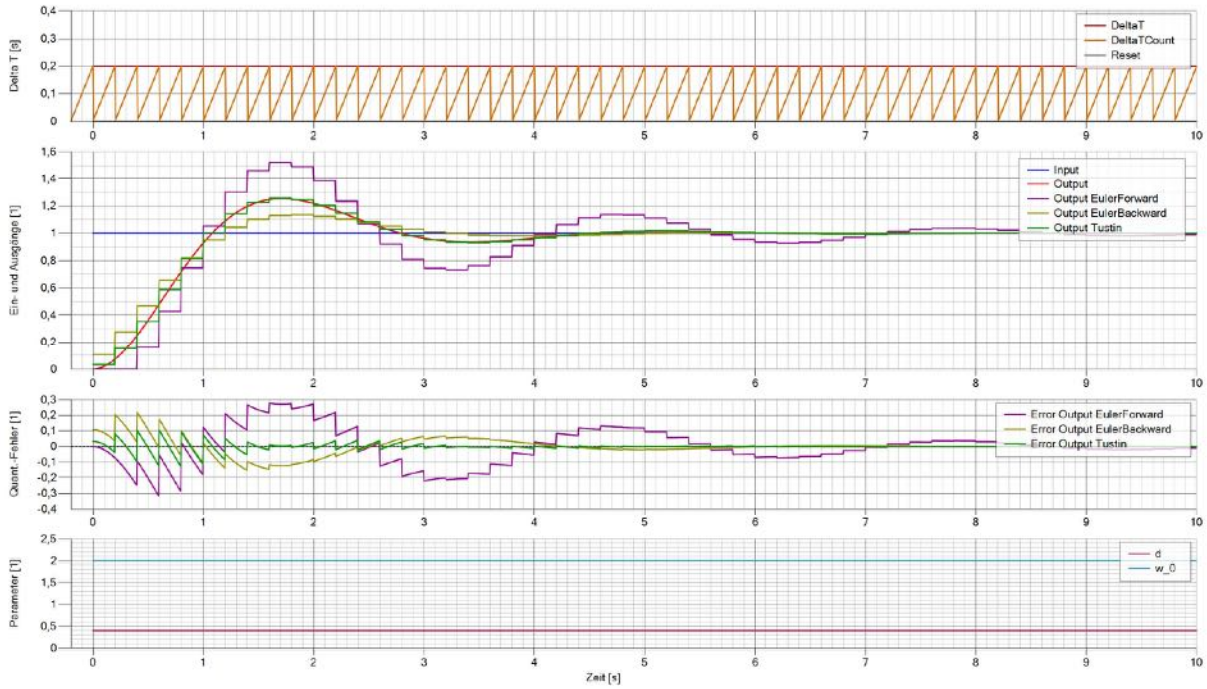


Abb. 1-19: Bildschirmaufnahme der Funktionsblock-Testumgebung

Im vorletzten Diagramm sind die Quantisierungsfehler zu sehen, die die Abweichungen der Simulationsergebnisse gegenüber dem realen Verhalten zeigen. Das letzte Diagramm zeigt die eingestellte Kreisfrequenz ω_0 und den Dämpfungsgrad d an.

Im Folgenden werden die Differenzengleichungen des „T2S"-Glieds ermittelt, damit es anschließend programmiert werden kann.

Um ein System auf einem Computer simulieren zu können, muss es zeitdiskret beschrieben werden. Die bereits ermittelte Differentialgleichung beschreibt das System mit dem Zusammenhang von Ausgangsfunktion $y(t)$ und Eingangsfunktion $u(t)$ zeitkontinuierlich.

$$\frac{1}{\omega_0^2} \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) + \frac{2 \cdot d}{\omega_0} \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = u(t)$$

Für die Programmierung muss nun aus der Differentialgleichung eine Differenzengleichung ermittelt werden, die den Ausgangswert $y(k)$ abhängig von den Eingangswerten $u(k-x)$ und den Ausgangswerten $y(k-x)$ beschreibt. Der Parameter k steht hierbei für die Anzahl der bisherigen Funktionsaufrufe und der Parameter x sagt aus, wie weit auf die vorherigen Ein- und Ausgangswerte zurückgegriffen wird. Beispielsweise werden mit der folgenden Differenzengleichung alle Eingangswerte aufsummiert.

$$y(k) = y(k-1) + u(k)$$

Die hierzu in VB.NET umgesetzte Funktion würde wie folgt aussehen:

```
Public Function y(ByVal u As Double)
    Static LastY As Double
    y = LastY + u
    LastY = y
    Return y
End Function
```

Hierbei handelt es sich um eine sehr einfache Differenzengleichung. Die Herleitung einer wesentlich aufwändigeren ist in der Dokumentation des „PIDT1“-Gliedes ab Seite 55 zu finden.

Nun sollen aber die Differenzengleichungen des „TS2“-Glieds auf Basis der Differentialgleichung bestimmt werden. Hierzu muss zunächst die Differentialgleichung mit Hilfe der Laplace-Transformation in den zeitkontinuierlichen Bildbereich übertragen werden. Die daraus resultierende Übertragungsfunktion des „T2S“-Glieds lässt sich wie folgt schreiben:

$$G(s) = \frac{1}{\frac{1}{\omega_0^2} \cdot s^2 + \frac{2 \cdot d}{\omega_0} \cdot s + 1}$$

Nun kann die Übertragungsfunktion durch einfaches Ersetzen des Parameters s mit den Verfahren Euler-Vorwärts, Euler-Rückwärts oder der Tustin-Methode in den zeitdiskreten Bildbereich übertragen werden. Die verschiedenen Approximationsverfahren wirken sich unterschiedlich auf die durch die Approximation entstehenden Quantisierungsfehler aus.

Euler-Vorwärts	Euler-Rückwärts	Tustin-Methode
$s \to \frac{z-1}{T}$	$s \to \frac{z-1}{Tz}$	$s \to \frac{2}{T} \cdot \frac{z-1}{z+1}$

Das „T2S“-Glied wird im Folgenden mit der Tustin-Methode approximiert.

Der durch die Approximation entstandene Parameter T steht für die Zykluszeit und gibt den Zeitabstand der einzelnen Funktionsaufrufe an.

$$G(z) = \frac{z^2 + 2 \cdot z + 1}{(4 \cdot a + 2 \cdot b + 1) \cdot z^2 - (8 \cdot a - 2) \cdot z + 4 \cdot a - 2 \cdot b + 1}$$

Durch folgende Substitution wurde die Übertragungsfunktion überschaubar gehalten:

$$a = \frac{1}{\omega_0^2 \cdot T^2} \qquad b = \frac{2 \cdot d}{\omega_0 \cdot T}$$

Nun ist es möglich, die Übertragungsfunktion mit Hilfe der z-Transformation zurück in den Wertebereich zu übertragen und somit die gesuchte Differenzengleichung zu erhalten:

$$y(k) = \frac{1}{4 \cdot a + 2 \cdot b + 1} \cdot$$
$$\big((2 \cdot b - 4 \cdot a - 1) \cdot y(k-2) + (8 \cdot a - 2) \cdot y(k-1) + u(k-2) + 2 \cdot u(k-1) + u(k)\big)$$

Mit dieser Differenzengleichung kann das „T2S“-Glied nun programmiert und als Funktionsblock mit in die Automatisierungsbibliothek aufgenommen werden.

1.7.3 Stabilität

Ob sich ein „T2S“-Glied stabil oder instabil verhält, lässt sich für zeitkontinuierliche Systeme anhand der Übertragungsfunkton einfach bestimmen:

$$G(s) = \frac{1}{\frac{1}{\omega_0^2} \cdot s^2 + \frac{2 \cdot d}{\omega_0} \cdot s + 1}$$

Liegen die von den Parametern abhängigen Polstellen der Übertragungsfunktion in der linken s-Halbebene, ist also ihr Realteil negativ, dann verhält sich das Übertragungsglied stabil. Somit ist das reale „T2S“-Glied stabil, wenn die folgenden Bedingungen erfüllt sind:

$$d > 0 \text{ und } \omega_0 > 0$$

Um die Stabilität des approximierten „T2S“-Glieds zu bestimmen, muss die Übertragungsfunktion des zeitdiskreten Bildbereichs betrachtet werden:

$$G(z) = \frac{z^2 + 2 \cdot z + 1}{(4 \cdot a + 2 \cdot b + 1) \cdot z^2 - (8 \cdot a - 2) \cdot z + 4 \cdot a - 2 \cdot b + 1}$$

Um das Stabilitätskriterium hier zu erfüllen, muss der Betrag aller Polstellen < 1 sein. Somit ist das zeitdiskrete System dann stabil, wenn die folgenden Bedingungen erfüllt werden:

$$T < \frac{2}{\omega_0} \text{ und } d > 0 \text{ und } \omega_0 > 0$$

1.7.4 Implementierung

Im Folgenden wird die Klasse „T2S“, mit der das T2S-Glied implementiert wurde, kommentiert. Sie ist in der Automatisierungsbibliothek unter dem Namensbereich „Controler“ zu finden.

```
Public Class T2S
```

Wie alle Regelglieder erbt die Klasse „T2S“ von der Basisklasse „FuncRetrospectiveTimeBased“, die gemeinsame Funktionalitäten enthält. Somit wird doppelter Programmcode vermieden und alle Funktionsblöcke können über eine einheitliche Schnittstelle aufgerufen werden:

```
    Inherits BaseClasses.FuncRetrospectiveTimeBased
```

Die Kreisfrequenz ω_0 und der Dämpfungsgrad d lassen sich direkt über öffentliche Eigenschaften der Klasse einstellen:

```
    Public Property w_0 As Double = 2 'Kreisfrequenz
    Public Property d As Double = 0.5 'Dämpfung
```

Folgende Funktion wird von der Oberklasse aufgerufen und berechnet je nach gewähltem Approximationsverfahren die dazugehörige Differenzengleichung. Hierzu stellt die Oberklasse die Zykluszeit T als privates Attribut und die Ein- und Ausgangswerte $y(k-x)$ und $u(k-x)$ vorheriger Funktionsaufrufe über eine Schieberegister-Funktion zur Verfügung:

```
    Protected Overrides Function Functionality() As Double
        Dim a As Double = 1 / (w_0 ^ 2 * T ^ 2)
        Dim b As Double = (2 * d) / (w_0 * T)
        Select Case ApproxType
            Case BaseClasses.ApproxTypeEnum.Forward
                Return 1 / a * ((b - a - 1) * y(k - 2) + (2 * a - b) * y(k - 1) + u(k - 2))
            Case BaseClasses.ApproxTypeEnum.Backward
                Return 1 / (a + b + 1) * ((2 * a + b) * y(k - 1) - a * y(k - 2) + u(k))
            Case BaseClasses.ApproxTypeEnum.SmalestError
                Return 1 / (4 * a + 2 * b + 1) * ((2 * b - 4 * a - 1) * y(k - 2) + (8 * a - 2) *
                       y(k - 1) + u(k - 2) + 2 * u(k - 1) + u(k))
        End Select
        Return Nothing
    End Function
```

Die folgenden Funktionen rechnen anhand der eingestellten Parameter ω_0 und d die Eigenkreisfrequenz ω_d und die Abklingkonstante δ aus und geben diese zurück:

```
    Public Function Get_w_d() As Double 'Eigenkreisfrequenz
        Return Math.Sqrt(w_0 ^ 2 - d ^ 2)
    End Function

    Public Function Get_delta() As Double 'Abklingkonstante
        Return w_0 * d
    End Function
```

Die letzte Funktion berechnet, ab welcher Zykluszeit T der Funktionsblock instabil wird:

```
    Public Function Get_StabilityLimitOfDeltaT() As Double 'Maximale Zykluszeit für Stabilität
        Return w_0 / 2
    End Function
```

Ende der Klasse „T2S“:

```
End Class
```

1.7.5 Anwendungsbeispiele

Ein gutes Beispiel für ein System, dessen Verhalten sich mit einem „T2S“-Glied simulieren lässt, ist aus dem Bereich Elektrotechnik der Reihenschwingkreis:

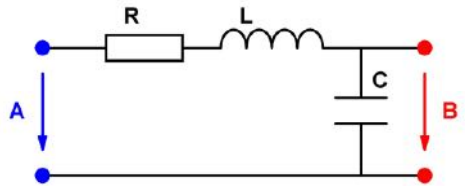


Abb. 1-20: Elektronischer Schwingkreis

Die Kreisfrequenz ω_0 und Dämpfungsgrad d für das „T2S“-Glied lassen sich aus den Bauelementwerten Widerstand R, Spule L und Kondensator C ableiten. Hierfür wird zunächst die Übertragungsfunktion des Schwingkreises über den Spannungsteiler-Ansatz aufgestellt:

$$G(s) = \frac{Y(s)}{\mathrm{U(s)}} = \frac{\mathrm{B(s)}}{\mathrm{A(s)}} = \frac{\frac{1}{s \cdot C}}{R + s \cdot L + \frac{1}{s \cdot C}} = \frac{1}{L \cdot C \cdot s^2 + R \cdot C \cdot s + L1}$$

Setzt man nun die Übertragungsfunktion eines „T2S“-Glieds mit der Übertragungsfunktion des Schwingkreises gleich, erkennt man, dass sie sich lediglich in ihren Koeffizienten unterscheiden:

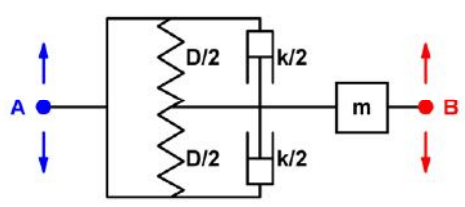
$$\frac{1}{L \cdot C \cdot s^2 + R \cdot C \cdot s + L1} = \frac{1}{\frac{1}{\omega_0^2} \cdot s^2 + \frac{2 \cdot d}{\omega_0} \cdot s + 1}$$

Mit einem Koeffizientenvergleich ist es nun möglich, aus den Bauelementwerten direkt die Parameter Kreisfrequenz ω_0 und Dämpfungsgrad d des „T2S“-Glieds zu bestimmen:

$$L \cdot C = \frac{1}{\omega_0^2} \quad \Rightarrow \quad \omega_0 = \sqrt{\frac{1}{L \cdot C}}$$

$$R \cdot C = \frac{2 \cdot d}{\omega_0} \quad \Rightarrow \quad d = \frac{1}{2} \cdot R \cdot \sqrt{\frac{C}{L}}$$

Analog zu diesem Beispiel lässt sich auch ein Beispiel aus der Mechanik finden. Eine an Federn und Dämpfern aufgehängte Masse bildet ebenfalls ein schwingungsfähiges System, das sich mit einem „T2S“-Glied beschreiben lässt. Die Kreisfrequenz ω_0 und der Dämpfungsgrad d wurden hierfür nach dem gleichen Prinzip, wie im Beispiel am Schwingkreis beschrieben, ermittelt:



$$\omega_0 = \sqrt{\frac{D}{m}}$$

$$d = \frac{1}{2} \cdot k \cdot \sqrt{\frac{1}{m \cdot D}}$$

Abb. 1-21: Schwingende Masse

2 Aufbau der Bibliothek

Die Automatisierungsbibliothek beinhaltet 50 Funktionsblöcke, die in 8 Namensbereiche untergliedert sind. Jede Funktionsblock-Klasse ist von einer Basisklasse abgeleitet und erbt deren Parameter. Zudem kann ein Funktionsblock über beliebig viele zusätzliche Parameter verfügen. In der folgenden Grafik sind die Namensbereiche, Basisklassen und Funktionsblock-Klassen sowie die dazugehörigen Parameter dargestellt.

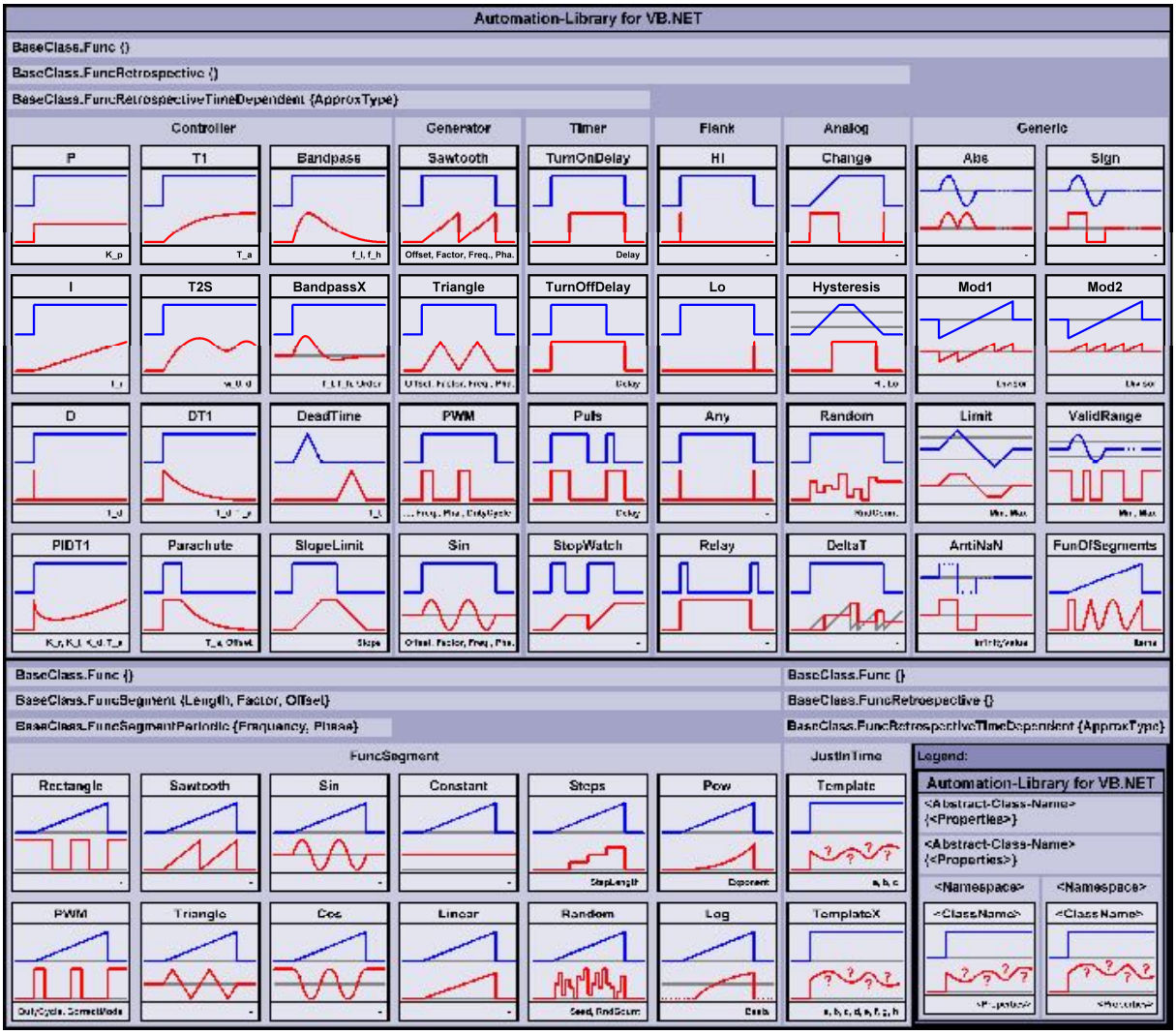


Abb. 2-1: Aufbau der Automatisierungsbibliothek

Die Legende in der Abbildung rechts unten beschreibt die Zuordnung der Beschriftungen. Beispielsweise ist der Funktionsblock mit dem Klassennamen „T2S“ im Namensbereich „Controller“ zu finden und verfügt über die Parameter „w_0“ und „d“. Zusätzlich erbt der Funktionsblock den weiteren Parameter „ApproxType“ seiner abstrakten Oberklasse „FuncRetrospectiveTimeDependent“, die von den Klassen „FuncRetrospective“ und „Func“ abgeleitet ist.

2.1 Basisklassen (BaseClass)

Funktionsblöcke des gleichen Typs sind in der Automatisierungsbibliothek in einem Namensbereich zusammengefasst und werden von derselben Basisklasse abgeleitet. Somit müssen identische Eigenschaften, Funktionen und Methoden nur einmal definiert werden, was die Wartbarkeit und Wiederverwendbarkeit der Bibliothek deutlich erhöht. Zudem verfügen die Funktionsbausteine über einheitliche Schnittstellen und können in spätere Projekte austauschbar eingebunden werden. Auch die für die Bibliothek entwickelte Testumgebung stützt sich auf diese Basisklassen, um alle Funktionsblöcke mit einer einheitlichen Schnittstelle zu testen. Alle Basisklassen sind im Namensbereich „BaseClass“ definiert und in der Abbildung mit ihren Ableitungen dargestellt.

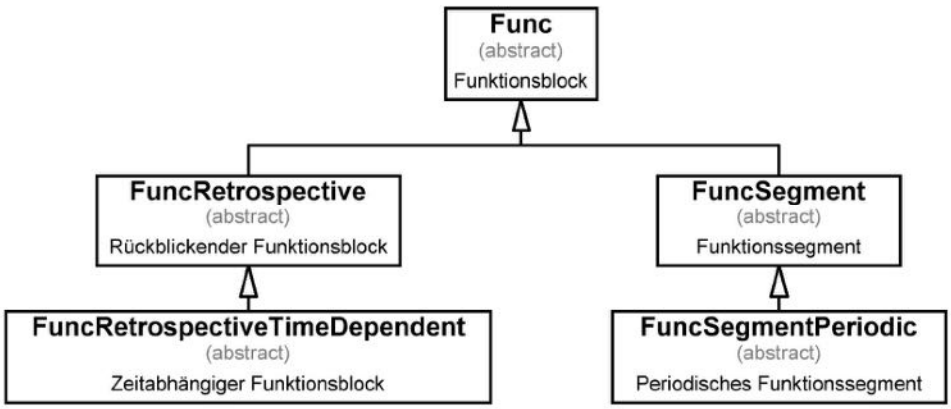


Abb. 2-2: Klassenstruktur aller Basisklassen

In den folgenden Unterkapiteln werden die Eigenschaften, Funktionen und Methoden aller Basisklassen durch Beschreibungen und Kommentare in Programmcodes näher erläutert.

2.1.1 Funktionsblock

Die erste abstrakte Klasse „Func“ enthält ausschließlich eine Funktion mit dem Namen „Output“ und vererbt diese an alle Funktionsblock-Klassen der Automatisierungsbibliothek. Somit verfügt jeder Funktionsblock über eine Ausgangsfunktion. Durch einen Aufruf dieser Funktion kann dem Funktionsbaustein ein Eingangswert übergeben werden, worauf der Funktionsblock seinen Ausgangswert berechnet und diesen zurückgibt.

```
'Funktionsblock, der über eine Ausgangsfunktion verfügt.
Public MustInherit Class Func

    'Ausgangsfunktion
    MustOverride Function Output(Input As Double) As Double

End Class
```

In den nächsten beiden Unterkapiteln werden die abstrakten Basisklassen des linken Zweigs der oben dargestellten Klassenstruktur erläutert.

2.1.2 Rückblickender Funktionsblock

Viele Funktionsblöcke greifen in der Ausgangsfunktion „Output“ für die Berechnung des neuen Ausgangswerts auf die Ein- und Ausgangswerte vorheriger Funktionsaufrufe zurück. Für solche Funktionsblock-Typen erweitert die Oberklasse „FuncRetrospective“ die bereits vorgestellte Klasse „Func“ um ein Schieberegister, in dem die Ein- und Ausgangswerte der letzten zwei Aufrufe gespeichert werden.

In allen Funktionsblöcken, die von der Basisklasse „FuncRetrospective“ abgeleitet werden, kann über die Funktion „u“ auf die aktuellen, letzten sowie vorletzten Eingangswerte und mit der Funktion „y“ auf die Ausgangswerte zugegriffen werden. Zum Beispiel gibt der Ausdruck „u(k-1)“ den letzten Eingangswert und „y(k-2)“ den vorletzten Ausgangswert zurück. Die Konstante „k“, die mit dem Wert „0“ initialisiert ist, ermöglicht es, die übliche Schreibweise einer Differenzengleichung auch im Programmcode verwenden zu können.

Die vererbte Funktion „Output“ bedient das Schieberegister. Deshalb ist zum Definieren der Ausgangsfunktionen für abgeleitete Klassen die Funktion „Functionality“ vorgesehen, die von Unterklassen überschrieben werden muss. Über die Methode „Reset“ und „SetInitialConditions“ kann das Schieberegister zurückgesetzt bzw. mit Anfangswerten belegt werden.

```
'Funktionsblock, der sich auf vorherige Ein- und Ausgangswerte bezieht.
Public MustInherit Class FuncRetrospective
    Inherits Func 'Vererbung

    'Zugriff auf Ein- und Ausgangswerte vorheriger Funktionsblockaufrufe ermöglichen
    Protected Const k As Integer = 0 'Für Schreibweise z. B. u(k-2)
    Protected Const Length As Integer = 2 'Schieberegister-Größe
    Private _u(0 To Length) As Double 'Eingangswerte
    Private _y(0 To Length) As Double 'Ausgangswerte
    Delegate Function FuncDelegate(ByVal i As Integer) As Double 'Schreibweise z. B. u(k-2)
    Protected u As FuncDelegate = Function(i As Integer) _u(-i) 'Schieberegister für Eingangswerte
    Protected y As FuncDelegate = Function(i As Integer) _y(-i) 'Schieberegister für Ausgangswerte

    'Ausgangsfunktion
    Public Overrides Function Output(ByVal Input As Double) As Double
        _u(k) = Input 'Eingangswert in Schieberegister speichern
        _y(k) = Double.NaN 'Ausgangswert ist noch nicht definiert
        _y(k) = Functionality() 'Ausgangswert berechnen
        For i As Integer = Length To 1 Step -1 'Werte des Schieberegisters verschieben
            _u(k + i) = _u(k + i - 1) 'Schieberegister Eingang
            _y(k + i) = _y(k + i - 1) 'Schieberegister Ausgang
        Next
        Return _y(k) 'Ausgangswert zurückgeben
    End Function

    'Definition der Funktionalität des Funktionsblocks (z. B. mit einer Differenzengleichung)
     Protected MustOverride Function Functionality() As Double

     'Schieberegister zurücksetzen
     Public Overridable Sub Reset()
        Array.Clear(_u, 0, _u.Length) : Array.Clear(_y, 0, _y.Length)
     End Sub

     'Anfangsbedingungen setzen
     Public Sub SetInitialConditions(ByVal u As Double(), ByVal y As Double())
        Reset() 'Schieberegister zurücksetzen
        u.CopyTo(_u, 0) : y.CopyTo(_y, 0) 'Werte der Anfangsbedingungen übernehmen
     End Sub

End Class
```

2.1.3 Zeitabhängiger Funktionsblock

Funktionsbausteine, deren Ausgangswerte von der Zeit abhängig sind, werden von der abstrakten Basisklasse „FuncRetrospectiveTimeDependent“ abgeleitet. Sie ist selbst eine Unterklasse von „FuncRetrospective“ und erweitert die Funktion „Output“ um den Parameter „DeltaT“. Dieser Wert wird für den Zugriff aus abgeleiteten Klassen im Parameter „T“ gespeichert, um ihn als Zykluszeit in Differenzengleichungen einsetzen zu können.

Von der Zykluszeit abhängige Übertragungsfunktionen können meist auf verschiedene Art und Weise approximiert bzw. umgesetzt werden. Die verschiedenen Approximations-Typen eines Funktionsblocks sind über eine Aufzählung „ApproxType“ einstellbar. Die Basisklasse definiert hierzu eine Enumeration, die von den abgeleiteten Klassen implementiert und je nach Funktionsblock mit den unterstützten Approximations-Typen belegt werden muss.

```
'Funktionsblock, der sich auf vorherige Ein- und Ausgangswerte bezieht und zeitabhängig ist.
Public MustInherit Class FuncRetrospectiveTimeDependent
    Inherits FuncRetrospective 'Vererbung

    Protected Property T As Double 'DeltaT
    Public MustOverride Property ApproxType As [Enum] 'Näherungsverfahren

    'Ausgangsfunktion
    Public Overloads Function Output(ByVal Input As Double, ByVal DeltaT As Double) As Double
        T = DeltaT 'DeltaT speichern
        Return MyBase.Output(Input)
    End Function

End Class
```

Die folgende Klasse soll als Beispiel für die Implementierung eines zeitabhängigen Funktionsblocks dienen und das bereits Erklärte veranschaulichen.

```
'Verzögerungsglied 2.Ordnung
Public Class T2S
    Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin 'Standard Approximationstyp
    Public Enum ApproxTypeEnum 'Unterstützte Approximationstypen
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property w_0 As Double = 2 'Parameter Kreisfrequenz
    Public Property d As Double = 0.5 'Parameter Dämpfung

    Protected Overrides Function Functionality() As Double 'Funktionalität des Funktionsblocks
        Dim a As Double = 1 / (w_0 ^ 2 * T ^ 2) 'Substitution
        Dim b As Double = (2 * d) / (w_0 * T) 'Substitution
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / a * ((b - a - 1) * y(k - 2) + (2 * a - b) * y(k - 1) + u(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (a + b + 1) * ((2 * a + b) * y(k - 1) - a * y(k - 2) + u(k))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (4 * a + 2 * b + 1) * ((2 * b - 4 * a - 1) * y(k - 2) + (8 * a - 2) * _
                       y(k - 1) + u(k - 2) + 2 * u(k - 1) + u(k))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

2.1.4 Funktionssegment

Die abstrakte Klasse „FuncSegment“ ist eine direkt abgeleitete Klasse der Oberklasse „Func“. Sie ist im rechten Zweig der Klassenstruktur (Seite 31) zu finden. Alle Funktionsblöcke, die von dieser Klasse erben, sind Funktionssegmente und können mit Hilfe des Funktionsblocks „FuncOfSegments“ (Seite 131) zu einer abschnittsweise definierten Funktion aneinandergefügt werden. Dabei legt der Parameter „Length“ jeweils die Länge der Funktionssegmente fest.

Die mathematische Funktion, die ein Funktionssegment repräsentieren soll, muss bei der Vererbung der Basisklasse „FuncSegment“ durch Überschreiben der Funktion „RawFunc“ in der Unterklasse definiert werden. Das Skalieren und Verschieben dieser Funktion ist in Y-Richtung über die Parameter „Factor“ bzw. „Offset“ möglich und wird direkt von der Oberklasse in der Ausgangsfunktion „Output“ durchgeführt.

```
'Funktionsblock, der eine mathematische Funktion in einem Definitionsbereich beschreibt.
Public MustInherit Class FuncSegment
    Inherits Func 'Vererbung

    Property Length As Double = 10 'Definitionslänge auf der X-Achse ([0; Length])
    Property Factor As Double = 1 'Streckung der Funktion in Y-Richtung
    Property Offset As Double = 0 'Verschiebung der Funktion in Y-Richtung

    'Ausgangsfunktion
    Public Overrides Function Output(Input As Double) As Double
        'Nur Definitionsbereich [0; Length] zulassen
        If Not (Input >= 0 And Input <= Length) Then Return Double.NaN
        'Streckung und Verschiebung der mathematischen Funktion
        Return RawFunc(Input) * Factor + Offset
    End Function

    'Mathematische Funktion
    Protected MustOverride Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])

End Class
```

Das folgende Beispiel zeigt den Programmcode des Funktionssegments „Steps“ (Seite 139), der eine mathematische Stufenfunktion beschreibt. Durch die Vererbung der Oberklasse „FuncSegment“ müssen im Funktionssegment nur die mathematische Funktion selbst und funktionsspezifische Parameter definiert werden. Das Skalieren, Verschieben und Beschneiden der Funktion auf einen Definitionsbereich wird von der Basisklasse übernommen.

```
'Stufenfunktion
Public Class Steps
    Inherits BaseClass.FuncSegment 'Vererbung

    Property StepLength As Double = 1 'Stufenlänge

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return Math.Truncate(Input / StepLength)
    End Function

End Class
```

2.1.5 Periodisches Funktionssegment

Die Basisklasse „FuncSegmentPeriodic“ ist von der zuletzt beschriebenen abstrakten Klasse „FuncSegment“ abgeleitet und für das Erstellen von Funktionssegmenten ausgelegt, die periodische mathematische Funktionen beschreiben. Klassen, die von dieser Oberklasse erben, müssen lediglich die Funktion „RawFuncOnePeriod“ mit einer Funktion überschreiben, in der eine Periode der mathematischen Funktion definiert ist. In der Funktion „RawFunc“ der Basisklasse wird diese Periode dann in eine unendliche periodische Funktion umgewandelt, wobei die Frequenz und Phasenlage mit einberechnet und über die Parameter „Frequency“ bzw. „Phase“ festgelegt werden.

```
'Funktionsblock, der eine periodische mathematische Funktion beschreibt.
Public MustInherit Class FuncSegmentPeriodic
    Inherits FuncSegment 'Vererbung

    Property Frequency As Double = 1 'Frequenz in Hz (]-∞; ∞[)
    Property Phase As Double = 0 'Phase in 1 = 360° (]-∞; ∞[)

    'Mathematische Funktion (wird von der Basisklasse in Y-Richtung skaliert und verschoben)
    Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0 to ∞])
        If Double.IsInfinity(Input) Then Return Double.NaN 'Unendlich -> NaN
        Input = Input * Frequency + Phase 'RawFunc in X-Richtung strecken und verschieben
        Input = Input Mod 1 'Eingangswert in Definitionsbereich ]-1 to 1[ bringen
        If Input < 0 Then Input += 1 'Eingangswert in den Definitionsbereich [0 to 1[ bringen (Mod2)
        Return RawFuncOnePeriod(Input)
    End Function

    'Eine Periode der mathematischen Funktion (in X- und Y-Richtung weder skaliert noch verschoben)
    Protected MustOverride Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)

End Class
```

Das folgende Beispiel zeigt die Implementierung des periodischen Funktionssegments „PWM“ (Seite 149) aus der Automatisierungsbibliothek. Wie zu erkennen ist, müssen auch hier in der Funktionsblock-Klasse nur die mathematische Funktion selbst und die funktionsspezifischen Parameter definiert werden. Das Skalieren und Verschieben der mathematischen Funktion in Richtung der X- und Y-Achse wird von den Basisklassen übernommen, die hierzu die Parameter „Factor“, „Offset“, „Frequency“ und „Phase“ bereitstellen. Der Entwickler kann sich somit beim Programmieren eines periodischen Funktions-Segments gezielt auf das Beschreiben einer Funktions-Periode konzentrieren und muss diese weder skalieren noch verschieben.

```
'Pulsweitenmoduliertes Rechtecksignal
'#### PWM-Funktion ####
Public Class PWM
    Inherits BaseClass.FuncSegmentPeriodic 'Vererbung

    Property DutyCycle As Double = 0.2 'Tastgrad (Verhältnis Impulsdauer zu Periodendauer)
    Property CorrectMode As Boolean = False 'frequenz- und phasenrichtig

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        If CorrectMode Then 'Frequenz- und phasenrichtige Ausgabe?
            Return If((Convert.ToSingle(Input) < Convert.ToSingle(DutyCycle / 2)) Or
                      (Convert.ToSingle(Input) >= Convert.ToSingle(1 - DutyCycle / 2)), 1, 0)
        Else
            Return If(Input < DutyCycle, 1, 0)
        End If
    End Function
End Class
```

2.2 Implementierung eines Funktionsblocks

Es gibt zwei verschiedene Möglichkeiten, einen Funktionsblock in spätere Softwareprojekte einzubinden, seine Parameter zu setzen und die Ausgangsfunktion aufzurufen. Die folgenden Beispiele erklären beide Methoden. Es werden jeweils mit einem anderen Programmcode Ausgangswerte (TrackBar2) über ein schwingungsfähiges T2S-Glied aus den Eingangswerten (TrackBar1) generiert. Je nachdem, ob eine schnelle Laufzeit oder wenig Codezeilen gefordert sind, ist die entsprechende Möglichkeit bei der Einbindung von Funktionsbausteinen zu wählen.

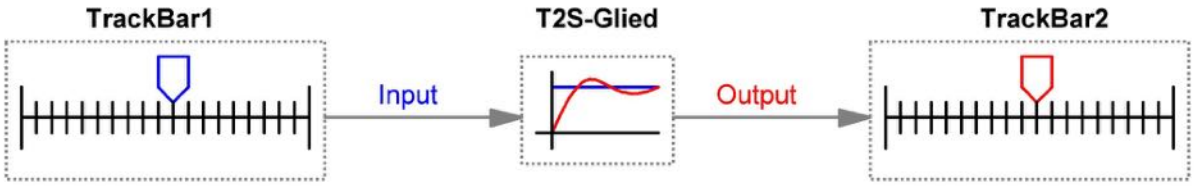


Abb. 2-3: Signalflussdiagramm der Beispiel-Anwendung

Möglichkeit 1: Instanz selbst erstellen

Bei der ersten Methode wird die Instanz des Funktionsblocks selbst erstellt und über diese direkt auf die Parameter und die Ausgangsfunktion des Funktionsblocks zugegriffen. Hierzu sind mehrere Zeilen Programmcode nötig, wobei diese Methode die bessere Laufzeit bietet.

```
Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick 'Zeitgeber für Zyklus

    'Funktionsblock-Instanz erstellen (statisch)
    Static T2S As New Controller.T2S 'Schwingungsfähiges Übertragungsglied

    'Funktionsblock-Parameter aktualisieren (zyklisch)
    T2S.w_0 = 2 'Kreisfrequenz
    T2S.d = 0.4 'Dämpfung

    'Ausgangsfunktion des Funktionsblocks aufrufen (zyklisch)
    'Output         = Function  (Input          , DeltaT)
    TrackBar2.Value = T2S.Output(TrackBar1.Value, 0.01)

End Sub
```

Möglichkeit 2: Instanz per Typ und Id suchen

Bei der zweiten Möglichkeit werden mit nur einem Aufruf der Funktion „CallInst“ der Funktionsblock instanziiert, seine Parameter gesetzt und die Ausgangsfunktion aufgerufen. Um bei weiteren Aufrufen die richtige Instanz anzusprechen, muss der Funktion bei Aufruf der gewünschte Funktionsblock-Typ und eine Id übergeben werden. Des Weiteren sind der Eingangswert und die Parameter zu übergeben. Das Anfügen der Zykluszeit ist optional. Dieser Funktionsaufruf nimmt nur eine Codezeile in Anspruch, ist aber rechenaufwändiger, weil bei jedem Aufruf die Funktionsblock-Instanz anhand des Typs und der Id gesucht werden muss.

```
Private Sub Timer2_Tick(sender As Object, e As EventArgs) Handles Timer2.Tick 'Zeitgeber für Zyklus

    'Ausgangsfunktion des Funktionsblocks anhand seines Typs und einer Id aufrufen
    'Output         = Function(TypeKey                , Id   , Input          , Parameter    , DeltaT)
    TrackBar2.Value = CallInst(GetType(Controller.T2S), "123", TrackBar1.Value, "d=0.4,w_0=2", 0.01)

End Sub
```

Programmcode der Funktion „CallInst“

Die Funktion „CallInst“ zum Aufrufen von Funktionsblöcken in der Automatisierungsbibliothek ist unter dem Modul „Utility“ zu finden. Der Programmcode der Funktion wird im Folgenden mit Kommentaren erklärt.

```
'Ruft die Instanz eines Funktionsblocks anhand seines Typs und einer Id auf.
Public Function CallInst(ByVal TypeKey As Type, ByVal Id As Object, ByVal Input As Double,
                         ByVal Paramter As String, Optional DeltaT As Double = Double.NaN) As Double
    'Beispielaufruf:
    'TrackBar2.Value = CallInst(GetType(Controller.T2S), "123", TrackBar1.Value, "w_0=1,d=0.2", 0.01)
    'TypeKey:   Funktionsblock-Typ                z. B. GetType(Controller.T2S)
    'Id:        Id zum Zuordnen der Instanzen     z. B. "123"
    'Input:     Eingangswert des Funktionsblocks  z. B. 5.4
    'Parameter: Parameter als String              z. B. "w_0=1,d=0.2"
    'DeltaT:    Nur bei zeitabh. Funktionsblöcken z. B. 0.01
    'return:    Ausgangswert des Funktionsblocks  z. B. 1.8

    'Die Instanzen der Funktionsblöcke werden in zwei ineinander verschachtelten Verzeichnissen
    '(Dictionarys) gespeichert. Das Äußere ordnet den Funktionsblock-Typ und das Innere die Id zu.

    'Instanz des äußeren Verzeichnisses erstellen
    Static TypeDictionarys As Dictionary(Of Type, Dictionary(Of Object, BaseClass.Func))
    If TypeDictionarys Is Nothing Then
        TypeDictionarys = New Dictionary(Of Type, Dictionary(Of Object, BaseClass.Func))
    End If
    'Für jeden Funktionsblock-Typ ein eigenes Unterverzeichnis anlegen
    If Not TypeDictionarys.ContainsKey(TypeKey) Then 'Unterverzeichnis noch nicht vorhanden?
        TypeDictionarys.Add(TypeKey, New Dictionary(Of Object, BaseClass.Func))
    End If
    'Funktionsblock-Verzeichnis mit passendem Funktionsblock-Typ suchen
    Dim IdDictionary As Dictionary(Of Object, BaseClass.Func) = TypeDictionarys(TypeKey)
    'Für jede Id eine eigene Funktionsblock-Instanz erstellen.
    If Not IdDictionary.ContainsKey(Id) Then 'Instanz zu dieser Id noch nicht vorhanden?
        IdDictionary.Add(Id, DirectCast(Activator.CreateInstance(TypeKey), BaseClass.Func))
    End If
    'Funktionsblock-Instanz mit passender Id suchen
    Dim Instance As BaseClass.Func = IdDictionary(Id)
    'Parameter des Funktionsblocks aktualisieren
    Dim Parameter As String() = Paramter.Replace(" ", "").Split(","c) 'Parameter-Zeichenkette teilen
    For Each ParameterStr As String In Parameter 'Alle Parameter durchlaufen
        Dim ParameterElements As String() = ParameterStr.Split("="c) 'Name und Wert trennen
        Dim ParameterName As String = ParameterElements(0) 'Name
        Dim ParameterValue As Double = Double.Parse(ParameterElements(1).Replace(".", ",")) 'Wert
        'Parameter (Property) setzen (per Reflection)
        CallByName(Instance, ParameterName, CallType.Set, New Object() {ParameterValue})
    Next
    'Aufruf der Funktion Output
    Dim OutputValue As Double = Double.NaN 'Funktionsblock-Ausgangswert
    Dim FuncOutputParameter As Object() = {Input} 'Parameter der Funktion "Output"
    If TypeOf Instance Is BaseClass.FuncRetrospectiveTimeDependent Then
        FuncOutputParameter = {Input, DeltaT} 'Bei zeitabhängigen Funktionsblöcken DeltaT mit übergeben
    End If
    'Aufruf der Funktion "Output" (per Reflection)
    OutputValue = Convert.ToDouble(CallByName(Instance, "Output", CallType.Method, _
                                   FuncOutputParameter))
    Return OutputValue 'Ausgangswert des Funktionsblocks zurückgeben

End Function
```

2.3 Hinweise zur Implementierung

Die folgenden Hinweise sind bei der Implementierung von Elementen der Automatisierungsbibliothek in zukünftige Projekte zu beachten:

Logische Funktionsbausteine arbeiten mit Gleitkommazahlen.
Beim Aufrufen der Ausgangsfunktionen wird bei allen Funktionsblöcken der Bibliothek für die Übergabe des Eingangswerts bzw. die Rückgabe des Ausgangswerts der Datentyp „Double“ verwendet. Dies ist zum Beispiel auch bei Zeitgliedern und Flankenerkennungen, die logische Signale verarbeiten, der Fall. Für die Bibliothek wurde deshalb die Konvention festgelegt, dass die Double-Werte „0“ und „NaN“ als logisches „False“ und alle anderen Werte einschließlich „+∞“ und „-∞“ als logisches „True“ interpretiert werden. Im Umkehrschluss gibt ein Funktionsblock als Ausgangswert eine „0“ für „False“ und eine „1“ für „True“ zurück. Außerdem wird auch bei logischen Funktionsbausteinen der Wert „NaN“ ausgegeben, wenn der Eingangswert oder ein Parameterwert nicht plausibel ist.

Wertänderungen werden auch als Flanken interpretiert.
Einige Funktionsbausteine der Bibliothek detektieren Flanken in Signalen des Typs „Double“. Dabei werden alle Wertänderungen in positiver Richtung als positive Flanke und Wertänderungen in negativer Richtung als negative Flanke erkannt. Dies gilt auch dann, wenn die Gleitkommazahl den Wert „+∞“ oder „-∞“ einnimmt oder verlässt. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird nicht als Flanke detektiert.

Ausgangssignale können sich in besonderen Double-Werten fangen.
Bei Funktionsblöcken, deren Ausgangswerte rückgekoppelt werden, ist zu beachten, dass besondere Double-Werte, wie zum Beispiel „NaN“, als Ergebnis der Ausgangsfunktion durch Rückkopplung beim nächsten Funktionsaufruf zwangsläufig wieder zum Ergebnis „NaN“ führen können. Das Zurücksetzen der internen Variablen eines Funktionsblocks ist dann nur noch mit einem Aufruf der Methode „Reset“ möglich.

Extrem große und kleine DeltaT-Werte sind zu vermeiden.
Einige zeitabhängige Funktionsblöcke können auf extrem große oder kleine DeltaT-Werte sehr empfindlich reagieren. Davon sind Übertragungsglieder des Namensbereichs „Controller“ betroffen, deren Ausgangswert sich in Abhängigkeit von der Zykluszeit auf die Ein- oder Ausgangswerte vorheriger Funktionsaufrufe bezieht.

Numerische Fehler werden auf Kosten der Genauigkeit unterdrückt.

Bei wenigen Funktionsblöcken der Bibliothek war es erforderlich, numerische Fehler zu unterdrücken. Dies hat zur Folge, dass der betroffene Funktionsblock zwar Werte mit der Genauigkeit „Double“ annimmt und zurückgibt, intern aber bestimmte Berechnungen nur mit der Genauigkeit „Single“ durchführen kann. Ist dieser Fall zutreffend, wird in der jeweiligen Dokumentation des Funktionsblocks ausdrücklich darauf hingewiesen.

Auch in Funktionsblöcken können Ausnahmefehler ausgelöst werden.

Die meisten Funktionsblöcke der Automatisierungsbibliothek antworten auf nicht plausible Eingangs- oder Parameterwerte mit einem „NaN“ als Ausgangswert. In einigen Fällen werden aber bei sehr speziellen Konstellationen Ausnahmefehler nicht unterdrückt oder sogar bewusst ausgelöst, um den Entwickler auf fatale Fehler in der Programmierung hinzuweisen.

Auch die Automatisierungsbibliothek kann Fehler enthalten.

Alle Funktionsblöcke der Automatisierungsbibliothek wurden mit der speziell dafür entwickelten Testumgebung ausgiebig getestet. Dennoch kann ein Fehlverhalten von Bibliothekselementen nicht unter allen möglichen Konstellationen ausgeschlossen werden. Sicherheitsrelevante Funktionen sind deshalb nicht mit der Automatisierungsbibliothek umzusetzen!

Die Interpretation von „Double“-Werten ist durch Funktionen beschrieben.

Die Automatisierungsbibliothek enthält unter dem Modul „Utility“ Funktionen, die beschreiben, wie „Double“-Werte von Bibliotheksbausteinen interpretiert werden.

```
'Abbildung von Double auf Boolean
Public Function DoubleToBoolean(ByVal Value As Double) As Boolean
    '0 oder NaN -> False, Sonst -> True
    Return If(Value > 0 Or Value < 0, True, False)
End Function

'Abbildung von Boolean auf Double
Public Function BooleanToDouble(ByVal Value As Boolean) As Double
    'True -> 1, Sonst -> 0
    Return If(Value = True, 1, 0)
End Function

'Positive Flanke erkennen
Public Function DetectHiFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung in positiver Richtung -> True, Sonst -> False
    Return If(NewValue > LastValue, True, False)
End Function

'Negative Flanke erkennen
Public Function DetectLoFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung in negativer Richtung -> True, Sonst -> False
    Return If(NewValue < LastValue, True, False)
End Function

'Flanke erkennen
Public Function DetectAnyFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung -> True, Sonst -> False
    Return If(NewValue > LastValue Or NewValue < LastValue, True, False)
End Function
```

2.4 Datentyp „Double“

Die folgenden Beispiele sollen zeigen, wie VB.NET Werte des Typs „Double“ verarbeitet.

Double-Werte

```
Double.PositiveInfinity                          => +unendlich
Double.NegativeInfinity                          => -unendlich
Double.MaxValue                                  => 1,79769313486232E+308
Double.MinValue                                  => -1,79769313486232E+308
Double.Epsilon                                   => 4,94065645841247E-324
Double.NaN                                       => NaN
```

Standardwert

```
(New Double)                                     => 0
Nothing = 0                                      => True
Nothing = Double.NaN                             => False
```

Illegale Rechenoperationen

```
0/0                                              => NaN
 1/0                                             => +unendlich
-1/0                                             => -unendlich
3 + Double.NaN                                   => NaN
```

Vergleiche

```
If(0, True, False)                               => False
If(5, True, False)                               => True
If(-5, True, False)                              => True
If(double.NaN, True, False)                      => True
If(Double.Epsilon, True, False)                  => True
If(Double.PositiveInfinity, True, False)         => True

Double.NaN = 0                                   => False
Double.NaN > 0                                   => False
Double.NaN = Double.NaN                          => False
Double.NaN <> Double.NaN                         => True

Double.Epsilon = 0                               => False
Double.Epsilon > 0                               => True
Double.Epsilon = Double.Epsilon                  => True

Double.MaxValue > 0                              => True
Double.MaxValue > Double.MaxValue                => False
Double.MaxValue = Double.MaxValue                => True

Double.PositiveInfinity > 0                      => True
Double.PositiveInfinity > Double.PositiveInfinity  => False
Double.PositiveInfinity = Double.PositiveInfinity  => True

-Double.PositiveInfinity = Double.NegativeInfinity  => True
Double.PositiveInfinity = Double.MaxValue        => False
Double.PositiveInfinity > Double.MaxValue        => True
```

Double-Funktionen

```
Double.NaN.Equals(Double.NaN)                    => True

Double.IsNaN(0)                                  => False
Double.IsNaN(Double.NaN)                         => True
Double.IsNaN(Double.PositiveInfinity)            => False

Double.IsInfinity(0)                             => False
Double.IsInfinity(Double.NaN)                    => False
Double.IsInfinity(Double.PositiveInfinity)       => True
Double.IsInfinity(Double.NegativeInfinity)       => True
```

Boolean zu Double

```
5 * False                                   => 0
5 * True                                    => -5
5 * Convert.ToDouble(False)                 => 0
5 * Convert.ToDouble(True)                  => 5
```

Double zu Integer

```
Convert.ToInt32( 3.5)                       => 4
Convert.ToInt32( 2.5)                       => 2
Convert.ToInt32( 1.5)                       => 2
Convert.ToInt32( 0.5)                       => 0
Convert.ToInt32(-0.5)                       => 0
Convert.ToInt32(-1.5)                       => -2
Convert.ToInt32(-2.5)                       => -2
Convert.ToInt32(-3.5)                       => -4
Convert.ToInt32(Double.NaN)                 => System.OverflowException
Convert.ToInt32(Double.MaxValue)            => System.OverflowException
```

Numerische Fehler

```
1   = 1.0                                   => True
0   = 0   + Double.Epsilon                  => False
0.1 = 0.1 + Double.Epsilon                  => True
0 + Double.Epsilon = 0 + 2 * Double.Epsilon => False
1 + Double.Epsilon = 1 + 2 * Double.Epsilon => True
Double.MaxValue = Double.MaxValue + 1       => True
Double.MaxValue = Double.MaxValue * 2       => False
```

Modulo

```
  0   Mod  5                                => 0
  5   Mod  5                                => 0
 -5   Mod  5                                => 0
 13.8 Mod  5                                => 3,8
-13.8 Mod  5                                => -3,8
 13   Mod -5                                => 3
5 Mod Double.Epsilon                        => 0
5 Mod Double.NaN                            => NaN
5 Mod Double.PositiveInfinity               => NaN
Double.NaN Mod 5                            => NaN
Double.Epsilon Mod 5                        => 4,94065645841247E-324
Double.PositiveInfinity Mod 5               => NaN
```

Mathematische Funktionen

```
System.Math.Abs(Double.NaN)                 => NaN
System.Math.Abs(Double.NegativeInfinity)    => +unendlich
System.Math.Sign(Double.NaN)                => System.ArithmeticException
System.Math.Sign(Double.NegativeInfinity)   => -1
System.Math.Sin(Double.PositiveInfinity)    => NaN
System.Math.Exp(Double.PositiveInfinity)    => +unendlich
System.Math.Exp(Double.NegativeInfinity)    => 0
System.Math.Truncate(Double.PositiveInfinity) => +unendlich
System.Math.Truncate(Double.NegativeInfinity) => -unendlich
```

Runden

```
Math.Round(-1 / 2, 0, MidpointRounding.AwayFromZero) => -1
Math.Round( 1 / 2, 0, MidpointRounding.AwayFromZero) => 1
Math.Round(-3 / 2, 0, MidpointRounding.AwayFromZero) => -2
Math.Round( 3 / 2, 0, MidpointRounding.AwayFromZero) => 2
Math.Round(-1 / 2, 0, MidpointRounding.ToEven)       => 0
Math.Round( 1 / 2, 0, MidpointRounding.ToEven)       => 0
Math.Round(-3 / 2, 0, MidpointRounding.ToEven)       => -2
Math.Round( 3 / 2, 0, MidpointRounding.ToEven)       => 2
```

3 Dokumentation der Funktionsblöcke

In diesem Kapitel sind alle Funktionsblöcke der Automatisierungsbibliothek dokumentiert. Die Dokumentation setzt sich aus kurzen Beschreibungen der Funktionsblöcke, Funktionsblock-Testergebnisse und Programmcodes, die mit Kommentaren erklärt werden, zusammen. Zudem behandelt dieses Kapitel die mathematischen Hintergründe der Funktionsblöcke und gibt wertvolle Tipps zur Einbindung in eigene Projekte. In der folgenden Übersicht werden die Seitenzahlen der einzelnen Namensbereiche angegeben, auf denen Verweise zu den enthaltenen Funktionsblöcken zu finden sind.

Regelglieder (Namensbereich: „Controller“ → Seite 45)

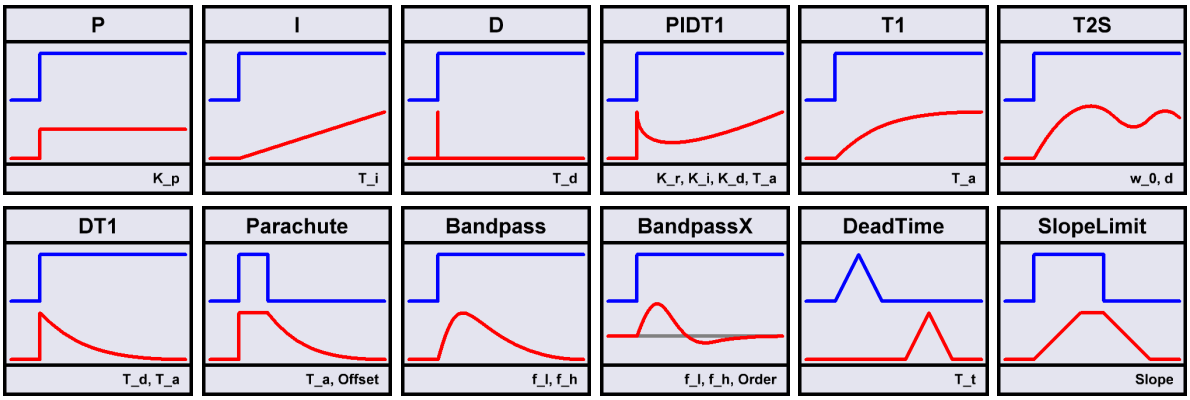


Abb. 3-1: Funktionsblöcke des Namensbereichs „Controller“

Signalgeneratoren (Namensbereich: „Generator“ → Seite 75)

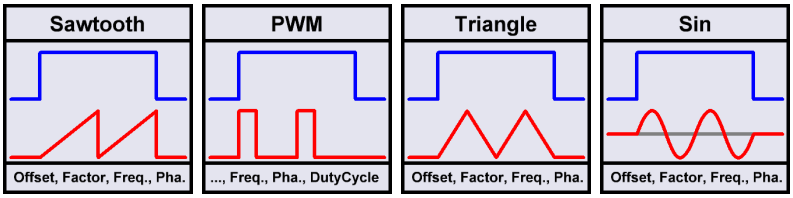


Abb. 3-2: Funktionsblöcke des Namensbereichs „Generator“

Zeitglieder (Namensbereich: „Timer“ → Seite 85)

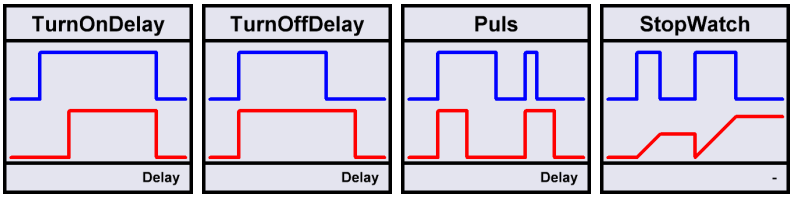


Abb. 3-3: Funktionsblöcke des Namensbereichs „Timer“

Flankenerkennungen (Namensbereich: „Flank“ → Seite 95)

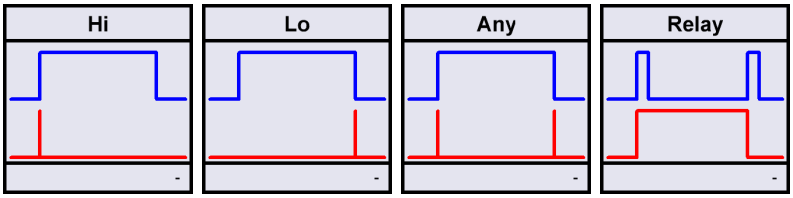


Abb. 3-4: Funktionsblöcke des Namensbereichs „Flank“

Analoge Übertragungsglieder (Namensbereich: „Analog“ → Seite 105)

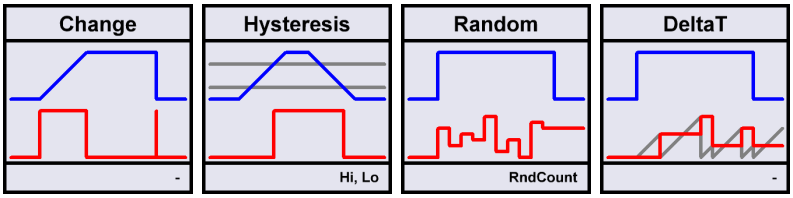


Abb. 3-5: Funktionsblöcke des Namensbereichs „Analog“

Grundlegende Funktionen (Namensbereich: „Generic“ → Seite 115)

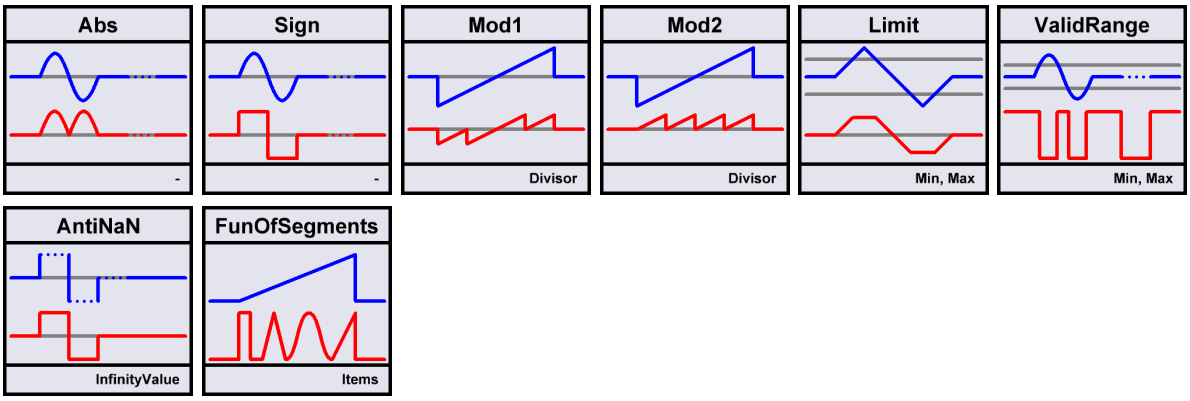


Abb. 3-6: Funktionsblöcke des Namensbereichs „Generic“

Funktions-Segmente (Namensbereich: „FuncSegment“ → Seite 133)

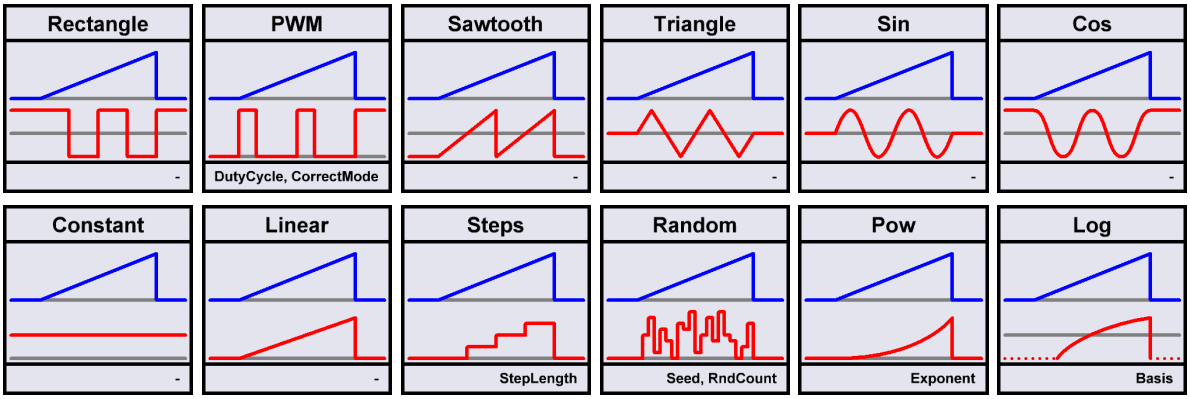


Abb. 3-7: Funktionsblöcke des Namensbereichs „FuncSegment“

Kompilierung zur Laufzeit (Namensbereich: „JustInTime“ → Seite 159)

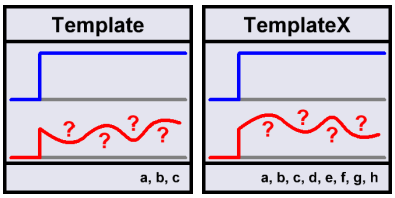


Abb. 3-8: Funktionsblöcke des Namensbereichs „JustInTime“

Verweise

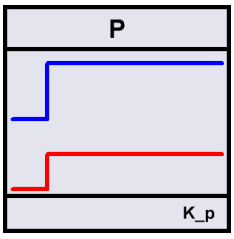
Aufbau der Bibliothek → Seite 29

Implementierung eines Funktionsblocks → Seite 36

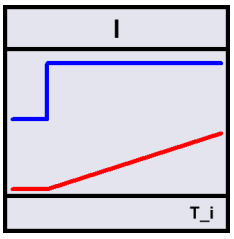
Hinweise zur Implementierung → Seite 38

3.1 Regelglieder (Controller)

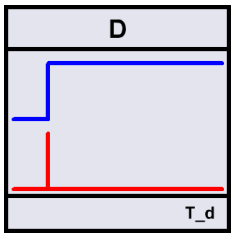
Der Namensbereich „Controller“ enthält Übertragungsglieder für Anwendungen aus der Regelungstechnik. Hierzu gehören zum Beispiel grundlegende Glieder, die ein Proportional-, Integrations- oder Differentiations-Verhalten aufweisen. Zudem sind Übertragungsglieder enthalten, die Signale filtern und auf verschiedene Art und Weise verzögern können. Ihren Einsatz finden die Funktionsblöcke überall dort, wo technische Systeme simuliert oder ihre Zustandsgrößen geregelt werden müssen.



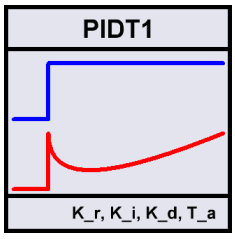
→ Seite 49



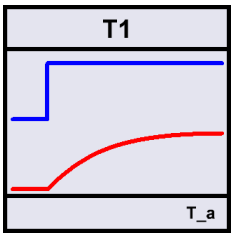
→ Seite 51



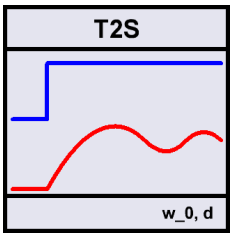
→ Seite 53



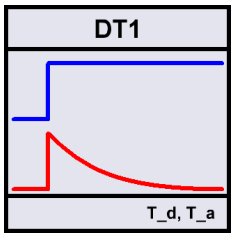
→ Seite 55



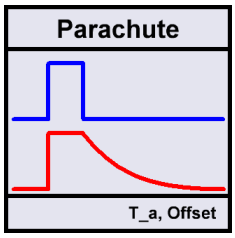
→ Seite 59



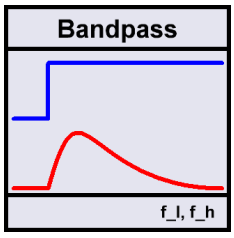
→ Seite 61



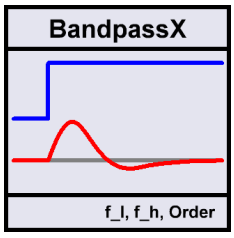
→ Seite 63



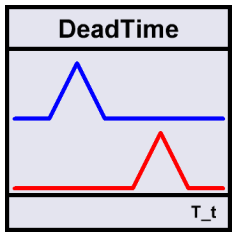
→ Seite 65



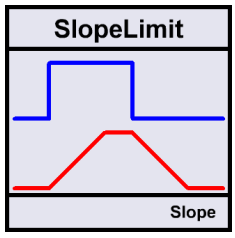
→ Seite 67



→ Seite 69



→ Seite 71



→ Seite 73

Verweise

Basisklasse „Zeitabhängiger Funktionsblock“	→ Seite 33
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38

Approximationsverfahren

Die Funktionsblöcke der Automatisierungstechnik sind für die Verarbeitung von zeit- und wertediskreten Signalen ausgelegt, wobei die Werte der Signale zudem weder in zeit- noch in werteäquidistanten Abständen vorliegen müssen. Um dies umzusetzen, war es erforderlich, bei der Programmierung der Funktionsblöcke des Namensbereichs „Controller“ die Übertragungsfunktionen der einzelnen Funktionsblöcke mittels Differenzengleichungen zu implementieren. Die hierzu angewandte Transformation wurde für jeden Funktionsblock mit verschiedenen Approximationsverfahren durchgeführt, die über den Parameter „ApproxType“ des Funktionsblocks zur Auswahl stehen.

Nachfolgend werden die genannten Begriffe „zeit- und wertediskret“ sowie „zeit- und werteäquidistant“ anhand von Beispielen mit einer Grafik veranschaulicht und erklärt.

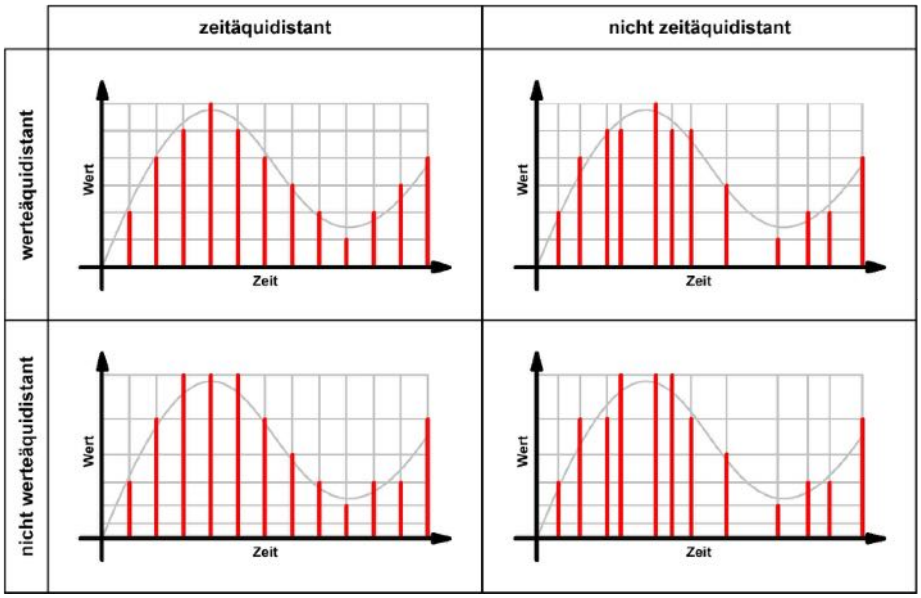


Abb. 3-9: Diskrete Zykluszeiten und Werte

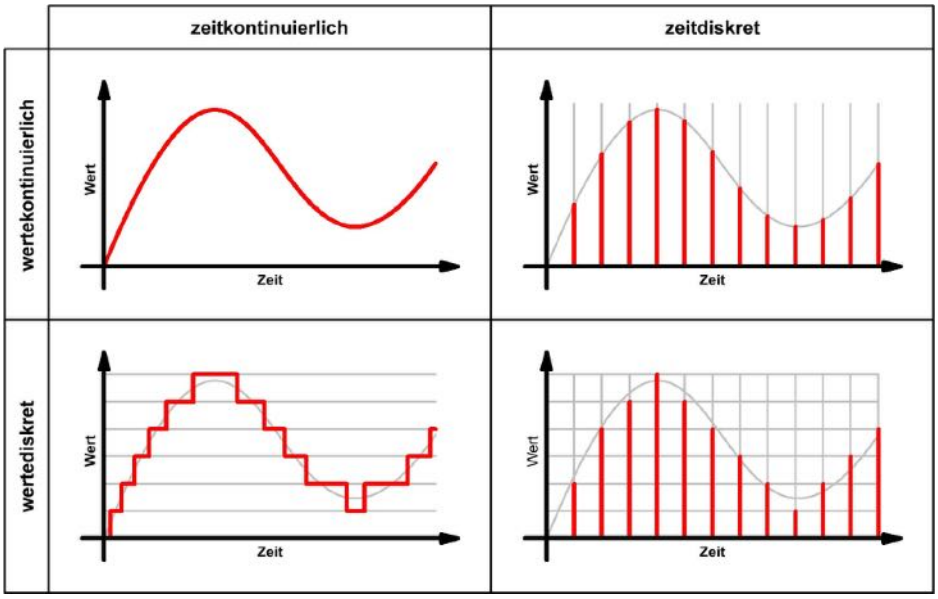


Abb. 3-10: Äquidistante Zykluszeiten und Werte

Die folgende Abbildung zeigt die notwendigen Schritte zur Herleitung einer Differenzengleichung am Beispiel des „T2S“-Gliedes. Die einzelnen Schritte werden hierzu auf Seite 24 erklärt. Details zu den mathematischen Hintergründen der jeweiligen Transformationen sowie Approximations-Verfahren sind zudem in der im Literaturverzeichnis aufgeführten einschlägigen Fachliteratur zu finden.

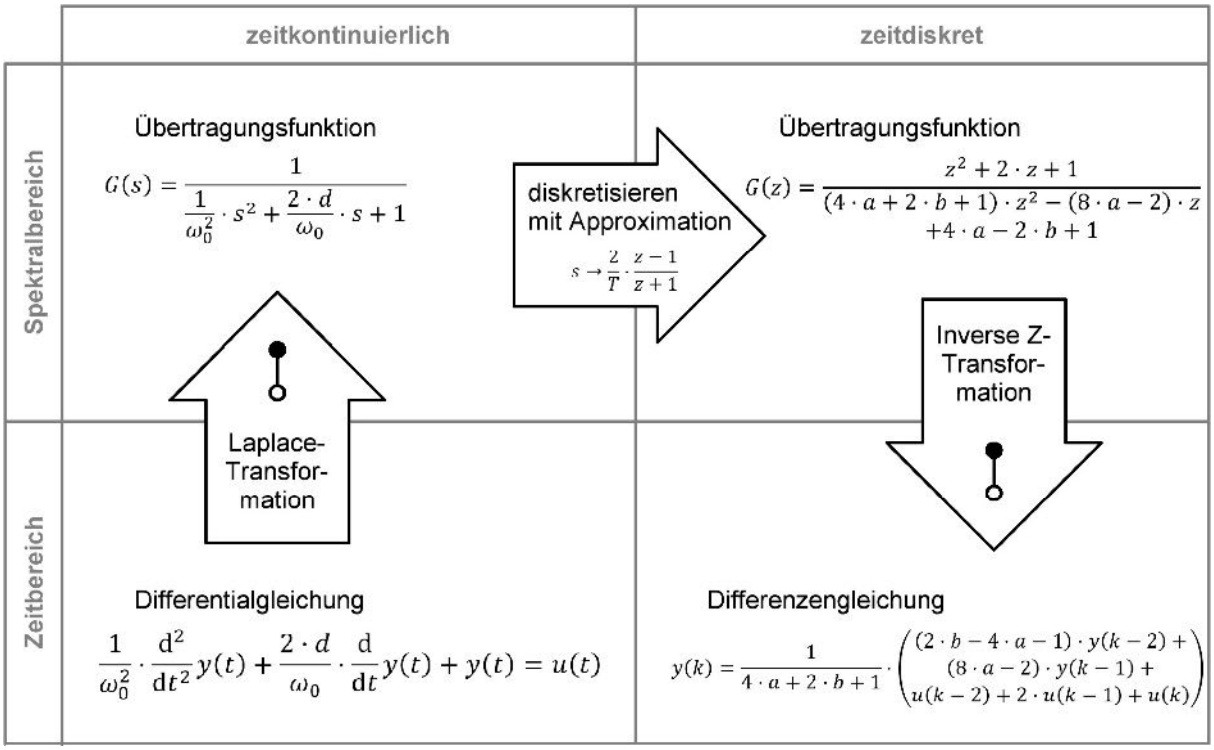


Abb. 3-11: Transformationen zum Berechnen einer Differenzengleichung

<u>Approximationsverfahren:</u>

Approximation nach Euler-Vorwärts-Methode

$$s \to \frac{z-1}{T} \quad \Rightarrow \text{(Übergang vom zeitkontinuierlichen in den zeitdiskreten Spektralbereich)}$$

Approximation nach Euler-Rückwärts-Methode

$$s \to \frac{z-1}{T \cdot z} \quad \Rightarrow \text{(Übergang vom zeitkontinuierlichen in den zeitdiskreten Spektralbereich)}$$

Approximation nach Tustin-Methode

$$s \to \frac{2}{T} \cdot \frac{z-1}{z+1} \quad \Rightarrow \text{(Übergang vom zeitkontinuierlichen in den zeitdiskreten Spektralbereich)}$$

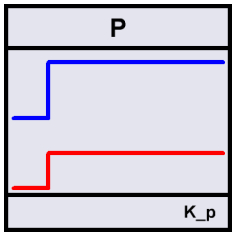
Approximation nach Matched-Z-Transformation

$$G(z) = K' \cdot \frac{\prod_{k=1}^{M}(z - e^{n_k \cdot T})}{\prod_{k=1}^{N}(z - e^{p_k \cdot T})}$$

n_k *sind Nullstellen von* $G(s)$
p_k *sind Polstellen von* $G(s)$
T *ist die Zykluszeit*
K' *ist ein Verstärkungsfaktor*

$\Rightarrow$ (*Übergang vom zeitkontinuierlichen in den zeitdiskreten Spektralbereich*)

3.1.1 Proportionalglied (Controller.P)



Das „P"-Glied ist ein Übertragungsglied, welches ein proportionales Übertragungsverhalten aufweist. Dies bedeutet, dass der Ausgangswert das Produkt aus der Eingangsgröße und einem Verstärkungsfaktor ist.

Eigenschaften

Übertragungsfunktion: $G(s) = K_p$

Parameter: K_p = Proportionalverstärkung

Funktionsblock-Testergebnis

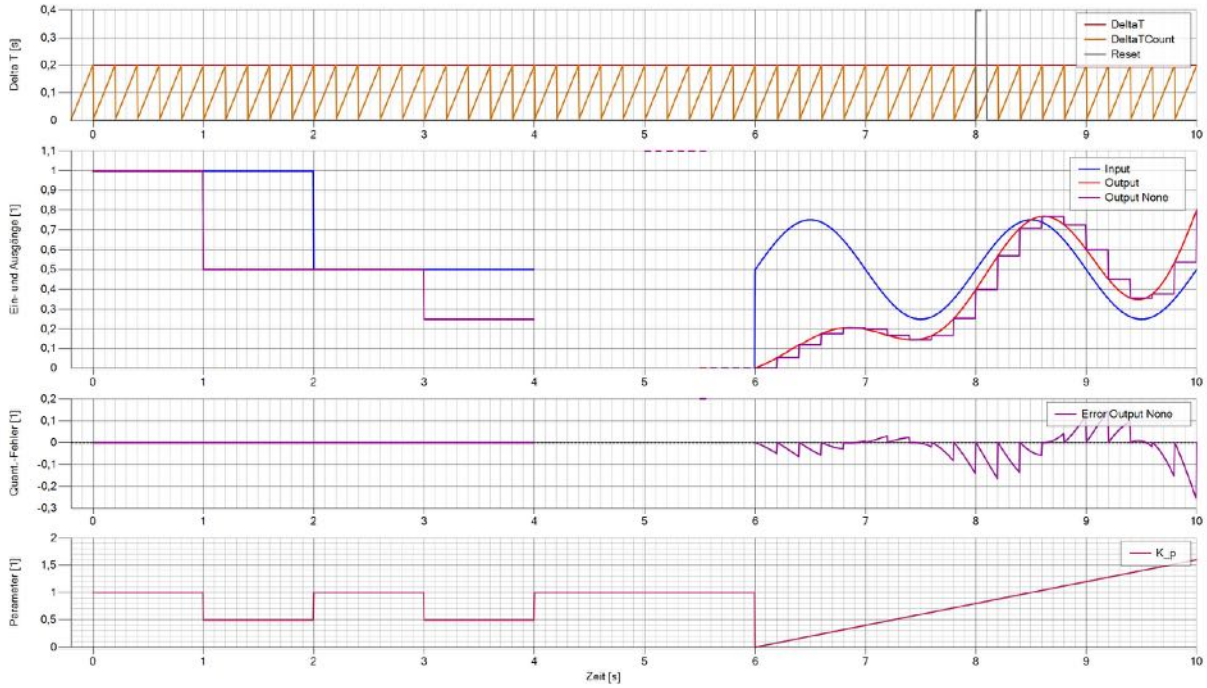


Abb. 3-12: Funktionsblock-Testergebnis: Proportionalglied

0 - 2 Sprungantwort auf das Eingangssignal $Input = 1$ bei $K_p = 1$ bzw. $K_p = 0{,}5$

2 - 4 Sprungantwort auf das Eingangssignal $Input = 0{,}5$ bei $K_p = 1$ bzw. $K_p = 0{,}5$

4 - 5 Reaktion auf $Input = NaN$

5 - 6 Reaktion auf $Input = \infty$ bzw. $Input = -\infty$

6 - 8 Sinusfunktion am Eingang wird durch eine Rampe an K_p in Y-Richtung skaliert

8 – 9 Zurücksetzen durch Aufrufen der Funktion „*Reset*" (hat keine Auswirkung)

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = K_p \cdot U(s)$

$y(t) = K_p \cdot u(t)$

Zeitdiskret: $Y(z) = K_p \cdot U(z)$

$y(k) = K_p \cdot u(k-1)$

Programmcode

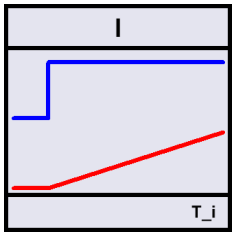
```
'#### Proportionalglied ####
Public Class P
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.None
    Public Enum ApproxTypeEnum
        None 'Keine Approximation
    End Enum

    Public Property K_p As Double = 1 'Proportionalverstärkung

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.None 'Differenzengleichung
                Return K_p * u(0)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.2 Integrationsglied (Controller.I)



Das „I"-Glied ist ein Übertragungsglied, welches ein integrierendes Übertragungsverhalten aufweist. Dies bedeutet, dass sich die Amplitude des Ausgangs proportional zum Zeitintegral des Eingangssignals verhält.

Eigenschaften

Übertragungsfunktion: $G(s) = \frac{1}{T_i} \cdot \frac{1}{s}$

Parameter: T_i = Integrationszeitkonstante

Funktionsblock-Testergebnis

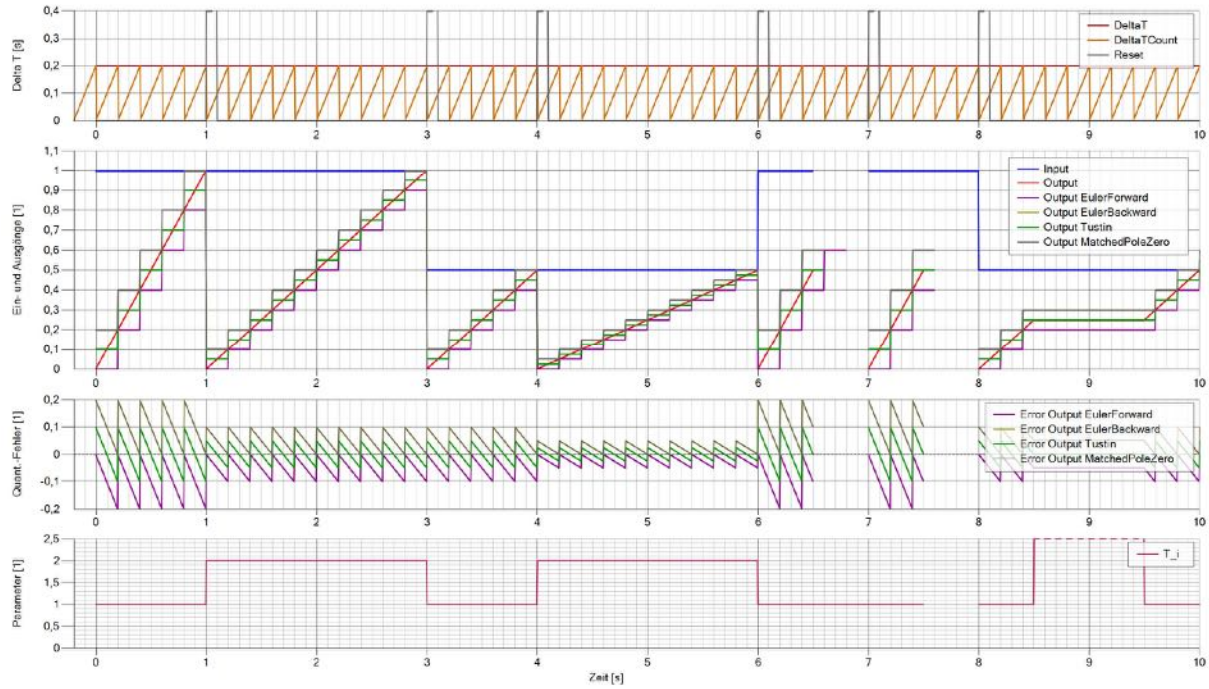


Abb. 3-13: Funktionsblock-Testergebnis: Integrationsglied

0 - 3 Sprungantwort auf das Eingangssignal $Input = 1$ bei $T_i = 1$ bzw. $T_i = 2$

3 - 6 Sprungantwort auf das Eingangssignal $Input = 0{,}5$ bei $T_i = 1$ bzw. $T_i = 2$

6 - 7 Reaktion auf $Input = NaN$

7 - 8 Reaktion auf $T_i = NaN$

8 - 9 Zurücksetzen durch Aufrufen der Funktion „*Reset*" (setzt Integrationswert zurück)

8 - 10 Reaktion auf $T_i = \infty$

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = \frac{1}{T_i} \cdot \frac{1}{s} \cdot U(s)$

$y(t) = \frac{1}{T_i} \cdot \int u(t) \cdot dt$

Euler-Vorwärts: $Y(z) = \frac{1}{T_i} \cdot \frac{T}{z-1} \cdot U(z)$

$y(k) = \frac{T}{T_i} \cdot u(k-1) + y(k-1)$

Euler-Rückwärts: $Y(z) = \frac{1}{T_i} \cdot \frac{T \cdot z}{z-1} \cdot U(z)$

$y(k) = \frac{T}{T_i} \cdot u(k) + y(k-1)$

Tustin-Methode: $Y(z) = \frac{1}{T_i} \cdot \frac{T}{2} \cdot \frac{z+1}{z-1} \cdot U(z)$

$y(k) = \frac{T}{2 \cdot T_i} \cdot \big(u(k-1) + u(k)\big) + y(k-1)$

Machted-Pole-Zero: $Y(z) = \frac{1}{T_i} \cdot \frac{T \cdot z}{z-1} \cdot U(z)$

$y(k) = \frac{T}{T_i} \cdot u(k) + y(k-1)$

Programmcode

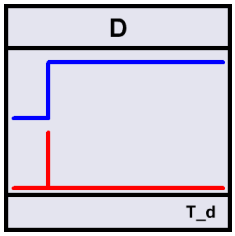
```
'#### Integrationsglied ####
Public Class I
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_i As Double = 1 'Integrationszeitkonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T / T_i * u(k - 1) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return T / T_i * u(k) + y(k - 1)
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return T / (2 * T_i) * (u(k - 1) + u(k)) + y(k - 1)
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return T / T_i * u(k) + y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.3 Differenzierer (Controller.D)



Das „D“-Glied ist ein Übertragungsglied, welches ein differenzierendes Übertragungsverhalten aufweist. Dies bedeutet, dass sich die Amplitude des Ausgangssignals proportional zur Steigung des Eingangssignals verhält. Die Übertragungsfunktion des „D“-Glieds weist im Zähler eine höhere Ordnung als im Nenner auf. Deshalb ist das Übertragungsglied technisch nur dann realisierbar, wenn es mit einer Verzögerung kombiniert oder, wie hier im Funktionsblock, approximiert im zeitdiskreten Bereich beschrieben wird.

Eigenschaften

Übertragungsfunktion: $G(s) = T_d \cdot s$

Parameter: T_d = Differentiationskonstante

Funktionsblock-Testergebnis

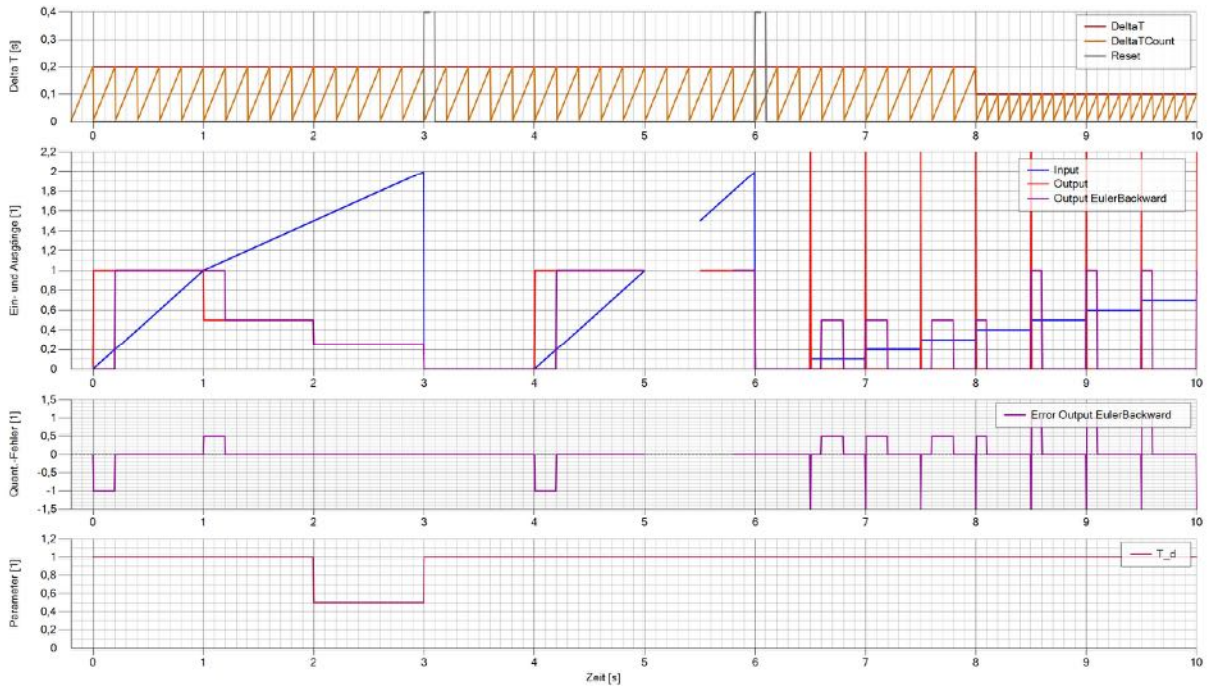


Abb. 3-14: Funktionsblock-Testergebnis: Differenzierer

0 - 1 Systemantwort auf eine Rampe am Eingang mit der Steigung 1 bei $T_d = 1$

1 - 2 Systemantwort auf eine Rampe am Eingang mit der Steigung $0{,}5$ bei $T_d = 1$

2 - 3 Systemantwort auf eine Rampe am Eingang mit der Steigung $0{,}5$ bei $T_d = 0{,}5$

3 - 4 Zurücksetzen durch Aufrufen der Funktion „*Reset*“ (negativen Impuls unterdrücken)

4 - 6 Reaktion auf $Input = NaN$ (Funktionsblock muss nicht zurückgesetzt werden)

6 - 10 Treppenfunktion mit einer Stufenhöhe $0{,}1$ am Eingang bei Zykluszeit $0{,}2$ bzw. $0{,}1$ (gleiche Impulsflächen, weil Impulshöhe des Ausgangs von der Zykluszeit abhängt)

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = T_d \cdot s \cdot U(s)$

$y(t) = T_d \cdot \frac{\mathrm{d}}{\mathrm{d}t} u(t)$

Euler-Vorwärts: $Y(z) = T_d \cdot \frac{z-1}{T} \cdot U(z)$

$y(k) = \frac{T_d}{T} \cdot \big(u(k+1) - u(k)\big)$

⇒ nicht kausal ⇒ nicht realisierbar ⇒ nicht implementiert

Euler-Rückwärts: $Y(z) = T_d \cdot \frac{z-1}{T \cdot z} \cdot U(z)$

$y(k) = \frac{T_d}{T} \cdot \big(u(k) - u(k-1)\big)$

Tustin-Methode: $Y(z) = T_d \cdot \frac{2}{T} \cdot \frac{z-1}{z+1} \cdot U(z)$

$y(k) = \frac{2 \cdot T_d}{T} \cdot \big(u(k) - u(k-1)\big) - y(k-1)$

⇒ Kausalität gegeben, aber grenzstabil ⇒ nicht implementiert

Programmcode

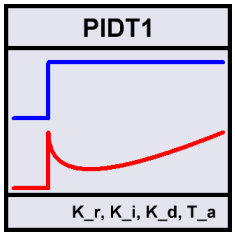
```
'#### Differenzierer ####
Public Class D
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.EulerBackward
    Public Enum ApproxTypeEnum
        EulerBackward 'Approximation nach Euler-Rückwärts
    End Enum

    Public Property T_d As Double = 1 'Differentiationskonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return T_d / T * (u(k) - u(k - 1))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.4 PIDT1-Übertragungsglied (Controller.PIDT1)



Das PIDT1-Glied setzt sich aus einem P-, I- und DT1-Glied (Seite 63) zusammen und weist somit anteilig ein proportionales, integratives und ein in 1. Ordnung verzögerndes differenzierendes Übertragungsverhalten auf. Zu beachten ist, dass der Parameter K_r auf alle drei Regleranteile wirkt.

Eigenschaften

Übertragungsfunktion: $G(s) = K_r \cdot \left(1 + \frac{1}{T_i \cdot s} + \frac{T_d \cdot s}{T_a \cdot s + 1}\right)$

Parameter:

K_r = Regler-Verstärkung

T_i = Integrationszeitkonstante bzw. Nachstellzeit

T_d = Differentiationskonstante bzw. Vorhaltzeit

T_a = Zeitkonstante des Differenzierers

Funktionsblock-Testergebnis

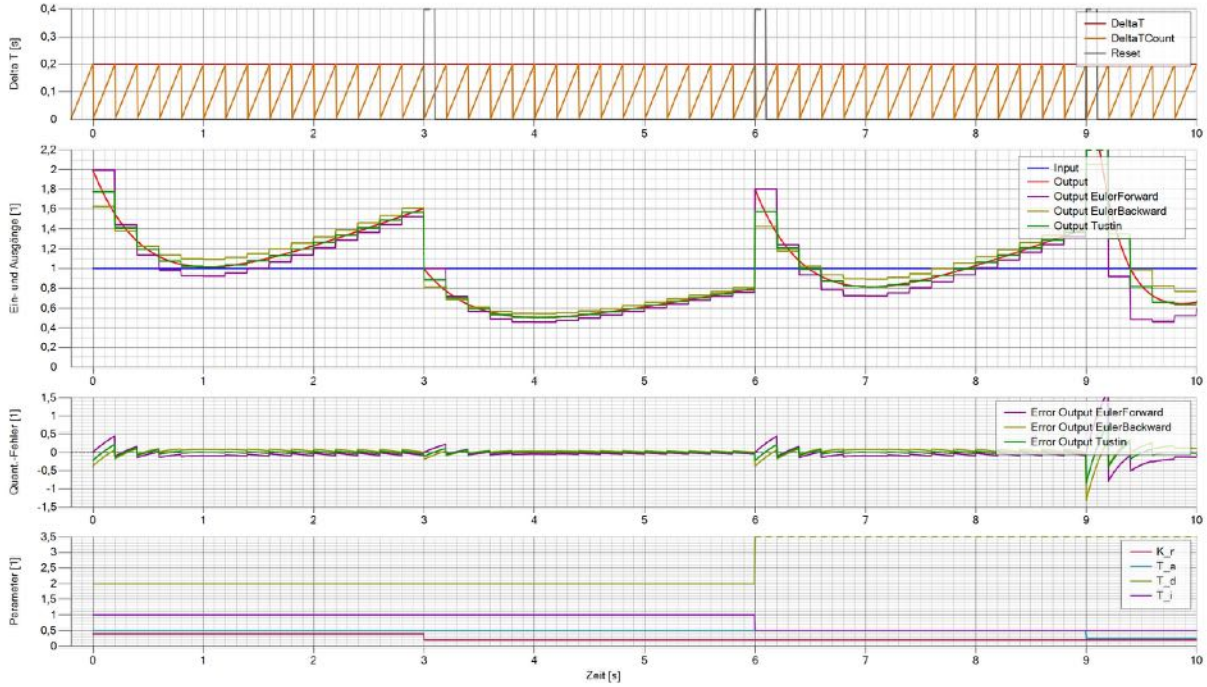


Abb. 3-15: Funktionsblock-Testergebnis: „PIDT1"-Übertragungsglied

0 - 3 Sprungantwort bei $K_r = 0{,}4$ $T_i = 1$ $T_d = 2$ $T_a = 0{,}5$

3 - 4 Sprungantwort bei $K_r = 0{,}2$ $T_i = 1$ $T_d = 2$ $T_a = 0{,}5$
(kleinere Regler-Verstärkung)

6 - 9 Sprungantwort bei $K_r = 0{,}2$ $T_i = 0{,}5$ $T_d = 4$ $T_a = 0{,}5$
(verhältnismäßig geringerer Proportionalanteil im Vergleich zu 0 - 3)

9 - 10 Sprungantwort bei $K_r = 0{,}2$ $T_i = 0{,}5$ $T_d = 4$ $T_a = 0{,}25$
(durch halbe Zeitkonstante wirkt der D-Anteil im ersten Moment doppelt so stark)

Parallelstruktur und Sprungantwort

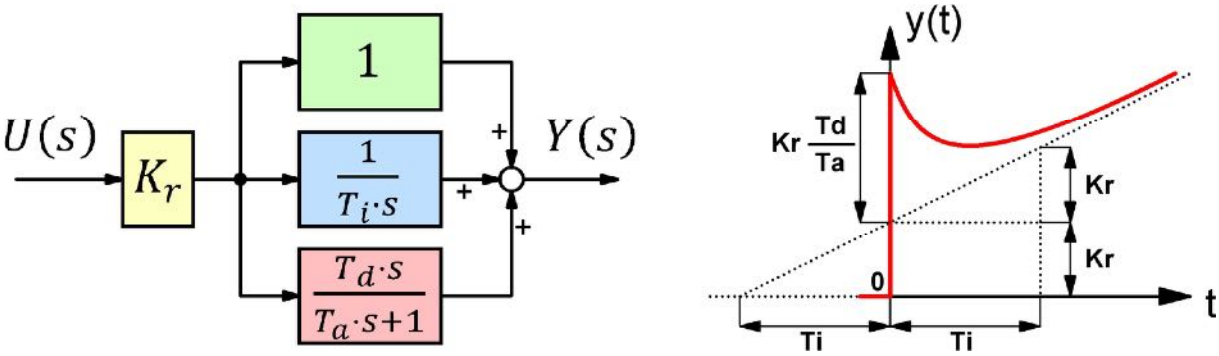


Abb. 3-16: Parallelstruktur und Sprungantwort des „PIDT1“-Übertragungsglieds

Mathematischer Hintergrund

Zeitkontinuierlich:

$$Y(s) = K_r \cdot \left(1 + \frac{1}{T_i \cdot s} + \frac{T_d \cdot s}{T_a \cdot s + 1}\right) \cdot U(s)$$

$$T_a \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = K_r \cdot \left(\frac{T_a + T_i}{T_i} \cdot u(t) + \frac{1}{T_i} \cdot \int u(t)dt + (T_a + T_d) \cdot \frac{\mathrm{d}}{\mathrm{d}t} u(t)\right)$$

Euler-Vorwärts:

$$Y(z) = K_r \cdot \left(1 + \frac{T}{T_i \cdot z - T_i} + \frac{T_d \cdot z - T_d}{T + T_a \cdot z - T_a}\right) \cdot U(z)$$

$$y(k) = \frac{1}{T_i \cdot T_a} \cdot \begin{pmatrix} K_r \cdot T_i \cdot (T_d + T_a) \cdot u(k) \\ +K_r \cdot \left(T_i \cdot \left(T - 2 \cdot (T_d + T_a)\right) + T_a \cdot T\right) \cdot u(k-1) \\ +K_r \cdot \left(T_i \cdot (T_d + T_a - T) - T \cdot (T_a - T)\right) \cdot u(k-2) \\ +T_i \cdot (2 \cdot T_a - T) \cdot y(k-1) \\ -T_i \cdot (T_a - T) \cdot y(k-2) \end{pmatrix}$$

Euler-Rückwärts:

$$Y(z) = K_r \cdot \left(1 + \frac{T \cdot z}{T_i \cdot z - T_i} + \frac{T_d \cdot z - T_d}{T \cdot z + T_a \cdot z - T_a}\right) \cdot U(z)$$

$$y(k) = \frac{1}{T_i \cdot (T_a + T)} \cdot \begin{pmatrix} K_r \cdot T_i \cdot (T_d + T_a) \cdot u(k-2) \\ -K_r \cdot (T_i \cdot (2 \cdot (T_d + T_a) + T) + T_a \cdot T) \cdot u(k-1) \\ +K_r \cdot \left(T_i \cdot (T_d + T_a + T) + T \cdot (T_a + T)\right) \cdot u(k) \\ -T_i \cdot T_a \cdot y(k-2) \\ +T_i \cdot (2 \cdot T_a + T) \cdot y(k-1) \end{pmatrix}$$

Tustin-Methode:

$$Y(z) = K_r \cdot \left(1 + \frac{T \cdot z + T}{T_i \cdot (2 \cdot z - 2)} + \frac{T_d \cdot (2 \cdot z - 2)}{T \cdot z + T + T_a \cdot (2 \cdot z - 2)}\right) \cdot U(z)$$

$$y(k) = \frac{1}{2 \cdot T_i \cdot (2 \cdot T_a + T)} \cdot \begin{pmatrix} K_r \cdot (2 \cdot (T_i \cdot (2 \cdot (T_d + T_a) - T) - T_a \cdot T) + T^2) \cdot u(k-2) \\ +2 \cdot K_r \cdot (-4 \cdot T_i \cdot (T_d + T_a) + T^2) \cdot u(k-1) \\ +K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) + T) + T \cdot (2 \cdot T_a + T)\right) \cdot u(k) \\ -2 \cdot T_i \cdot (2 \cdot T_a - T) \cdot y(k-2) \\ +8 \cdot T_i \cdot T_a \cdot y(k-1) \end{pmatrix}$$

Herleitung der Differenzengleichung approximiert nach der Tustin-Methode

Übertragungsfunktion:

$$\frac{Y(s)}{U(s)} = G(s) = K_r \cdot \left(1 + \frac{1}{T_i \cdot s} + \frac{T_d \cdot s}{1 + T_a \cdot s}\right)$$

Approximations-Methode nach Tustin:

$$s \to \frac{2}{T} \cdot \frac{z-1}{z+1} = \frac{2 \cdot z - 2}{T \cdot z + T}$$

Übertragungsfunktion des zeitdiskreten Bereichs approximiert nach der Tustin-Methode:

$$\frac{Y(z)}{U(z)} = G(z) = K_r \cdot \left(1 + \frac{1}{T_i \cdot \frac{2 \cdot z - 2}{T \cdot z + T}} + \frac{T_d \cdot \frac{2 \cdot z - 2}{T \cdot z + T}}{1 + T_a \cdot \frac{2 \cdot z - 2}{T \cdot z + T}}\right) \qquad G(z) = \frac{K_r}{1} + \frac{K_r \cdot T \cdot z + K_r \cdot T}{2 \cdot T_i \cdot z - 2 \cdot T_i} + \frac{K_r \cdot T_d \cdot 2 \cdot z - 2 \cdot K_r \cdot T_d}{T \cdot z + T + 2 \cdot T_a \cdot z - 2 \cdot T_a}$$

Übertragungsfunktion des zeitdiskreten Bereichs auf einen Hauptnenner gebracht:

$$G(z) = \frac{K_r \cdot (2 \cdot T_i \cdot z - 2 \cdot T_i) \cdot (T \cdot z + T + 2 \cdot T_a \cdot z - 2 \cdot T_a) + (K_r \cdot T \cdot z + K_r \cdot T) \cdot (T \cdot z + T + 2 \cdot T_a \cdot z - 2 \cdot T_a) + (K_r \cdot T_d \cdot 2 \cdot z - 2 \cdot K_r \cdot T_d) \cdot (2 \cdot T_i \cdot z - 2 \cdot T_i)}{(2 \cdot T_i \cdot z - 2 \cdot T_i) \cdot (T \cdot z + T + 2 \cdot T_a \cdot z - 2 \cdot T_a)}$$

Zähler und Nenner der Übertragungsfunktion des zeitdiskreten Bereichs ausmultipliziert:

$$G(z) = \frac{\begin{array}{c} 2 \cdot K_r \cdot T_i \cdot T \cdot z^2 + 2 \cdot K_r \cdot T_i \cdot T \cdot z + 4 \cdot K_r \cdot T_i \cdot T_a \cdot z^2 - 4 \cdot K_r \cdot T_i \cdot T_a \cdot z - 2 \cdot K_r \cdot T_i \cdot T \cdot z - 2 \cdot K_r \cdot T_i \cdot T - 4 \cdot K_r \cdot T_i \cdot T_a \cdot z + 4 \cdot K_r \cdot T_i \cdot T_a \\ + K_r \cdot T^2 \cdot z^2 + K_r \cdot T^2 \cdot z + 2 \cdot K_r \cdot T_a \cdot T \cdot z^2 - 2 \cdot K_r \cdot T_a \cdot T \cdot z + K_r \cdot T^2 \cdot z + K_r \cdot T^2 + 2 \cdot K_r \cdot T_a \cdot T \cdot z - 2 \cdot K_r \cdot T_a \cdot T \\ + 4 \cdot K_r \cdot T_i \cdot T_d \cdot z^2 - 4 \cdot K_r \cdot T_i \cdot T_d \cdot z - 4 \cdot K_r \cdot T_i \cdot T_d \cdot z + 4 \cdot K_r \cdot T_i \cdot T_d \end{array}}{2 \cdot T_i \cdot T \cdot z^2 + 2 \cdot T_i \cdot T \cdot z + 4 \cdot T_i \cdot T_a \cdot z^2 - 4 \cdot T_i \cdot T_a \cdot z - 2 \cdot T_i \cdot T \cdot z - 2 \cdot T_i \cdot T - 4 \cdot T_i \cdot T_a \cdot z + 4 \cdot T_i \cdot T_a}$$

Zähler und Nenner nach Variable z sortiert:

$$G(z) = \frac{\begin{array}{c} 2 \cdot K_r \cdot T_i \cdot T \cdot z^2 + 4 \cdot K_r \cdot T_i \cdot T_a \cdot z^2 + K_r \cdot T^2 \cdot z^2 + 2 \cdot K_r \cdot T_a \cdot T \cdot z^2 + 4 \cdot K_r \cdot T_i \cdot T_d \cdot z^2 \\ + 2 \cdot K_r \cdot T^2 \cdot z - 8 \cdot K_r \cdot T_i \cdot T_a \cdot z - 8 \cdot K_r \cdot T_i \cdot T_d \cdot z \\ + 4 \cdot K_r \cdot T_i \cdot T_d - 2 \cdot K_r \cdot T_i \cdot T + 4 \cdot K_r \cdot T_i \cdot T_a + K_r \cdot T^2 - 2 \cdot K_r \cdot T_a \cdot T \end{array}}{\begin{array}{c} 2 \cdot T_i \cdot T \cdot z^2 + 4 \cdot T_i \cdot T_a \cdot z^2 \\ - 8 \cdot T_i \cdot T_a \cdot z \\ - 2 \cdot T_i \cdot T + 4 \cdot T_i \cdot T_a \end{array}}$$

Definition der Übertragungsfunk.:

$$G(z) = \frac{Y(z)}{U(z)}$$

Variable z ausgeklammert und $G(z)$ ausgeschrieben:

$$1 = \frac{\begin{array}{c} (2 \cdot K_r \cdot T_i \cdot T + 4 \cdot K_r \cdot T_i \cdot T_a + K_r \cdot T^2 + 2 \cdot K_r \cdot T_a \cdot T + 4 \cdot K_r \cdot T_i \cdot T_d) \cdot U(z) \cdot z^2 \\ + (2 \cdot K_r \cdot T^2 - 8 \cdot K_r \cdot T_i \cdot T_a - 8 \cdot K_r \cdot T_i \cdot T_d) \cdot U(z) \cdot z \\ + (4 \cdot K_r \cdot T_i \cdot T_d - 2 \cdot K_r \cdot T_i \cdot T + 4 \cdot K_r \cdot T_i \cdot T_a + K_r \cdot T^2 - 2 \cdot K_r \cdot T_a \cdot T) \cdot U(z) \end{array}}{\begin{array}{c} (2 \cdot T_i \cdot T + 4 \cdot T_i \cdot T_a) \cdot Y(z) \cdot z^2 \\ + (-8 \cdot T_i \cdot T_a) \cdot Y(z) \cdot z \\ + (-2 \cdot T_i \cdot T + 4 \cdot T_i \cdot T_a) \cdot Y(z) \end{array}}$$

Durch Ausklammern vereinfacht:

$$1 = \frac{\begin{array}{c} K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) + T) + T \cdot (2 \cdot T_a + T)\right) \cdot U(z) \cdot z^2 \\ + 2 \cdot K_r \cdot \left(-4 \cdot T_i \cdot (T_d + T_a) + T^2\right) \cdot U(z) \cdot z \\ + K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) - T) - 2 \cdot T_a \cdot T + T^2\right) \cdot U(z) \end{array}}{\begin{array}{c} 2 \cdot T_i \cdot (2 \cdot T_a + T) \cdot Y(z) \cdot z^2 \\ - 8 \cdot T_i \cdot T_a \cdot Y(z) \cdot z \\ + 2 \cdot T_i \cdot (2 \cdot T_a - T) \cdot Y(z) \end{array}}$$

Für negative Exponenten durch z^2 dividiert:

$$1 = \frac{\begin{array}{c} K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) + T) + T \cdot (2 \cdot T_a + T)\right) \cdot U(z) \\ + 2 \cdot K_r \cdot \left(-4 \cdot T_i \cdot (T_d + T_a) + T^2\right) \cdot U(z) \cdot z^{-1} \\ + K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) - T) - 2 \cdot T_a \cdot T + T^2\right) \cdot U(z) \cdot z^{-2} \end{array}}{\begin{array}{c} 2 \cdot T_i \cdot (2 \cdot T_a + T) \cdot Y(z) \\ - 8 \cdot T_i \cdot T_a \cdot Y(z) \cdot z^{-1} \\ + 2 \cdot T_i \cdot (2 \cdot T_a - T) \cdot Y(z) \cdot z^{-2} \end{array}}$$

Inverse z-Transformation durchgeführt:

$$1 = \frac{\begin{array}{c} K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) + T) + T \cdot (2 \cdot T_a + T)\right) \cdot u(k) \\ + 2 \cdot K_r \cdot \left(-4 \cdot T_i \cdot (T_d + T_a) + T^2\right) \cdot u(k-1) \\ + K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) - T) - 2 \cdot T_a \cdot T + T^2\right) \cdot u(k-2) \end{array}}{\begin{array}{c} 2 \cdot T_i \cdot (2 \cdot T_a + T) \cdot y(k) \\ - 8 \cdot T_i \cdot T_a \cdot y(k-1) \\ + 2 \cdot T_i \cdot (2 \cdot T_a - T) \cdot y(k-2) \end{array}}$$

Differenzengleichung nach $y(k)$ aufgelöst:

$$y(k) = \frac{1}{2 \cdot T_i \cdot (2 \cdot T_a + T)} \cdot \begin{pmatrix} K_r \cdot (2 \cdot (T_i \cdot (2 \cdot (T_d + T_a) - T) - T_a \cdot T) + T^2) \cdot u(k-2) \\ + 2 \cdot K_r \cdot (-4 \cdot T_i \cdot (T_d + T_a) + T^2) \cdot u(k-1) \\ + K_r \cdot \left(2 \cdot T_i \cdot (2 \cdot (T_d + T_a) + T) + T \cdot (2 \cdot T_a + T)\right) \cdot u(k) \\ - 2 \cdot T_i \cdot (2 \cdot T_a - T) \cdot y(k-2) \\ + 8 \cdot T_i \cdot T_a \cdot y(k-1) \end{pmatrix}$$

Programmcode

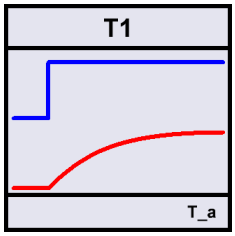
```
'#### PIDT1-Übertragungsglied ####
Public Class PIDT1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property K_r As Double = 0.4 'Regler-Verstärkung
    Public Property T_i As Double = 1 'Integrationszeitkonstante bzw. Nachstellzeit
    Public Property T_d As Double = 2 'Differentiationskonstante bzw. Vorhaltzeit
    Public Property T_a As Double = 0.5 'Zeitkonstante des Differenzierers

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / (T_i * T_a) *
                           (K_r * T_i * (T_d + T_a) * u(k) _
                           + K_r * (T_i * (T - 2 * (T_d + T_a)) + T_a * T) * u(k - 1) _
                           + K_r * (T_i * (T_d + T_a - T) - T * (T_a - T)) * u(k - 2) _
                           + T_i * (2 * T_a - T) * y(k - 1) _
                           - T_i * (T_a - T) * y(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_i * (T_a + T)) *
                           (K_r * T_i * (T_d + T_a) * u(k - 2) _
                           - K_r * (T_i * (2 * (T_d + T_a) + T) + T_a * T) * u(k - 1) _
                           + K_r * (T_i * (T_d + T_a + T) + T * (T_a + T)) * u(k) _
                           - T_i * T_a * y(k - 2) _
                           + T_i * (2 * T_a + T) * y(k - 1))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (2 * T_i * (2 * T_a + T)) *
                           (K_r * (2 * (T_i * (2 * (T_d + T_a) - T) - T_a * T) + T ^ 2) * u(k - 2) _
                           + 2 * K_r * (-4 * T_i * (T_d + T_a) + T ^ 2) * u(k - 1) _
                           + K_r * (2 * T_i * (2 * (T_d + T_a) + T) + T * (2 * T_a + T)) * u(k) _
                           - 2 * T_i * (2 * T_a - T) * y(k - 2) _
                           + 8 * T_i * T_a * y(k - 1))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.5 Verzögerungsglied 1. Ordnung (Controller.T1)



Das „T1“-Glied ist ein Übertragungsglied, welches ein verzögerndes Übertragungsverhalten der 1. Ordnung aufweist. Dies bedeutet, dass die Steigung der Ausgangsgröße proportional abhängig von der Differenz zwischen Eingangs- und Ausgangswert ist. Beispiele für Systeme, die ein solches Verhalten zeigen, sind in der Elektrotechnik der Tiefpass und im Maschinenbau ein Wärmespeicher, bei dem ein Temperaturausgleich stattfindet.

Eigenschaften

Übertragungsfunktion: $G(s) = \frac{1}{T_a \cdot s + 1}$

Parameter: T_a = Zeitkonstante

Funktionsblock-Testergebnis

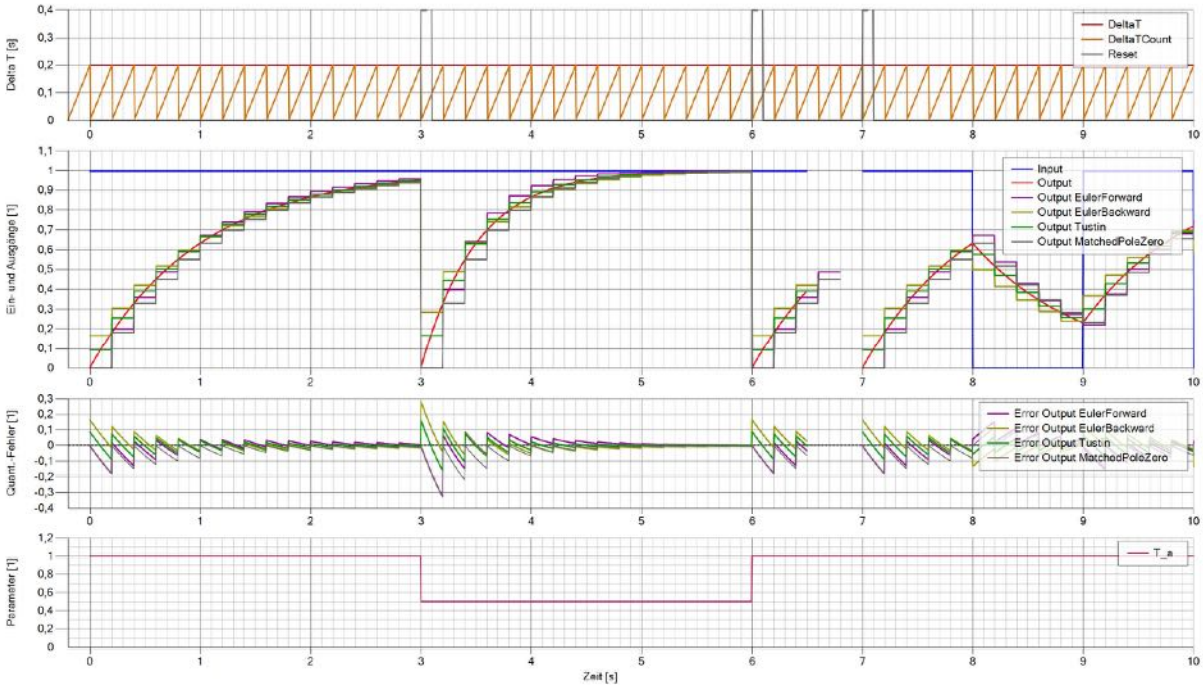


Abb. 3-17: Funktionsblock-Testergebnis: Verzögerungsglied 1. Ordnung

0 - 3 Sprungantwort auf das Eingangssignal $Input = 1$ bei $T_a = 1$

3 - 6 Sprungantwort auf das Eingangssignal $Input = 1$ bei $T_a = 0{,}5$

6 - 7 Reaktion auf $Input = NaN$

7 - 8 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

8 - 10 Wertesprünge am Eingang (Ausgang schwankt um den Mittelwert)

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = \frac{1}{T_a \cdot s + 1} \cdot U(s)$

$$T_a \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = u(t)$$

Euler-Vorwärts: $Y(z) = \frac{T}{T_a \cdot z - T_a + T} \cdot U(z)$

$$y(k) = \frac{T}{T_a} \cdot \big(u(k-1) - y(k-1)\big) + y(k-1)$$

Euler-Rückwärts: $Y(z) = \frac{T \cdot z}{(T_a + T) \cdot z - T_a} \cdot U(z)$

$$y(k) = \frac{1}{T_a + T} \cdot \big(T_a \cdot y(k-1) + T \cdot u(k)\big)$$

Tustin-Methode: $Y(z) = \frac{T \cdot z + T}{(2 \cdot T_a + T) \cdot z - 2 \cdot T_a + T} \cdot U(z)$

$$y(k) = \frac{1}{2 \cdot T_a + T} \cdot \big((2 \cdot T_a - T) \cdot y(k-1) + T \cdot u(k-1) + T \cdot u(k)\big)$$

Matched-Pole-Zero: $Y(z) = \left(1 - e^{-\frac{1}{T_a} \cdot T}\right) \cdot \frac{1}{z - e^{-\frac{1}{T_a} T}} \cdot U(z)$

$$y(k) = \left(1 - e^{-\frac{1}{T_a} T}\right) \cdot u(k-1) + e^{-\frac{1}{T_a} T} \cdot y(k\ -\ 1)$$

Programmcode

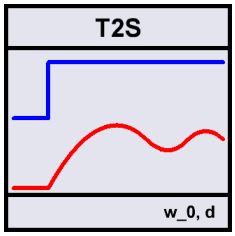
```
'#### Verzögerungsglied 1. Ordnung ####
Public Class T1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_a As Double = 1 'Zeitkonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T / T_a * (u(k - 1) - y(k - 1)) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_a + T) * (T_a * y(k - 1) + T * u(k))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (2 * T_a + T) * ((2 * T_a - T) * y(k - 1) + T * u(k - 1) + T * u(k))
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return (1 - Math.Exp(-1 / T_a * T)) * u(k - 1) + Math.Exp(-1 / T_a * T) * y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.6 Verzögerungsglied 2. Ordnung (Controller.T2S)



Das „T2S“-Glied ist ein schwingungsfähiges Übertragungsglied, welches ein verzögerndes Übertragungsverhalten der Verzögerung 2. Ordnung aufweist. Beispiele für Systeme, die ein solches Verhalten zeigen, sind in der Elektrotechnik der RLC-Schwingkreis und im Maschinenbau das Federpendel.

Eigenschaften

Übertragungsfunktion: $G(s) = \frac{1}{\frac{1}{\omega_0^2} \cdot s^2 + \frac{2 \cdot d}{\omega_0} \cdot s + 1}$

Parameter: ω_0 = Kennkreisfrequenz

d = Dämpfungskonstante

Funktionsblock-Testergebnis

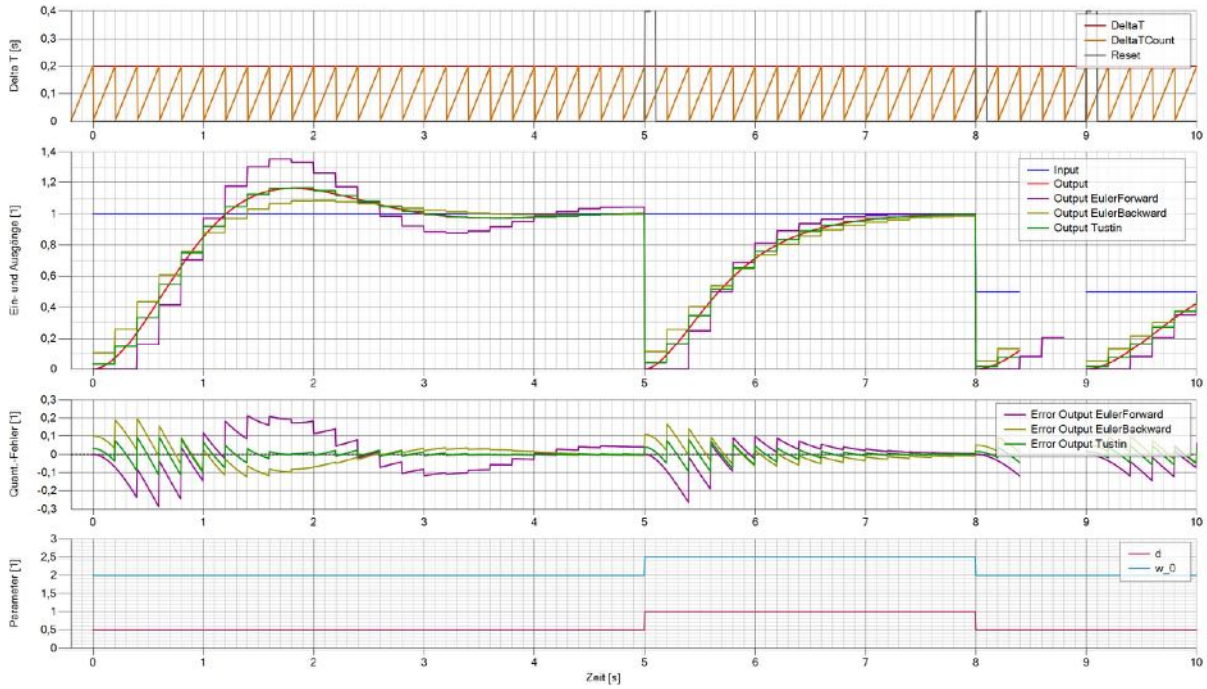


Abb. 3-18: Funktionsblock-Testergebnis: Verzögerungsglied 2. Ordnung

0 - 5 Sprungantwort auf das Eingangssignal $Input = 1$ bei $\omega_0 = 2$ und $d = 0{,}5$

5 - 8 Sprungantwort auf das Eingangssignal $Input = 1$ bei $\omega_0 = 2{,}5$ und $d = 1$ (Aperiodischer Grenzfall)

8 - 9 Reaktion auf $Input = NaN$

9 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = \frac{1}{\frac{1}{\omega_0^2}\cdot s^2 + \frac{2\cdot d}{\omega_0}\cdot s + 1} \cdot U(s)$

$$\frac{1}{\omega_0^2} \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) + \frac{2\cdot d}{\omega_0} \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = u(t)$$

Substitution: $a = \frac{1}{\omega_0^2 \cdot T^2} \qquad b = \frac{2\cdot d}{\omega_0 \cdot T}$

Euler-Vorwärts: $Y(z) = \frac{1}{a\cdot z^2 - (2\cdot a - b)\cdot z + a - b + 1} \cdot U(z)$

$$y(k) = \frac{1}{a} \cdot \big((b - a - 1) \cdot y(k-2) + (2 \cdot a - b) \cdot y(k-1) + u(k-2)\big)$$

Euler-Rückwärts: $Y(z) = \frac{z^2}{(a+b+1)\cdot z^2 - (2\cdot a + b)\cdot z + a} \cdot U(z)$

$$y(k) = \frac{1}{a+b+1} \cdot \big((2 \cdot a + b) \cdot y(k-1) - a \cdot y(k-2) + u(k)\big)$$

Tustin-Methode: $Y(z) = \frac{z^2 + 2\cdot z + 1}{(4\cdot a + 2\cdot b + 1)\cdot z^2 - (8\cdot a - 2)\cdot z + 4\cdot a - 2\cdot b + 1} \cdot U(z)$

$$y(k) = \frac{1}{4\cdot a + 2\cdot b + 1} \cdot \begin{pmatrix} (2 \cdot b - 4 \cdot a - 1) \cdot y(k-2) \\ +(8 \cdot a - 2) \cdot y(k-1) \\ +u(k-2) + 2 \cdot u(k-1) + u(k) \end{pmatrix}$$

Programmcode

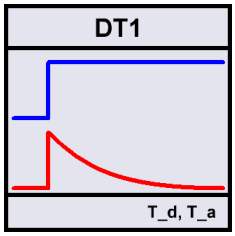
```
'#### Verzögerungsglied 2. Ordnung ####
Public Class T2S
    Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum 'Unterstützte Approximationstypen
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property w_0 As Double = 2 'Kennkreisfrequenz
    Public Property d As Double = 0.5 'Dämpfungskonstante

    Protected Overrides Function Functionality() As Double 'Funktionalität des Funktionsblocks
        Dim a As Double = 1 / (w_0 ^ 2 * T ^ 2) 'Substitution
        Dim b As Double = (2 * d) / (w_0 * T) 'Substitution
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / a * ((b - a - 1) * y(k - 2) + (2 * a - b) * y(k - 1) + u(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (a + b + 1) * ((2 * a + b) * y(k - 1) - a * y(k - 2) + u(k))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (4 * a + 2 * b + 1) * ((2 * b - 4 * a - 1) * y(k - 2) _
                                                + (8 * a - 2) * y(k - 1) + u(k - 2) _
                                                + 2 * u(k - 1) + u(k))
            Case Else
                Return Double.NaN
        End Select
    End Function
End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.7 Verzögernder Differenzierer (Controller.DT1)



Das „DT1“-Glied ist ein Übertragungsglied, welches ein differenzierendes Übertragungsverhalten mit Verzögerung 1. Ordnung aufweist. Ein Beispiel für Systeme, die ein solches Verhalten zeigen, ist in der Elektrotechnik der Hochpass aus den Komponenten Kondensator und Widerstand.

Eigenschaften

Übertragungsfunktion: $G(s) = \frac{T_d \cdot s}{T_a \cdot s + 1}$

Parameter: T_d = Differentiationskonstante

T_a = Zeitkonstante

Funktionsblock-Testergebnis

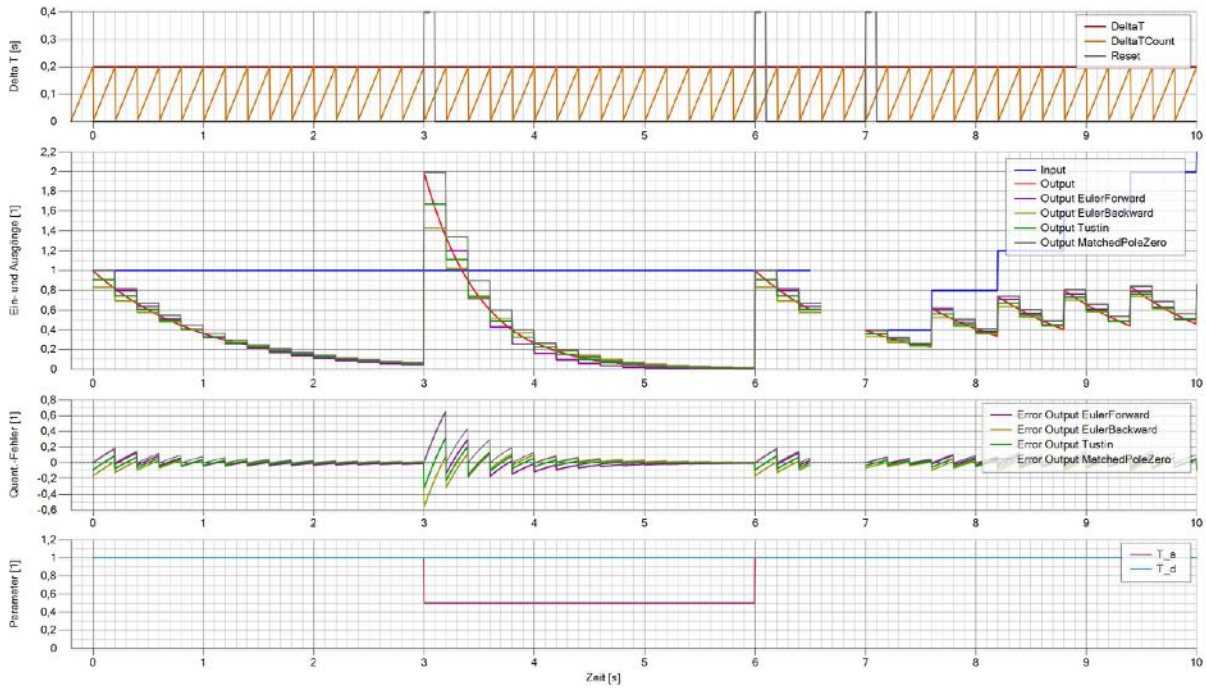


Abb. 3-19: Funktionsblock-Testergebnis: Verzögernder Differenzierer

0 - 3 Sprungantwort auf das Eingangssignal $Input = 1$ bei $T_d = 1$ und $T_a = 1$

3 - 6 Sprungantwort auf das Eingangssignal $Input = 1$ bei $T_d = 1$ und $T_a = 0{,}5$
(Fläche unter Sprungantwort bleibt gleich)

6 - 7 Reaktion auf $Input = NaN$

7 - 8 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

8 - 10 Positive Wertesprünge mit jeweils einer Flankenhöhe von $0{,}4$ am Eingang

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = \frac{T_d \cdot s}{T_a \cdot s + 1} \cdot U(s)$

$$T_a \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = T_d \cdot \frac{\mathrm{d}}{\mathrm{d}t} u(t)$$

Euler-Vorwärts: $Y(z) = \frac{T_d \cdot z - T_d}{T_a \cdot z - T_a + T} \cdot U(z)$

$$y(k) = \frac{T_d}{T_a} \cdot \big(u(k) - u(k-1)\big) + \left(1 - \frac{T}{T_a}\right) \cdot y(k-1)$$

Euler-Rückwärts: $Y(z) = \frac{T_d \cdot z - T_d}{(T_a + T) \cdot z - T_a} \cdot U(z)$

$$y(k) = \frac{1}{T + T_a} \cdot \Big(T_d \cdot \big(u(k) - u(k-1)\big) + T_a \cdot y(k-1)\Big)$$

Tustin-Methode: $Y(z) = \frac{T_d \cdot z - T_d}{\left(T_a + \frac{T}{2}\right) \cdot z - T_a + \frac{T}{2}} \cdot U(z)$

$$y(k) = \frac{1}{T + 2 \cdot T_a} \cdot \Big(T_d \cdot \big(2 \cdot u(k) - 2 \cdot u(k-1)\big) - (T - 2 \cdot T_a) \cdot y(k\ -\ 1)\Big)$$

Matched-Pole-Zero: $Y(z) = \frac{T_d}{T_a} \cdot \frac{z - 1}{z - e^{-\frac{1}{T_a} \cdot T}} \cdot U(z)$

$$y(k) = \frac{T_d}{T_a} \cdot \big(u(k) - u(k-1)\big) + e^{-\frac{1}{T_a} \cdot T} \cdot\ y(k-1)$$

Programmcode

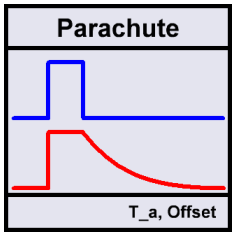
```
'#### Verzögernder Differenzierer ####
Public Class DT1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_d As Double = 1 'Differentiationskonstante
    Public Property T_a As Double = 1 'Zeitkonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T_d / T_a * (u(k) - u(k - 1)) + (1 - T / T_a) * y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T + T_a) * (T_d * (u(k) - u(k - 1)) + T_a * y(k - 1))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (T + 2 * T_a) * (T_d * (2 * u(k) - 2 * u(k - 1)) - (T - 2 * T_a) * y(k - 1))
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return T_d / T_a * (u(k) - u(k - 1)) + Math.Exp(-1 / T_a * T) * y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function
End Class
```

Verweise

Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.1.8 Fallschirm (Controller.Parachute)



Das „Parachute“-Glied ist ein Übertragungsglied, welches abhängig von der Differenz aus Ein- und Ausgangswert ein anderes Übertragungsverhalten aufweist. Ist der Eingangswert größer oder gleich dem Ausgangswert, so folgt es dem Signalverlauf. Ist jedoch der Ausgangswert kleiner als der Eingangswert, verhält sich das Übertragungsglied wie ein „T1“-Glied (Seite 59) und nähert seinen Ausgangswert langsam an den kleineren Eingangswert an.

Eigenschaften

Übertragungsfunktion: $G(s) = \begin{cases} 1 & wenn\ u(t) \geq y(t) \\ \frac{1}{T_a \cdot s+1} & wenn\ u(t) < y(t) \end{cases}$ $bei\ Offset = 0$

Parameter:
T_a = Zeitkonstante
$Offset$ = Versatz des Bezugspunkts

Funktionsblock-Testergebnis

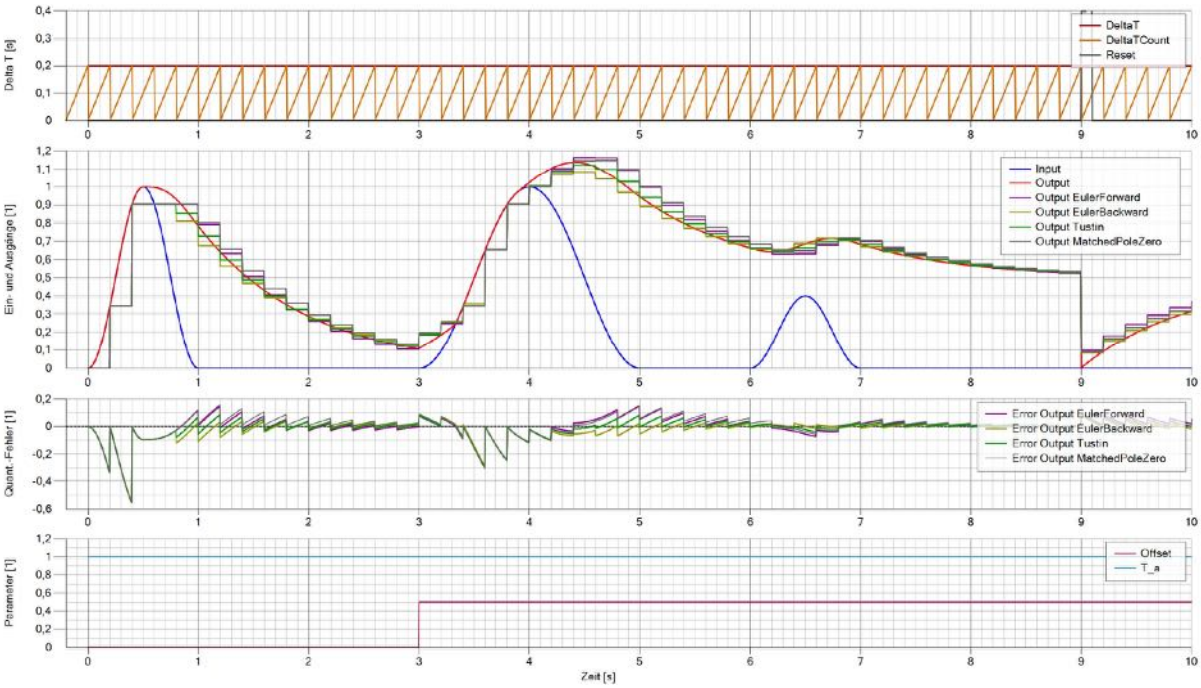


Abb. 3-20: Funktionsblock-Testergebnis: Fallschirm

0 - 3 Verhalten bei $T_a = 1$ und $Offset = 0$
(Der Ausgangswert nähert sich dem Eingangswert $Input$ mit der Zeitkonstante T_a. Außer wenn das Ausgangssignal überschritten wird, folgt es dem Eingangswert.)

3 - 9 Verhalten bei $T_a = 1$ und $Offset = 0{,}5$
(Der Ausgangswert nähert sich dem Wert $Input + Offset$ mit der Zeitkonstante T_a. Außer wenn das Ausgangssignal eingeholt wird, folgt es dem Eingangswert.)

9 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Mathematischer Hintergrund

Die Formeln beschreiben das Verhalten im Fall $u(t) < y(t)$ und $Offset = 0$

Zeitkontinuierlich:
$$Y(s) = \frac{1}{T_a \cdot s + 1} \cdot U(s)$$
$$T_a \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = u(t)$$

Euler-Vorwärts:
$$Y(z) = \frac{T}{T_a \cdot z - T_a + T} \cdot U(z)$$
$$y(k) = \frac{T}{T_a} \cdot \big(u(k-1) - y(k-1)\big) + y(k-1)$$

Euler-Rückwärts:
$$Y(z) = \frac{T \cdot z}{(T_a + T) \cdot z - T_a} \cdot U(z)$$
$$y(k) = \frac{1}{T_a + T} \cdot \big(T_a \cdot y(k-1) + T \cdot u(k)\big)$$

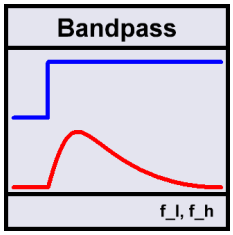
Tustin-Methode:
$$Y(z) = \frac{T \cdot z + T}{(2 \cdot T_a + T) \cdot z - 2 \cdot T_a + T} \cdot U(z)$$
$$y(k) = \frac{1}{2 \cdot T_a + T} \cdot \big((2 \cdot T_a - T) \cdot y(k-1) + T \cdot u(k-1) + T \cdot u(k)\big)$$

Matched-Pole-Zero:
$$Y(z) = \left(1 - e^{-\frac{1}{T_a} \cdot T}\right) \cdot \frac{1}{z - e^{-\frac{1}{T_a} \cdot T}} \cdot U(z)$$
$$y(k) = \left(1 - e^{-\frac{1}{T_a} \cdot T}\right) \cdot u(k-1) + e^{-\frac{1}{T_a} \cdot T} \cdot y(k\ -\ 1)$$

Programmcode

```
'#### Fallschirm ####
Public Class Parachute
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_a As Double = 1 'Zeitkonstante
    Public Property Offset As Double = 0 'Versatz des Bezugspunkts

    Protected Overrides Function Functionality() As Double
        Dim OutputTmp As Double = Double.NaN
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                OutputTmp = T / T_a * ((u(k - 1) + Offset) - y(k - 1)) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                OutputTmp = 1 / (T_a + T) * (T_a * y(k - 1) + T * (u(k) + Offset))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                OutputTmp = 1 / (2 * T_a + T) * ((2 * T_a - T) * y(k - 1) + _
                                                 T * (u(k - 1) + Offset) + T * (u(k) + Offset))
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                OutputTmp = (1 - Math.Exp(-1 / T_a * T)) * (u(k - 1) + Offset) +
                            Math.Exp(-1 / T_a * T) * y(k - 1)
        End Select
        If u(k - 0) > OutputTmp Then OutputTmp = u(k - 0) 'Fallschirm anheben
        Return OutputTmp
    End Function

End Class
```

3.1.9 Bandpass (Controller.Bandpass)



Der Bandpass setzt sich aus einem Tiefpass- und einem Hochpassfilter zusammen, die in Serie geschaltet sind. Bandpässe werden zum Filtern von Frequenzspektren, die mit einer unteren und einer oberen Grenzfrequenz angegeben werden, eingesetzt. Frequenzen, die in diesem Bereich liegen, lässt der Bandpass nahezu ungedämpft durch. Frequenzen, die außerhalb dieses Frequenzspektrums liegen, werden hingegen mit 20 dB pro Dekade gedämpft.

Eigenschaften

Übertragungsfunktion: $$G(s) = \underbrace{\frac{1}{T_l \cdot s+1}}_{Tiefpass} \cdot \underbrace{\frac{T_h \cdot s}{T_h \cdot s+1}}_{Hochpass} = \underbrace{\frac{T_h \cdot s}{T_h \cdot T_l \cdot s^2+(T_h+T_l) \cdot s+1}}_{Bandpass}$$

Parameter:

f_l = Untere Grenzfrequenz $= \frac{1}{2 \cdot \pi \cdot T_h}$ (beachte $l \rightsquigarrow h$)

f_h = Obere Grenzfrequenz $= \frac{1}{2 \cdot \pi \cdot T_l}$ (beachte $h \rightsquigarrow l$)

Funktionsblock-Testergebnis

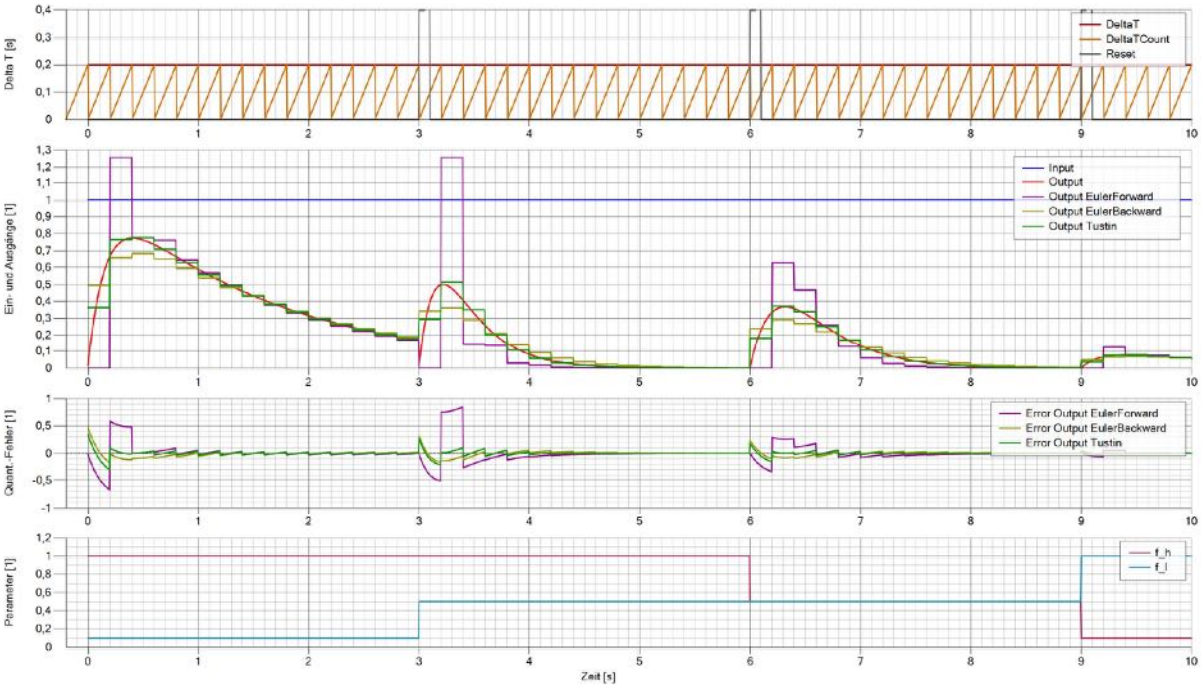


Abb. 3-21: Funktionsblock-Testergebnis: Bandpass

0 - 3 Sprungantwort bei $f_l = 0{,}1$ und $f_h = 1$ (Grenzfrequenzen liegen weit auseinander)

3 - 6 Sprungantwort bei $f_l = 0{,}5$ und $f_h = 1$ (Grenzfrequenzen liegen auseinander)

6 - 9 Sprungantwort bei $f_l = 0{,}5$ und $f_h = 0{,}5$ (Grenzfrequenzen liegen aufeinander)

9 - 10 Sprungantwort bei $f_l = 1$ und $f_h = 0{,}1$ (Grenzfrequenzen überschneiden sich)

3, 6, 9 Zurücksetzen durch Aufrufen der Funktion „*Reset*“

Mathematischer Hintergrund

Zeitkontinuierlich: $Y(s) = \frac{T_h \cdot s}{T_h \cdot T_l \cdot s^2 + (T_h + T_l) \cdot s + 1} \cdot U(s)$

$$T_h \cdot T_l \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) + (T_h + T_l) \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = T_h \cdot u(t)$$

Substitution: $T_l = \frac{1}{2 \cdot \pi \cdot f_h}$ $\quad T_h = \frac{1}{2 \cdot \pi \cdot f_l}$ $\quad$ (beachte $l \leftrightsquigarrow h$ bzw. $h \leftrightsquigarrow l$)

Euler-Vorwärts: $Y(z) = \frac{T \cdot T_h \cdot z - T \cdot T_h}{T_l \cdot T_h \cdot z^2 + (-2 \cdot T_l \cdot T_h + T \cdot T_l + T \cdot T_h) \cdot z + (T_l \cdot T_h - T \cdot T_l - T \cdot T_h + T^2)} \cdot U(z)$

$$y(k) = \frac{1}{T_l \cdot T_h} \cdot \begin{pmatrix} T \cdot T_h \cdot \big(u(k-1) - u(k-2)\big) \\ +\big(2 \cdot T_l \cdot T_h - T \cdot (T_l + T_h)\big) \cdot y(k-1) \\ -(T_l \cdot T_h - T \cdot (T_l + T_h) + T^2) \cdot y(k-2) \end{pmatrix}$$

Euler-Rückwärts: $Y(z) = \frac{T \cdot T_h \cdot z^2 - T \cdot T_h \cdot z}{(T_l \cdot T_h + T \cdot T_l + T \cdot T_h + T^2) \cdot z^2 + (-2 \cdot T_l \cdot T_h - T \cdot T_l - T \cdot T_h) \cdot z + (T_l \cdot T_h)} \cdot U(z)$

$$y(k) = \frac{1}{T_l \cdot T_h + T \cdot (T_l + T_h) + T^2} \cdot \begin{pmatrix} T \cdot T_h \cdot \big(u(k) - u(k-1)\big) \\ +\big(2 \cdot T_l \cdot T_h + T \cdot (T_l + T_h)\big) \cdot y(k-1) \\ -(T_l \cdot T_h) \cdot y(k-2) \end{pmatrix}$$

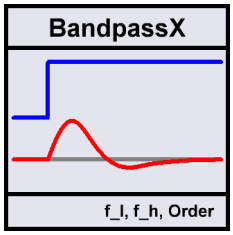
Tustin-Methode: $Y(z) = \frac{2 \cdot T \cdot T_h \cdot z^2 \cdot z - 2 \cdot T_h \cdot T}{(4 \cdot T_l \cdot T_h + 2 \cdot T \cdot T_l + 2 \cdot T \cdot T_h + T^2) \cdot z^2 + (-8 \cdot T_l \cdot T_h + 2 \cdot T^2) \cdot z + (4 \cdot T_l \cdot T_h - 2 \cdot T \cdot T_l - 2 \cdot T_h \cdot T + T^2)} \cdot U(z)$

$$y(k) = \frac{1}{4 \cdot T_l \cdot T_h + T \cdot (2 \cdot (T_l + T_h) + T)} \cdot \begin{pmatrix} 2 \cdot T \cdot T_h \cdot \big(u(k) - u(k-2)\big) \\ +(8 \cdot T_l \cdot T_h - 2 \cdot T^2) \cdot y(k-1) \\ -\big(4 \cdot T_l \cdot T_h - T \cdot (2 \cdot (T_l + T_h) - T)\big) \cdot y(k-2) \end{pmatrix}$$

Programmcode

```
'#### Bandpass ####
Public Class Bandpass
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property f_l As Double = 0.1 'Untere Grenzfrequenz
    Public Property f_h As Double = 1 'Obere Grenzfrequenz

    Protected Overrides Function Functionality() As Double
        Dim T_l As Double = 1 / (2 * Math.PI * f_h) : Dim T_h As Double = 1 / (2 * Math.PI * f_l)
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
        Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / (T_l * T_h) * (T * T_h * (u(k - 1) - u(k - 2)) _
                                        + (2 * T_l * T_h - T * (T_l + T_h)) * y(k - 1) _
                                        - (T_l * T_h - T * (T_l + T_h) + T ^ 2) * y(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_l * T_h + T * (T_l + T_h) + T ^ 2) * (
                                        T * T_h * (u(k) - u(k - 1)) _
                                        + (2 * T_l * T_h + T * (T_l + T_h)) * y(k - 1) _
                                        - (T_l * T_h) * y(k - 2))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (4 * T_l * T_h + T * (2 * (T_l + T_h) + T)) * (
                                        2 * T * T_h * (u(k) - u(k - 2)) _
                                        + (8 * T_l * T_h - 2 * T ^ 2) * y(k - 1) _
                                        - (4 * T_l * T_h - T * (2 * (T_l + T_h) - T)) * y(k - 2))
            Case Else
                Return Double.NaN
        End Select
    End Function
End Class
```

3.1.10 Bandpass höherer Ordnung (Controller.BandpassX)



Der Funktionsbaustein „BandpassX“ basiert auf dem Funktionsbaustein „Bandpass“ und erweitert diesen um den Parameter „Order“, mit dem ein Bandpass mit beliebiger Ordnung erstellt werden kann. Der Parameter gibt genauer an, wie viele Bandpässe hintereinander geschaltet werden. Somit hat zum Beispiel der Parameterwert 4 einen Bandpass der 8. Ordnung zur Folge, der somit Frequenzen, die außerhalb des Spektrums liegen, mit 80 dB pro Dekade dämpft.

Eigenschaften

Übertragungsfunktion: $G(s) = \left(\frac{T_h \cdot s}{T_h \cdot T_l \cdot s^2 + (T_h + T_l) \cdot s + 1}\right)^{Order}$

Parameter:

f_l = Untere Grenzfrequenz $= \frac{1}{2 \cdot \pi \cdot T_h}$ (beachte $l \leftrightsquigarrow h$)

f_h = Obere Grenzfrequenz $= \frac{1}{2 \cdot \pi \cdot T_l}$ (beachte $h \leftrightsquigarrow l$)

$Order$ = Ordnung (z. B. 1 ≙ 2. Ordnung, 2 ≙ 4. Ordnung, ...)

Funktionsblock-Testergebnis

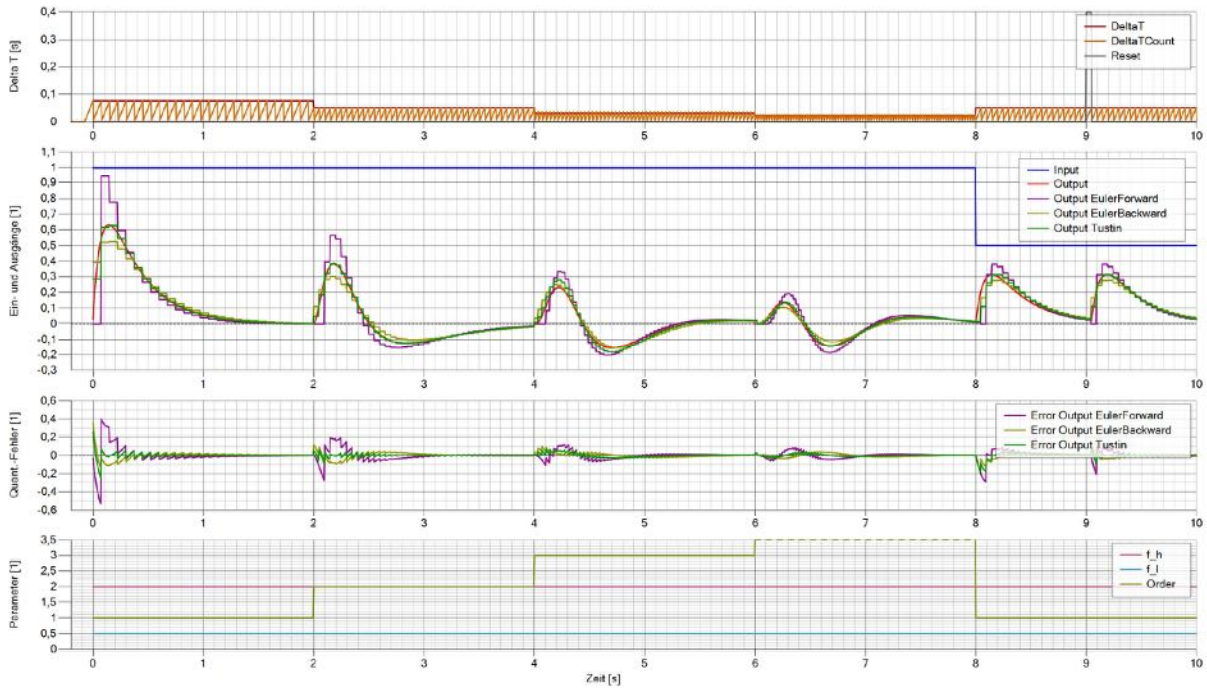


Abb. 3-22: Funktionsblock-Testergebnis: Bandpass höherer Ordnung

0 - 2 Sprungantwort bei $f_l = 0{,}5$ $f_h = 2$ $Order = 1$ ≙ 2. Ordnung

2 - 4 Sprungantwort bei $f_l = 0{,}5$ $f_h = 2$ $Order = 2$ ≙ 4. Ordnung

4 - 6 Sprungantwort bei $f_l = 0{,}5$ $f_h = 2$ $Order = 3$ ≙ 6. Ordnung

6 - 8 Sprungantwort bei $f_l = 0{,}5$ $f_h = 2$ $Order = 4$ ≙ 8. Ordnung

8 - 10 Zurücksetzen durch Aufrufen der Funktion „*Reset*“

Mathematischer Hintergrund

Die Formeln beschreiben das Verhalten im Fall $Order = 1 \;\widehat{=}$ Bandpass 2. Ordnung

Zeitkontinuierlich: $$Y(s) = \left(\frac{T_h \cdot s}{T_h \cdot T_l \cdot s^2 + (T_h + T_l) \cdot s + 1}\right)^1 \cdot U(s)$$

$$T_h \cdot T_l \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) + (T_h + T_l) \cdot \frac{\mathrm{d}}{\mathrm{d}t} y(t) + y(t) = T_h \cdot u(t)$$

Substitution: $$T_l = \frac{1}{2 \cdot \pi \cdot f_h}\ ,\ T_h = \frac{1}{2 \cdot \pi \cdot f_l}$$

Euler-Vorwärts: $$Y(z) = \frac{T \cdot T_h \cdot z - T \cdot T_h}{T_l \cdot T_h \cdot z^2 + (-2 \cdot T_l \cdot T_h + T \cdot T_l + T \cdot T_h) \cdot z + (T_l \cdot T_h - T \cdot T_l - T \cdot T_h + T^2)} \cdot U(z)$$

$$y(k) = \frac{1}{T_l \cdot T_h} \cdot \begin{pmatrix} T \cdot T_h \cdot \big(u(k-1) - u(k-2)\big) \\ +\big(2 \cdot T_l \cdot T_h - T \cdot (T_l + T_h)\big) \cdot y(k-1) \\ -(T_l \cdot T_h - T \cdot (T_l + T_h) + T^2) \cdot y(k-2) \end{pmatrix}$$

Euler-Rückwärts: $$Y(z) = \frac{T \cdot T_h \cdot z^2 - T \cdot T_h \cdot z}{(T_l \cdot T_h + T \cdot T_l + T \cdot T_h + T^2) \cdot z^2 + (-2 \cdot T_l \cdot T_h - T \cdot T_l - T \cdot T_h) \cdot z + (T_l \cdot T_h)} \cdot U(z)$$

$$y(k) = \frac{1}{T_l \cdot T_h + T \cdot (T_l + T_h) + T^2} \cdot \begin{pmatrix} T \cdot T_h \cdot \big(u(k) - u(k-1)\big) \\ +\big(2 \cdot T_l \cdot T_h + T \cdot (T_l + T_h)\big) \cdot y(k-1) \\ -(T_l \cdot T_h) \cdot y(k-2) \end{pmatrix}$$

Tustin-Methode: $$Y(z) = \frac{2 \cdot T \cdot T_h \cdot z^2 \cdot z - 2 \cdot T_h \cdot T}{(4 \cdot T_l \cdot T_h + 2 \cdot T \cdot T_l + 2 \cdot T \cdot T_h + T^2) \cdot z^2 + (-8 \cdot T_l \cdot T_h + 2 \cdot T^2) \cdot z + (4 \cdot T_l \cdot T_h - 2 \cdot T \cdot T_l - 2 \cdot T_h \cdot T + T^2)} \cdot U(z)$$

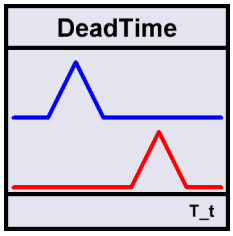
$$y(k) = \frac{1}{4 \cdot T_l \cdot T_h + T \cdot (2 \cdot (T_l + T_h) + T)} \cdot \begin{pmatrix} 2 \cdot T \cdot T_h \cdot \big(u(k) - u(k-2)\big) \\ +(8 \cdot T_l \cdot T_h - 2 \cdot T^2) \cdot y(k-1) \\ -\big(4 \cdot T_l \cdot T_h - T \cdot (2 \cdot (T_l + T_h) - T)\big) \cdot y(k-2) \end{pmatrix}$$

Programmcode

Der vollständige Programmcode ist im Anhang zu finden.

```
'#### Bandpass höherer Ordnung ####
Public Class BandpassX
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property f_l As Double = (New Bandpass).f_l 'Untere Grenzfrequenz
    Public Property f_h As Double = (New Bandpass).f_h 'Obere Grenzfrequenz
    Public Property Order As Double = 2 'Ordnung (1 => 2. Ordnung, 2 => 4. Ordnung, 3 => 6. Ordnung...)

    Private Bandpasss() As Bandpass 'Alle Bandpässe (pro 2 Ordnungen 1 Bandpass)

    Protected Overrides Function Functionality() As Double
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
        'Bandpässe neu instanziieren, falls sich die Ordnung geändert hat.
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
        'Parameter aktualisieren, falls sich einer geändert hat.
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
        'Ausgangsfunktion der Bandpässe aufrufen (Alle Bandpässe in Serie)
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    End Function
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
End Class
```

3.1.11 Totzeitglied (Controller.DeadTime)



Das Totzeitglied verzögert das Eingangssignal um eine über den Parameter „T_t“ einstellbare Zeit. Hierzu speichert der Funktionsblock beim Aufruf der Ausgangsfunktion den Eingangswert zusammen mit einem Zeitstempel ab und gibt einen verzögerten Wert zurück. Bei zeitäquidistanten Funktionsaufrufen wird selten ein Wert im Speicher vorliegen, der genau um die Zeit „T_t“ verzögert wurde. Deshalb gibt die Ausgangsfunktion je nach eingestelltem Approximations-Typ den nächst jüngeren oder älteren Wert zurück. Ein dritter Approximations-Typ führt eine lineare Interpolation durch und berechnet Werte mit möglichst kleinen Quantisierungsfehlern.

Eigenschaften

Übertragungsfunktion: $G(s) = e^{-s \cdot T_t}$

Parameter: T_t = Verzögerungszeit

Funktionsblock-Testergebnis

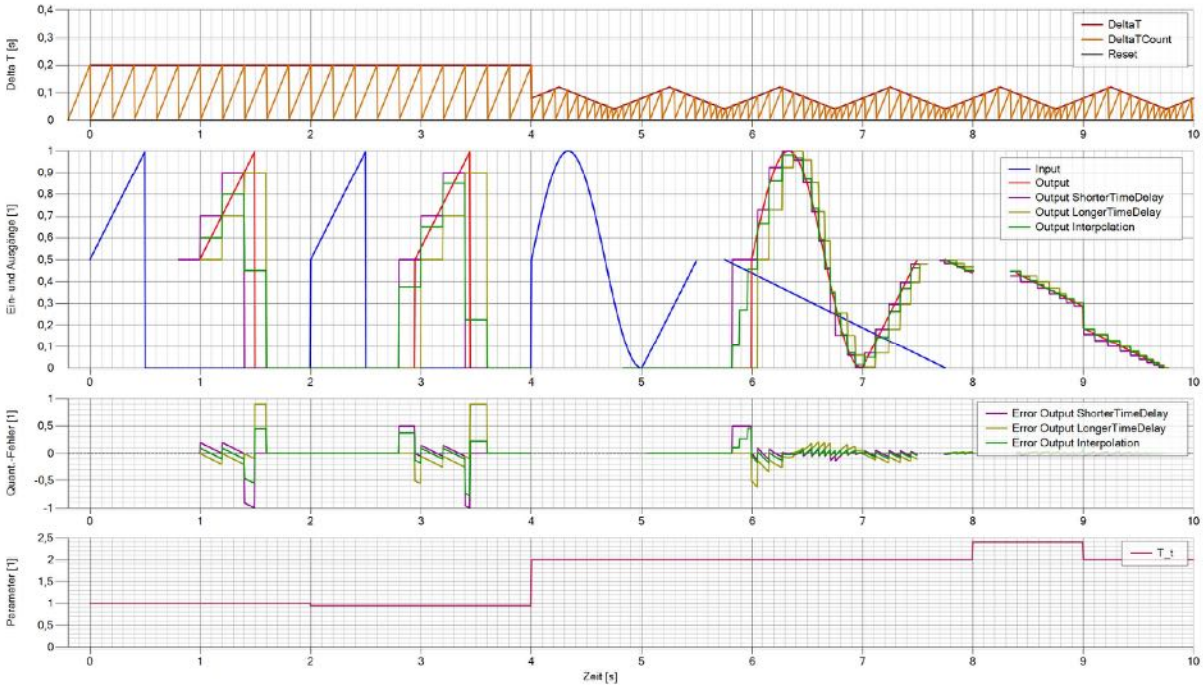


Abb. 3-23: Funktionsblock-Testergebnis: Totzeitglied

0 - 2 Verzögern des Eingangssignals um die Zeit $T_t = 1$

2 - 4 Verzögern des Eingangssignals um die Zeit $T_t = 0{,}95$
(Das gleiche Eingangssignal wurde mit den gleichen Werten abgetastet, soll nun aber um eine minimal kleinere Zeitspanne verzögert werden. Zu beachten ist, wie der Funktionsblock mit dem interpolierten Ausgangssignal (grün) versucht, möglichst geringe Quantisierungsfehler zu erreichen.

5 - 10 Reaktion auf $Input = NaN$ und Verstellen der Verzögerungszeit T_t zur Laufzeit

Mathematischer Hintergrund

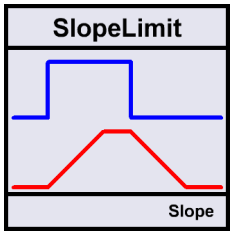
Zeitkontinuierlich: $Y(s) = e^{-s \cdot T_t} \cdot U(s)$

$$y(t) = u(t - T_t)$$

Programmcode

```
'#### Totzeitglied ####
Public Class DeadTime
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.ShorterTimeDelay
    Public Enum ApproxTypeEnum
        ShorterTimeDelay 'Kürzere Verzögerungszeit
        LongerTimeDelay 'Längere Verzögerungszeit
        Interpolation 'Genaueste Verzögerungszeit
    End Enum

    Property T_t As Double = 1 'Verzögerungszeit

    Private PQueue As New System.Collections.Queue() 'Alle gespeicherten Punkte
    Private Structure DeltaTAndValue 'Punkt bestehend aus Zykluszeit und Wert
        Property DeltaT As Double : Property Value As Double
        Sub New(ByVal DeltaT As Double, ByVal Value As Double)
            Me.DeltaT = DeltaT : Me.Value = Value
        End Sub
    End Structure

    Protected Overrides Function Functionality() As Double
        'Verzögern eines nicht zeitäquidistanten abgetasteten Signals.
        ' P5         P4     P?              P3            P2         P1         P0 (Punkte)
        ' |          |      !               |            |          |          !
        ' |-DeltaTP4-|----------DeltaTP3----|--DeltaTP2---|-DeltaTP1-|-DeltaTP0-!  (DeltaT einzeln)
        '                     !              |<------TimeAfter-------------------!  (DeltaT danach)
        '            |<-----------------------------TimeBefore------------------!  (DeltaT davor)
        '                   !<--Delay=-(T_t-SmallestDeltaT/2)-------------------!  (Verzögerungszeit)

        'Die Werte und dazugehörigen Zykluszeiten aller Funktionsblockaufrufe werden gespeichert.
        'Zum Verzögern des Signals wird im Verlauf nach einem T_t-SmallestDeltaT/2 alten Wert gesucht
        'und je nach Approximations-Typ der Wert davor oder danach oder eine Interpolation der beiden
        'Werte zurückgegeben. Dabei wird T_t um SmallestDeltaT/2 korrigiert, weil der ausgegebene Wert
        'voraussichtlich über eine Zykluszeit lang aktuell sein wird. Die Korrektur erfolgt um die im
        'Verlauf kleinste vorkommende Zykluszeit, da bei schwankenden Zykluszeiten sonst evtl. Werte
        'zurückgegeben werden, die älter sind als bereits zurückgegebene Werte. Nicht mehr benötigte
        'Werte werden aus dem Verlauf gelöscht.

        'Falls noch kein Punkt im Speicher ist, einen Punkt mit dem Wert NaN hinzufügen
        If PQueue.Count = 0 Then PQueue.Enqueue(New DeltaTAndValue(1 / 0, Double.NaN))
        'Aktuellen Punkt (DeltaT mit Wert) hinzufügen
        PQueue.Enqueue(New DeltaTAndValue(T, u(0))) 'T=Zykluszeit, u(0)=Eingangswert
        'Kleinstes DeltaT aller Punkte ermitteln
        Dim SmallestDeltaT As Double = Double.PositiveInfinity 'Kleinste Zykluszeit
        For Each DeltaTAndValue As DeltaTAndValue In PQueue 'Alle Punkte durchlaufen
            If DeltaTAndValue.DeltaT < SmallestDeltaT Then SmallestDeltaT = DeltaTAndValue.DeltaT
        Next
        'Queue für bessere Laufzeit in ein Array kopieren
        Dim PArray(PQueue.Count - 1) As DeltaTAndValue 'Array, das den Werteverlauf enthält
        Dim PIndex As Integer = PQueue.Count - 1 'Array-Index
        For Each DeltaTAndValue As DeltaTAndValue In PQueue 'Queue durchlaufen
            PArray(PIndex) = DeltaTAndValue 'Array rückwärts beschreiben (PArray(0) = Aktuellster)
            PIndex -= 1 'Array-Index dekrementieren
        Next
        'Ausgangswert berechnen
        Dim TimeBefore, TimeAfter As Double 'Zeitpunkt vor und nach dem gesuchten Punkt Px
        'Bis zur geforderten Verzögerungszeit zurückzählen, um einen verzögerten Wert auszugeben.
        For i As Integer = 1 To PArray.Count - 1 'Alle Index der Punkte durchlaufen
            'Zeitpunkt davor durch Aufsubtrahieren der P(x).DeltaT ermitteln
            TimeBefore -= PArray(i - 1).DeltaT 'Zeitpunkt negativ, weil er in der Vergangenheit liegt
            'Gesuchten Punkt ermitteln (T_t wird um DeltaTAverage/2 korrigiert (Beschreibung oben))
            Dim Delay As Double = -(T_t - SmallestDeltaT / 2)
            'Liegt der gesuchte Punkt in der Zukunft, suche den aktuellsten Punkt.
            If Delay > 0 Then Delay = 0
            'Liegt der gesuchte Punkt zwischen den beiden aktuellen Punkten (DateBefore und DateAfter)?
            If TimeBefore <= Delay And TimeAfter >= Delay Then '>= und <= weil Zykluszeit 0 sein kann
                'Überflüssige Punkte aus der Queue löschen
                'i+1, weil evtl. darauf zurückgegriffen wird
                For k As Integer = i + 1 To PArray.Count - 1
                    PQueue.Dequeue() 'Überflüssigen Punkt der Vergangenheit löschen
                Next
                Select Case DirectCast(ApproxType, ApproxTypeEnum)
                    Case ApproxTypeEnum.ShorterTimeDelay 'Kürzere Verzögerungszeit
                        Return PArray(i - 1).Value
                    Case ApproxTypeEnum.LongerTimeDelay 'Längere Verzögerungszeit
                        Return PArray(i).Value
                    Case ApproxTypeEnum.Interpolation 'Genaueste Verzögerungszeit
                        'Verhältnis für Interpolation
                        Dim Ratio As Double = (Delay - TimeBefore) / (TimeAfter - TimeBefore)
                        Return Ratio * PArray(i - 1).Value + (1 - Ratio) * PArray(i).Value
                End Select
            End If
            TimeAfter = TimeBefore 'Der Punkt davor wird im nächsten Durchlauf der Punkt danach sein.
        Next
        Return Double.NaN 'Der Punkt zur gesuchten Verzögerungszeit liegt außerhalb des Verlaufs.
    End Function

End Class
```

3.1.12 Steigungsbegrenzer (Controller.SlopeLimit)



Der Funktionsblock Steigungsbegrenzer nähert seinen Ausgangswert dem Eingangswert linear an und überschreitet dabei die über den Parameter „Slope“ einstellbare Steigung nicht.

Eigenschaften

Differentialgleichung: $\frac{d}{dt}y(t) = Slope \cdot sgn\big(u(t) - y(t)\big)$

Parameter: $Slope$ = Maximale Steigung

Funktionsblock-Testergebnis

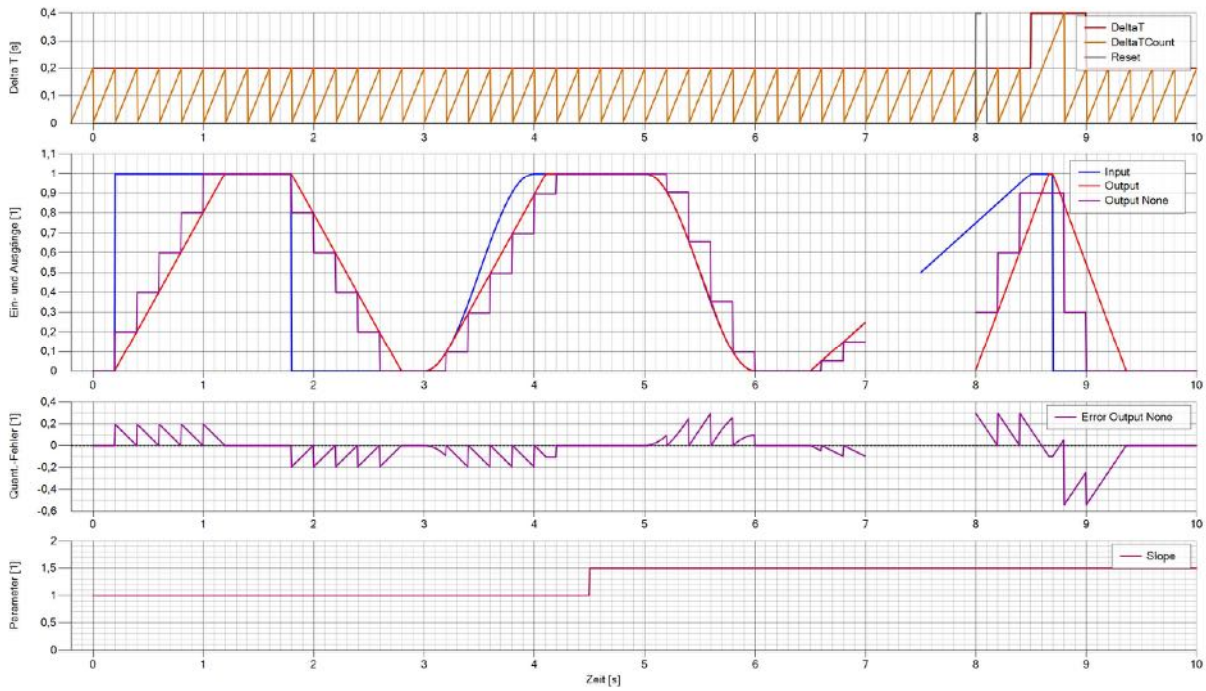


Abb. 3-24: Funktionsblock-Testergebnis: Steigungsbegrenzer

0 - 3 Ausgangsverhalten bei $Slope = 1$

3 - 6 Ausgangsverhalten bei $Slope = 1$ und $Slope = 1{,}5$

6 - 8 Reaktion auf $Input = NaN$ (Funktionsblock muss zurückgesetzt werden)

8 - 9 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

8 - 10 Doppelte Zykluszeiten verdoppeln die Sprungweite

Mathematischer Hintergrund

Zeitkontinuierlich: $\frac{\mathrm{d}}{\mathrm{d}t} y(t) = Slope \cdot sgn\big(u(t) - y(t)\big)$

Programmcode

```
'#### Steigungsbegrenzer ####
Public Class SlopeLimit
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.None
    Public Enum ApproxTypeEnum
        None 'Keine Approximation
    End Enum

    Public Property Slope As Double = 1 'Maximale Steigung

    Protected Overrides Function Functionality() As Double
        Dim Output As Double = y(k - 1) 'Ausgang speichern
        Dim Sign As Double = Double.NaN 'Richtung, in die der Ausgangswert laufen soll
        If y(k - 1) < u(k) Then Sign = 1 'Positive Richtung
        If y(k - 1) > u(k) Then Sign = -1 'Negative Richtung
        If y(k - 1) = u(k) Then Sign = 0 'Stagnieren
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.None
                Output += Sign * T * Slope 'Ausgangswert verändern
                'Springe auf den Soll-Wert, anstatt ihn zu überspringen
                If Sign * Output > Sign * u(k) Then Output = u(k)
                Return Output
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class
```

Verweise

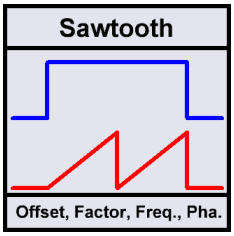
Namensbereich „Regelglieder“ → Seite 45

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

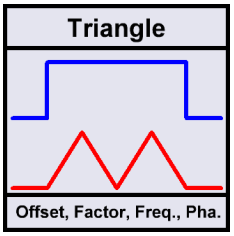
3.2 Signalgeneratoren (Generator)

Unter dem Namensbereich „Generator“ stellt die Automatisierungsbibliothek Signalgeneratoren zur Verfügung, mit denen einfache Sägezahn-, Dreieck-, PWM- und Sinus-Signale erzeugt werden können. Diese Signale können sogar moduliert werden, indem an den Parametereingängen weitere Signalgeneratoren angeschlossen werden, um zum Beispiel ein Zirp-Signal zu generieren. Zudem lässt sich das Ausgangssignal eines Generators auch durch weitere Funktionsblöcke der Bibliothek leiten, um zum Beispiel mit dem Funktionssegment „Random“ ein weißes Rauschen zu erzeugen, das mit einem zusätzlichen Tiefpass beispielsweise zu rosa Rauschen gefiltert werden kann.

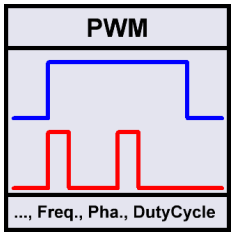
Auch einem Signalgenerator muss beim Aufrufen der Ausgangsfunktion ein Eingangswert übergeben werden. Stellt dieser Wert ein logisches „True“ dar, startet der Generator mit der Ausgabe seiner Funktion. Mit dem Wert „NaN“ am Eingang wird der Generator vorübergehend angehalten und mit einem logischen „False“ bzw. dem Wert „0“ am Eingang zurückgesetzt.



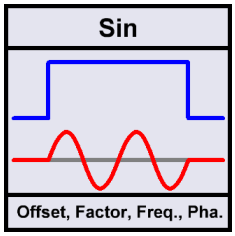
→ Seite 77



→ Seite 79



→ Seite 81



→ Seite 83

Verweise

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

Aufbau der Bibliothek → Seite 29

Implementierung eines Funktionsblocks → Seite 36

Hinweise zur Implementierung → Seite 38

Approximationstypen der Signalgeneratoren

Die zeitliche Position in der mathematischen Ausgangsfunktion wird durch Aufaddieren der Zykluszeiten bzw. der Werte „DeltaT“ gesteuert, die beim Aufruf der Ausgangsfunktion „Output“ übergeben werden. Je nachdem, ob der Approximationstyp „ReturnToZero“ oder „Continuously“ eingestellt ist, setzt der Generator diese zeitliche Position bei Erreichen oder Überschreiten einer Periodenlänge der mathematischen Ausgangsfunktion auf den Wert „0“ zurück oder zieht eine Periodenlänge davon ab. Somit kann je nach Anwendungsfall entweder nach jeder Periode der Ausgangswert $f(0)$ (bei Sägezahn ist dies zum Beispiel der Wert „0“) erzwungen oder die eingestellte Frequenz exakt eingehalten werden. Ein Überlauf der Variablen, die die zeitliche Position in der mathematischen Ausgangsfunktion beschreibt, wird in beiden Fällen vermieden. Alle Signalgeneratoren sind auch für negative Zykluszeiten ausgelegt und können durch die Übergabe negativer „DeltaT“-Werte auch rückwärts laufen.

Der Unterschied der beiden Approximationstypen wird mit folgendem Beispiel klar ersichtlich.

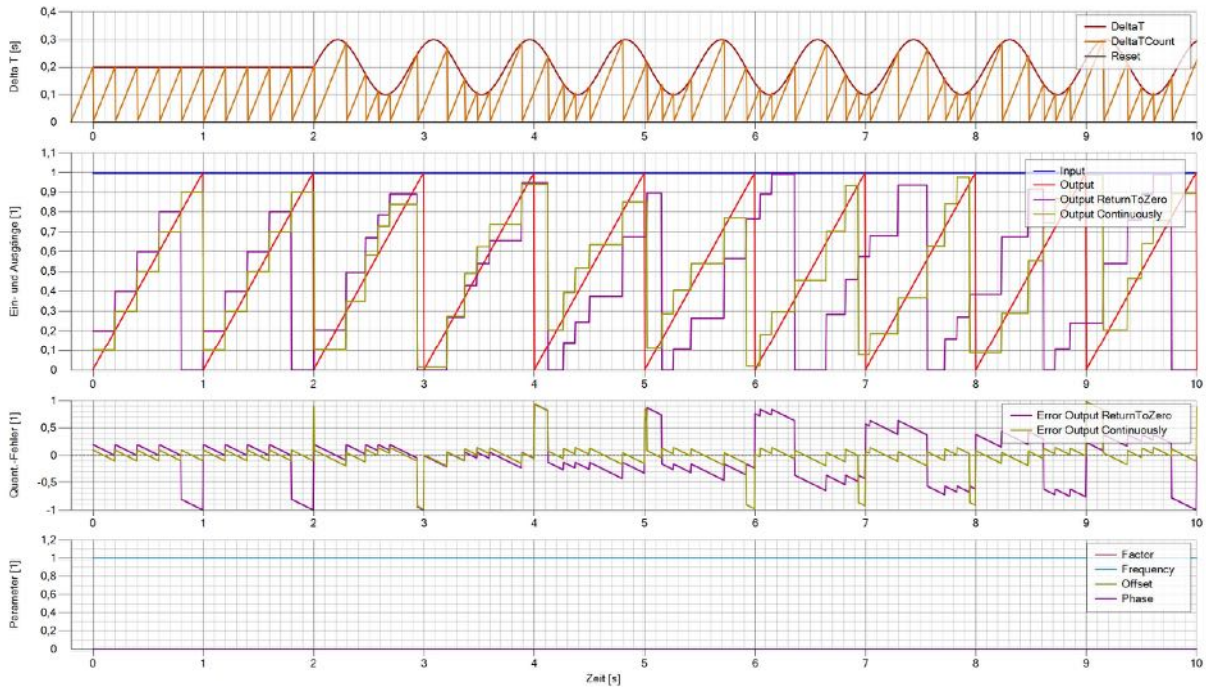


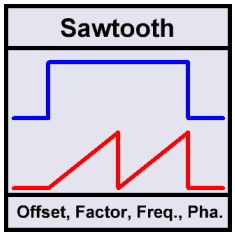
Abb. 3-25: Funktionsblock-Testergebnis: Signalgenerator (Approximationstypen)

0 - 2 Zeitäquidistante Zykluszeiten

2 - 10 Nicht zeitäquidistante Zykluszeiten

(Der Funktionsbaustein mit dem Approximations-Typ „ReturnToZero“ wird nach jeder Periode, unabhängig davon, wie weit die Periodenlänge über- oder unterschritten wurde, auf den Wert 0 zurückgesetzt. Deshalb entstehen zeitliche Fehler, wodurch die ausgegebene Frequenz nicht exakt der eingestellten entspricht und sich das Ausgangssignal zum idealen Signal mit fortschreitender Simulationszeit langsam verschiebt.

3.2.1 Sägezahn-Generator (Generator.Sawtooth)



Der Sägezahn-Generator gibt ein linear ansteigendes Signal aus, das periodisch zurückgesetzt wird. Alle folgenden Generator-Funktionsblöcke erzeugen ihre Ausgangssignale, indem sie diesen Sägezahn-Generator implementieren und dessen Ausgangssignal durch eine mathematische Funktion leiten, die eine Periode ihrer eigenen Signalform beschreibt.

Eigenschaften

Ausgangsfunktion:	$y(t)$	$= Offset + Factor \cdot f(\text{Mod2}(Frequency \cdot t + Phase))$
Signalform:	$f(x)$	$= x \qquad x \in [0; 1[$
Parameter:	$Offset$	= Offset (Verschiebung in Y-Richtung)
	$Factor$	= Verstärkung (Skalierung in Y-Richtung)
	$Phase$	= Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
	$Frequency$	= Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

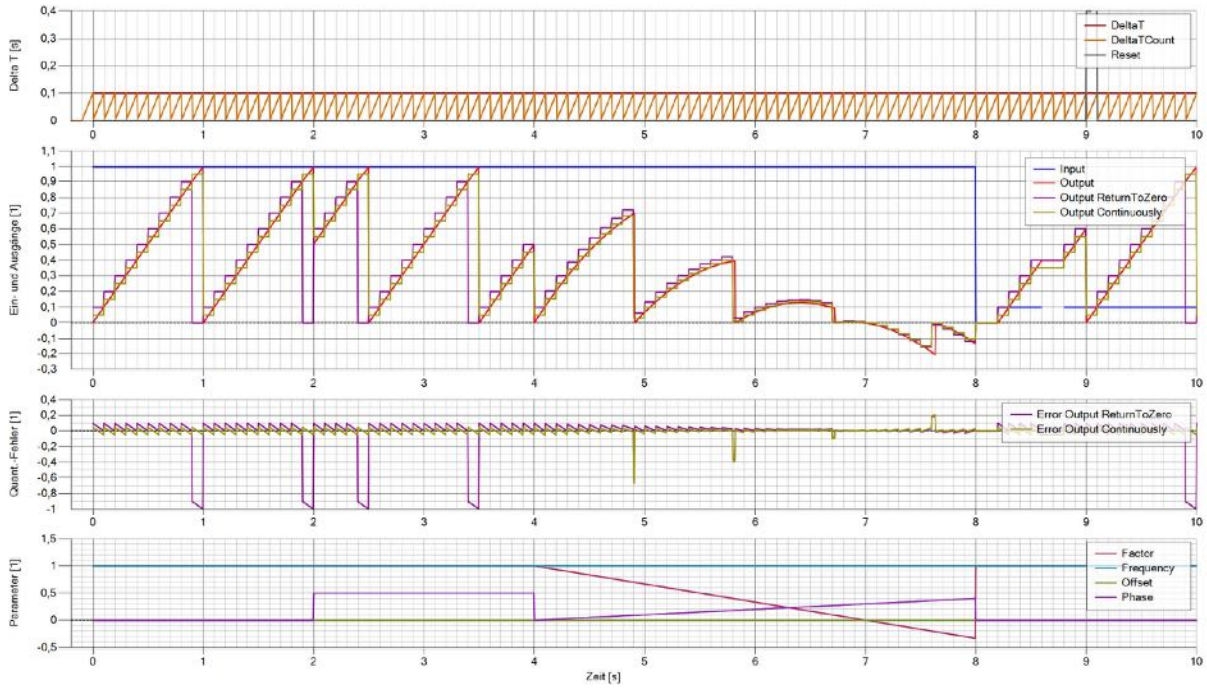


Abb. 3-26: Funktionsblock-Testergebnis: Sägezahn-Generator

0 - 2 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$

2 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0{,}5$

4 - 8 Modulieren des Ausgangssignals durch Verändern der Parameter

8 - 9 Reaktion auf $Input = NaN$ (Signalgenerator wird eingefroren)

9 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Programmcode

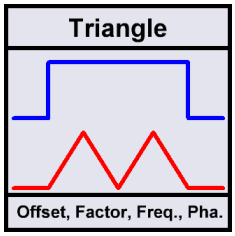
```
'#### Sägezahn-Generator ####
Public Class Sawtooth
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously
    Public Enum ApproxTypeEnum
        ReturnToZero 'X-Position wird nach jeder Periode zurückgesetzt
        Continuously 'X-Position wird nach jeder Periode um eine Periodenlänge zurückgesetzt
    End Enum

    Property Offset As Double = 0 'Offset (Verschiebung in Y-Richtung)
    Property Factor As Double = 1 'Verstärkung (Skalierung in Y-Richtung)
    Property Phase As Double = 0 'Phase (Verschiebung in X-Richtung)
    Property Frequenz As Double = 1 'Frequenz (Skalierung in X-Richtung)

    Private PosInFunc As Double 'X-Position in "RawFunc"-Funktion [0; 1[

    Protected Delegate Function RawFunc_Delagate(ByVal Input As Double) As Double
    Protected RawFunc As RawFunc_Delagate = Function(ByVal Input As Double) As Double 'Periode [0; 1[
                                                Return Input 'Eine Periode der Sägezahn-Funktion
                                            End Function

    Protected Overrides Function Functionality() As Double
        If Double.IsNaN(u(k - 0)) Then Return y(k - 1) 'Bei NaN am Eingang Generator einfrieren
        If Not (u(k - 0) > 0 Or u(k - 0) < 0) Then 'Bei False am Eingang
            PosInFunc = 0 'X-Position zurücksetzen
            Return Offset 'Offset zurückgeben
        End If

        'Bei Frequenzänderung bleibt die Funktion stetig, bei Phasenänderung nicht.
        PosInFunc += T * Frequenz 'X-Position abhängig von der Frequenz erhöhen
        Dim PosInFuncTmp As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.ReturnToZero 'Approximations-Typ (siehe oben)
                If PosInFunc >= 1 Then PosInFunc = 0
                PosInFuncTmp = PosInFunc
            Case ApproxTypeEnum.Continuously 'Approximations-Typ (siehe oben)
                PosInFunc = PosInFunc Mod 1
                PosInFuncTmp = PosInFunc
                PosInFuncTmp -= T * Frequenz / 2 'Korrektur um halbe Zykluszeit bei "Continuously"
            Case Else
                Return Double.NaN
        End Select
        PosInFuncTmp += Phase 'Phasenverschiebung
        PosInFuncTmp = PosInFuncTmp Mod 1 'Wert in den Bereich ]-1; 1[ bringen
        If PosInFuncTmp < 0 Then PosInFuncTmp += 1 'Wert in den Definitionsbereich [0; 1[ bringen
        PosInFuncTmp = Offset + Factor * RawFunc(PosInFuncTmp) 'Offset und Skalierung
        Return PosInFuncTmp
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        PosInFunc = 0 'X-Position zurücksetzen
    End Sub

End Class
```

Verweise

Namensbereich „Signalgeneratoren“ → Seite 75

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.2.2 Dreieck-Generator (Generator.Triangle)



Der Dreieck-Generator gibt ein Signal aus, das abwechselnd linear bis zu einem Maximalwert ansteigt und anschließend wieder linear auf einen Minimalwert abfällt. Der Generator erzeugt das Signal, indem er einen Sägezahn-Generator (Seite 77) implementiert und dessen Ausgangssignal durch das Funktionssegment „Triangle“ (Seite 153) leitet.

Eigenschaften

Ausgangsfunktion: $y(t) = Offset + Factor \cdot f(\mathrm{Mod2}(Frequency \cdot t + Phase))$

Signalform: $f(x) = \begin{cases} 2 \cdot x & \text{für } x < \frac{1}{2} \\ 2 - 2 \cdot x & \text{für } x \geq \frac{1}{2} \end{cases} \quad x \in [0;1[$

Parameter:

- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

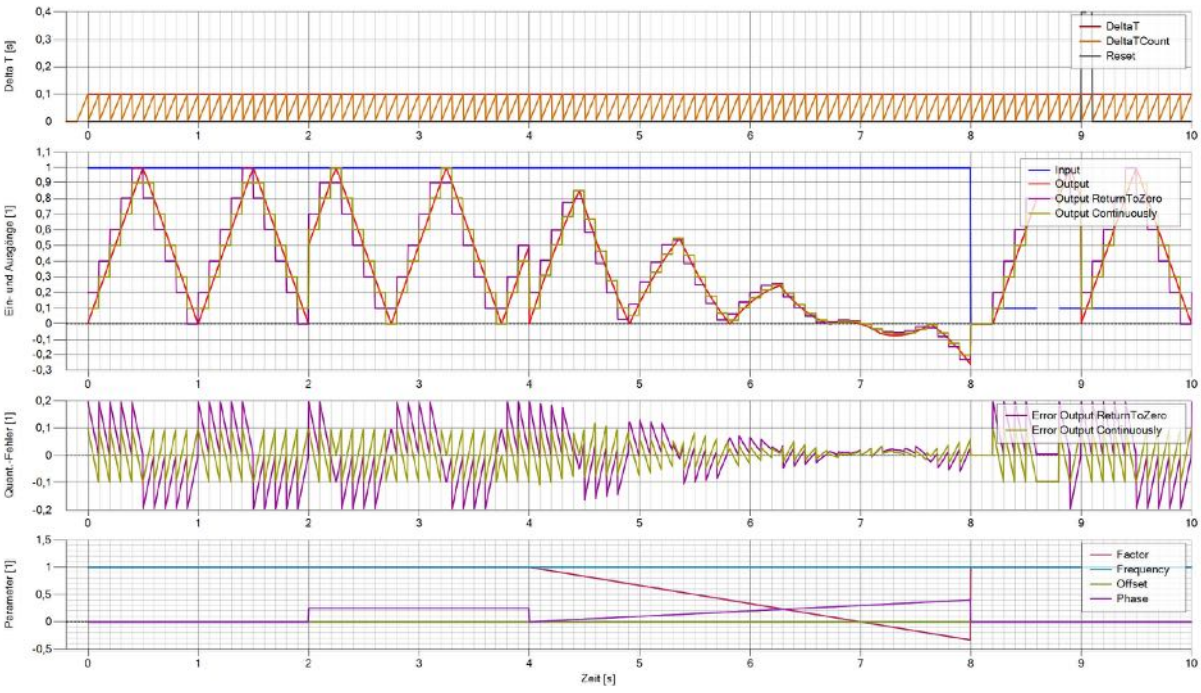


Abb. 3-27: Funktionsblock-Testergebnis: Dreieck-Generator

0 - 2 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$

2 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0{,}25$

4 - 8 Modulieren des Ausgangssignals durch Verändern der Parameter

8 - 9 Reaktion auf $Input = NaN$ (Signalgenerator wird eingefroren)

9 - 10 Zurücksetzen durch Aufrufen der Funktion „*Reset*“

Programmcode

```
'#### Dreieck-Generator ####
Public Class Triangle
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Private FuncSegmentTriangle As New FuncSegment.Triangle 'Zugrundeliegendes Funktionssegment

    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentTriangle.Offset = 1 / 2
        FuncSegmentTriangle.Factor = 1 / 2
        FuncSegmentTriangle.Frequency = 1
        FuncSegmentTriangle.Phase = -1 / 4
        FuncSegmentTriangle.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentTriangle.Output(Input)
                  End Function
    End Sub

End Class
```

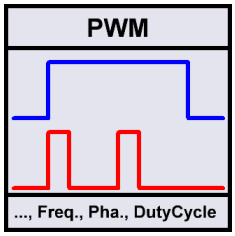
Verweise

Namensbereich „Signalgeneratoren“ → Seite 75

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

Funktionssegment „Dreieck-Funktion“ → Seite 153

3.2.3 PWM-Generator (Generator.PWM)



Der PWM-Generator gibt ein pulsweitenmoduliertes Rechtecksignal aus, dessen Tastgrad mit dem Parameter „DutyCycle“ eingestellt werden kann. Der Generator erzeugt das Signal, indem er einen Sägezahn-Generator (Seite 77) implementiert und dessen Ausgangssignal durch das Funktionssegment „PWM“ (Seite 149) leitet.

Eigenschaften

Ausgangsfunktion: $y(t) = Offset + Factor \cdot f(\mathrm{Mod2}(Frequency \cdot t + Phase))$

Signalform: $f(x) = \begin{cases} 1 & \text{für } x < DutyCycle \\ 0 & \text{für } x \geq DutyCycle \end{cases} \quad x \in [0;1[$

Parameter:

- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)
- $DutyCycle$ = Tastgrad (Verhältnis Impulsdauer zu Periodendauer)

Funktionsblock-Testergebnis

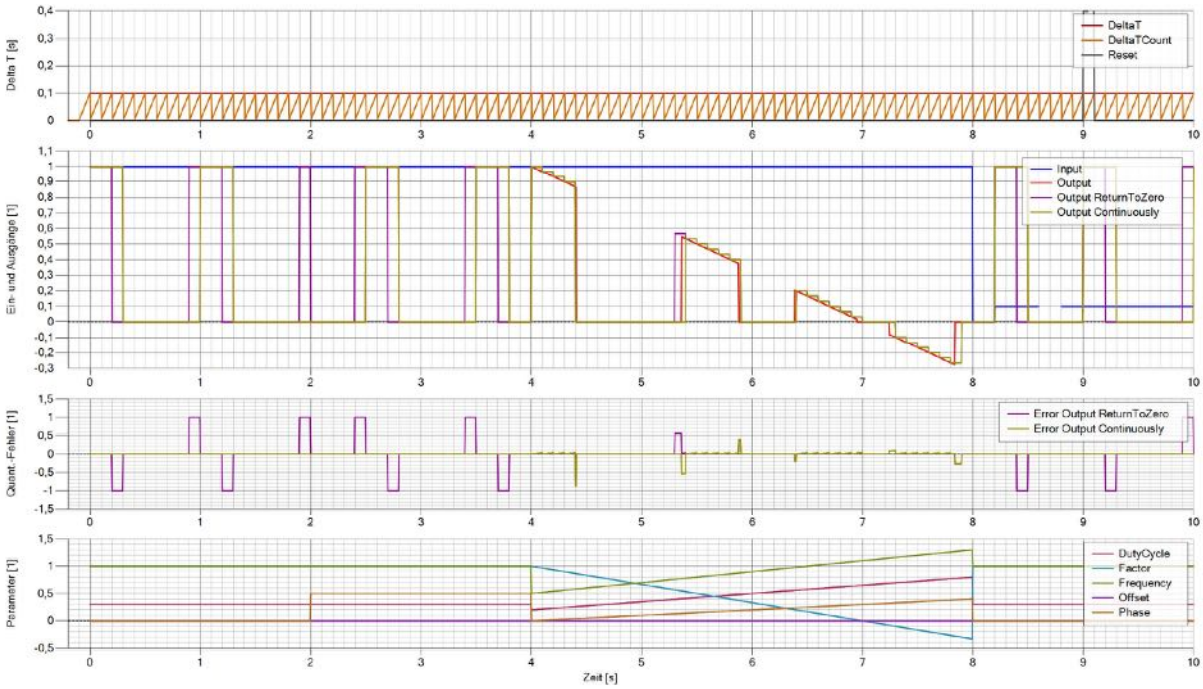


Abb. 3-28: Funktionsblock-Testergebnis: PWM-Generator

0 - 2 Verhalten bei $Factor = 1$ $Frequency = 1$ $Phase = 0$ $DutyCycle = 0{,}3$

2 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Phase = 0{,}5$ $DutyCycle = 0{,}3$

4 - 8 Modulieren des Ausgangssignals durch Verändern der Parameter

8 - 9 Reaktion auf $Input = NaN$ (Signalgenerator wird eingefroren)

9 - 10 Zurücksetzen durch Aufrufen der Funktion „*Reset*“

Programmcode

```
'#### PWM-Generator ####
Public Class PWM
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Property DutyCycle As Double = 0.2 'Tastgrad (Verhältnis Impulsdauer zu Periodendauer)

    Private FuncSegmentPWM As New FuncSegment.PWM 'Zugrundeliegendes Funktionssegment

    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentPWM.Offset = 0
        FuncSegmentPWM.Factor = 1
        FuncSegmentPWM.Frequency = 1
        FuncSegmentPWM.Phase = 0
        FuncSegmentPWM.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentPWM.Output(Input)
                  End Function
    End Sub

    Protected Overrides Function Functionality() As Double
        FuncSegmentPWM.DutyCycle = DutyCycle 'Tastgrad bei Aufruf der Ausgangsfunktion aktualisieren
        Return MyBase.Functionality
    End Function

End Class
```

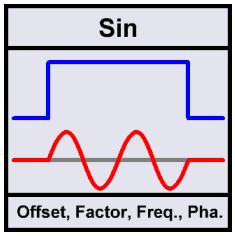
Verweise

Namensbereich „Signalgeneratoren“ → Seite 75

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

Funktionssegment „PWM-Funktion“ → Seite 149

3.2.4 Sinus-Generator (Generator.Sin)



Der Sinus-Generator gibt ein Signal aus, dessen zeitlicher Verlauf der Sinusfunktion folgt. Der Generator erzeugt das Signal, indem er einen Sägezahn-Generator (Seite 77) implementiert und dessen Ausgangssignal durch das Funktionssegment „Sin“ (Seite 155) leitet.

Eigenschaften

Ausgangsfunktion:	$y(t)$	$= Offset + Factor \cdot f(\text{Mod2}(Frequency \cdot t + Phase))$
Signalform:	$f(x)$	$= \sin(x) \quad x \in [0; 1[$
Parameter:	$Offset$	= Offset (Verschiebung in Y-Richtung)
	$Factor$	= Verstärkung (Skalierung in Y-Richtung)
	$Phase$	= Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
	$Frequency$	= Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

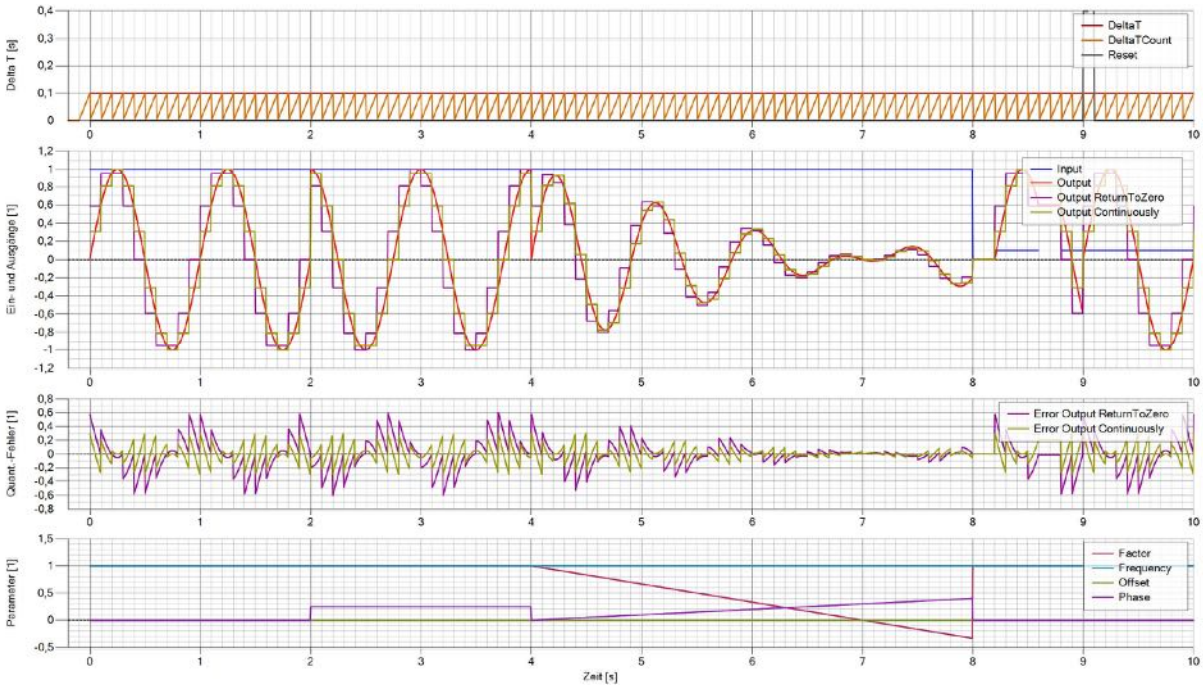


Abb. 3-29: Funktionsblock-Testergebnis: Sinus-Generator

0 - 2 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$

2 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0{,}25$

4 - 8 Modulieren des Ausgangssignals durch Verändern der Parameter

8 - 9 Reaktion auf $Input = NaN$ (Signalgenerator wird eingefroren)

9 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Programmcode

```
'#### Sinus-Generator ####
Public Class Sin
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Private FuncSegmentSin As New FuncSegment.Sin 'Zugrundeliegendes Funktionssegment

    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentSin.Offset = 0
        FuncSegmentSin.Factor = 1
        FuncSegmentSin.Frequency = 1
        FuncSegmentSin.Phase = 0
        FuncSegmentSin.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentSin.Output(Input)
                  End Function
    End Sub

End Class
```

Verweise

Namensbereich „Signalgeneratoren“ → Seite 75

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

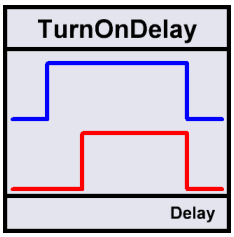
Funktionssegment „Sinus-Funktion“ → Seite 155

3.3 Zeitglieder (Timer)

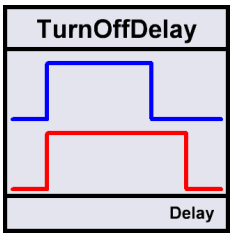
Mit den Zeitgliedern des Namensbereichs „Timer“ können Impulse zeitlich verkürzt, verlängert oder auf eine eingestellte Länge gebracht werden. Der Anwendungsbereich für Zeitglieder ist groß. Einige Beispielanwendungen sind die Treppenlicht-Schaltung, eine nachlaufende Lüfter-Steuerung und das Entprellen von Taster-Signalen.

Bei den ersten drei Funktionsblöcken des Namensbereichs können die Approximations-Typen „TooLate“, „TooEarly“ und „Punctually“ eingestellt werden. Je nach Approximations-Typ reagiert das jeweilige Zeitglied im ungünstigsten Fall einen Zyklus zu früh, einen Zyklus zu spät oder mit dem dritten Approximations-Typ in dem Zyklus, der unter Betracht der aktuellen Zykluszeit möglichst nahe am idealen Zeitpunkt liegt.

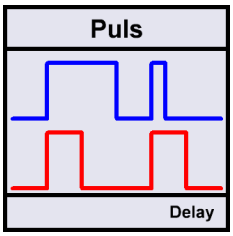
Der vierte Funktionsblock „StopWatch“ misst die zeitliche Länge eines Impulses. Er stellt hierzu die drei Approximations-Typen „ResetToZero“, „ResetToDeltaT“ und „ResetToHalfDeltaT“ zur Verfügung. Je nach gewähltem Typ wird die Stoppuhr zum Messen der Impulslänge bei einem Reset auf den Wert „0“, auf die aktuelle Zykluszeit oder auf deren Hälfte zurückgesetzt. Die daraus resultierenden Unterschiede sind in den Schaubildern der einzelnen Dokumentationen klar zu erkennen.



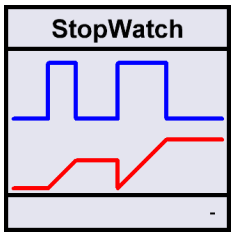
→ Seite 87



→ Seite 89



→ Seite 91

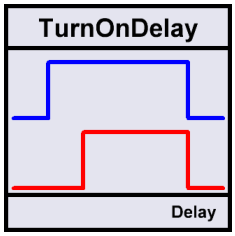


→ Seite 93

Verweise

Basisklasse „Zeitabhängiger Funktionsblock“	→ Seite 33
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38

3.3.1 Anzugsverzögerung (Timer.TurnOnDelay)



Der Funktionsbaustein „TurnOnDelay“ verkürzt einen Impuls um die über den Parameter „Delay“ einstellbare Zeit.

Konvertierungen

Eingang (Double → Boolean):	1 und $\pm\infty$	→ *True*
	0 und *NaN*	→ *False*
Ausgang (Boolean → Double):	*True*	→ 1
	False	→ 0

Eigenschaften

Parameter: *Delay* = Verzögerungszeit

Funktionsblock-Testergebnis

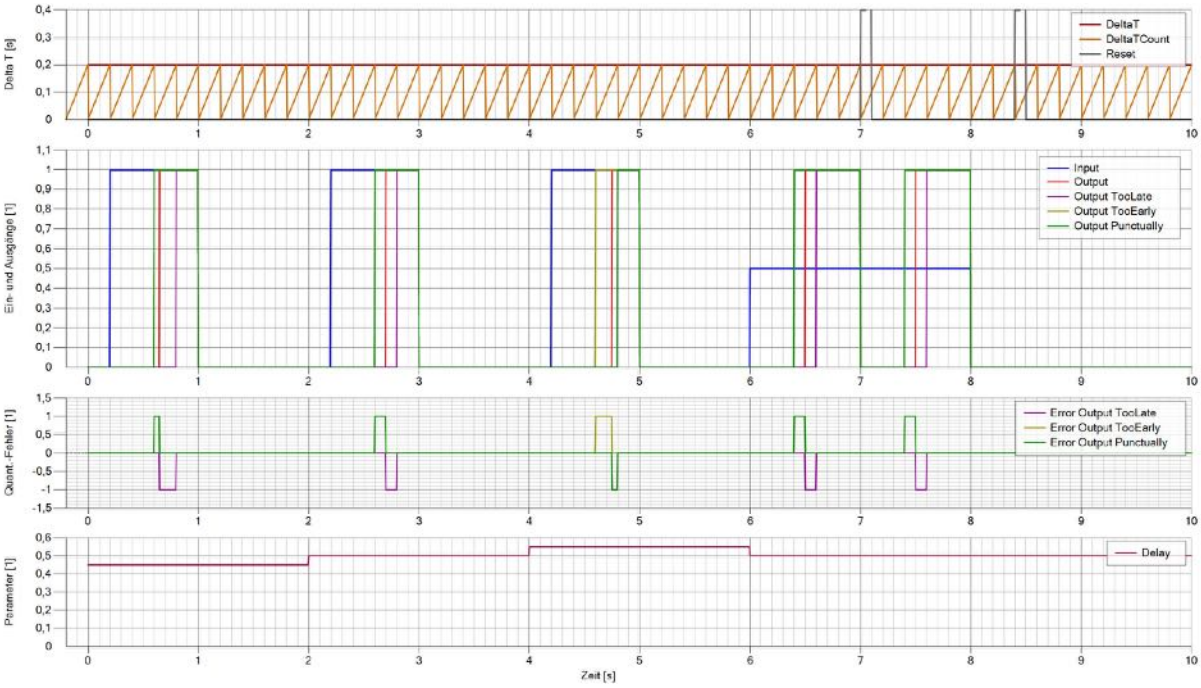


Abb. 3-30: Funktionsblock-Testergebnis: Anzugsverzögerung

0 - 2 Verhalten bei $Delay = 0{,}45$ (gelb verbirgt sich hinter grün)

2 - 4 Verhalten bei $Delay = 0{,}5$ (gelb verbirgt sich hinter grün)

4 - 6 Verhalten bei $Delay = 0{,}55$ (violett verbirgt sich hinter grün)

6 - 10 Zurücksetzen durch Aufrufen der Funktion „*Reset*“

Programmcode

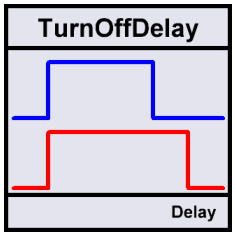
```
'#### Anzugsverzögerung ####
Public Class TurnOnDelay
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Verzögerungszeit

    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch >= Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T >= Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 >= Delay, 1, 0)
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else 'Bei Input = False
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 0
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class
```

Verweise

Namensbereich „Zeitglieder“ → Seite 85

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.3.2 Abfallverzögerung (Timer.TurnOffDelay)



Der Funktionsbaustein „TurnOffDelay“ verlängert einen Impuls um die über den Parameter „Delay“ einstellbare Zeit.

Konvertierungen

Eingang (Double → Boolean):	1 und $\pm\infty$	$\rightarrow True$
	0 und NaN	$\rightarrow False$
Ausgang (Boolean → Double):	$True$	$\rightarrow 1$
	$False$	$\rightarrow 0$

Eigenschaften

Parameter: $Delay$ = Verzögerungszeit

Funktionsblock-Testergebnis

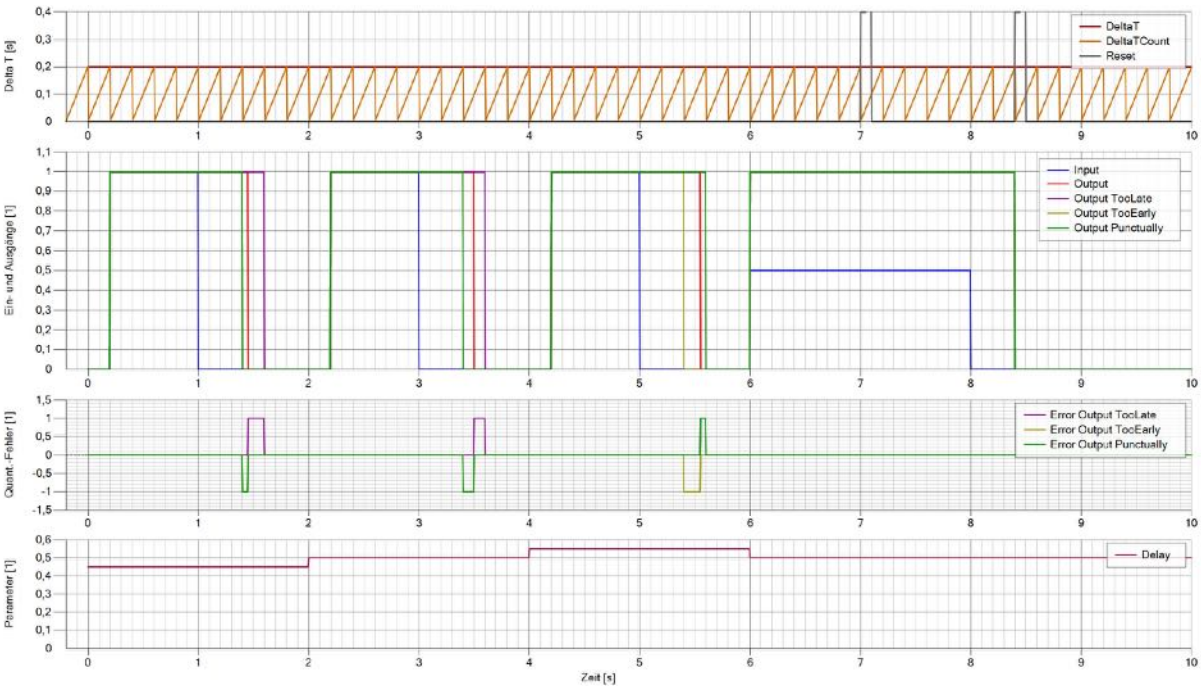


Abb. 3-31: Funktionsblock-Testergebnis: Abfallverzögerung

0 - 2 Verhalten bei $Delay = 0{,}45$ (gelb verbirgt sich hinter grün)

2 - 4 Verhalten bei $Delay = 0{,}5$ (gelb verbirgt sich hinter grün)

4 - 6 Verhalten bei $Delay = 0{,}55$ (violett verbirgt sich hinter grün)

6 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Programmcode

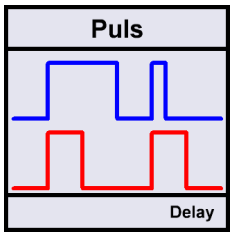
```
'#### Abfallverzögerung ####
Public Class TurnOffDelay
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Verzögerungszeit

    Private StopWatch As Double = Double.PositiveInfinity 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If Not (u(k - 0) > 0 Or u(k - 0) < 0) Then 'Bei Input = False
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch < Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T < Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 < Delay, 1, 0)
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else 'Bei Input = True
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 1
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = Double.PositiveInfinity 'Zeitzähler zurücksetzen
    End Sub

End Class
```

Verweise

Namensbereich „Zeitglieder“ → Seite 85

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.3.3 Impuls (Timer.Puls)



Der Funktionsbaustein „Impuls“ passt einen Impuls so an, dass die Länge der über den Parameter „Delay“ einstellbaren Zeit entspricht.

Konvertierungen

Eingang (Double → Boolean):	1 und $\pm\infty$	→ *True*
	0 und *NaN*	→ *False*
Ausgang (Boolean → Double):	*True*	→ 1
	False	→ 0

Eigenschaften

Parameter: *Delay* = Impulslänge

Funktionsblock-Testergebnis

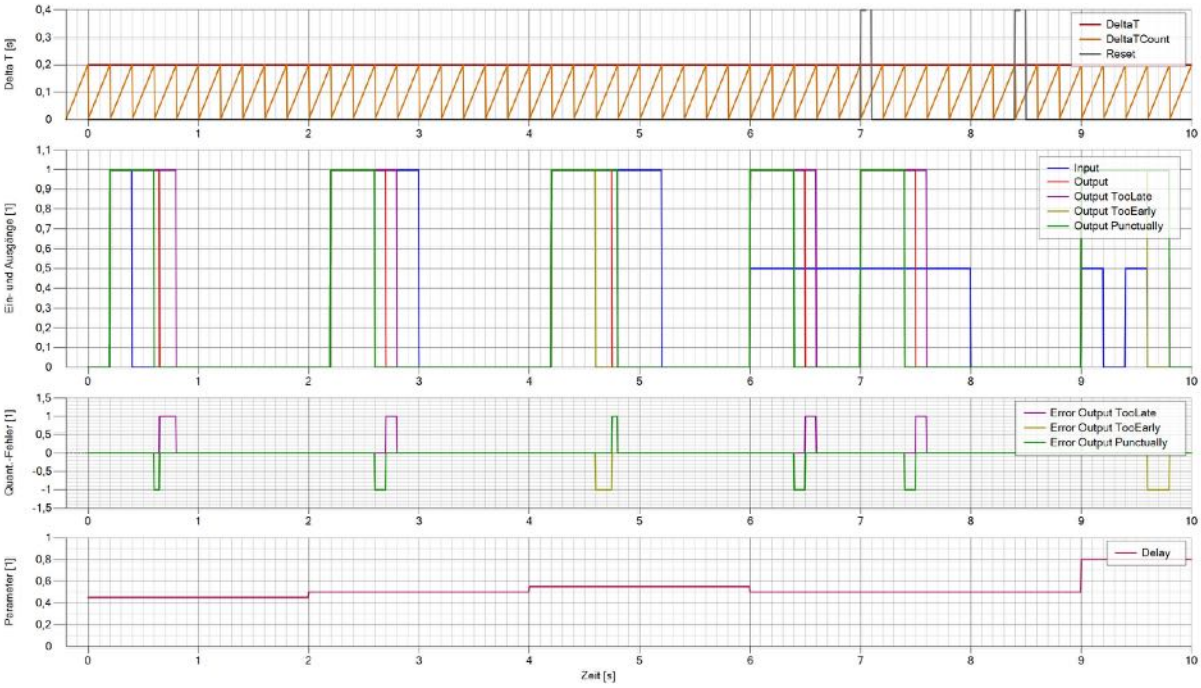


Abb. 3-32: Funktionsblock-Testergebnis: Abfallverzögerung

0 - 2 Verhalten bei *Delay* = 0,45 (gelb verbirgt sich hinter grün)

2 - 4 Verhalten bei *Delay* = 0,5 (gelb verbirgt sich hinter grün)

4 - 6 Verhalten bei *Delay* = 0,55 (violett verbirgt sich hinter grün)

6 - 9 Zurücksetzen durch Aufrufen der Funktion *„Reset“*

9 - 10 Verhalten bei *Delay* = 0,8 (violett verbirgt sich hinter grün)
(Der zweite Impuls wird ignoriert, weil der erste noch verlängert wird.
→ nicht nachtriggerbar)

Programmcode

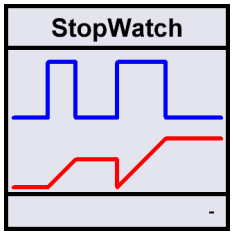
```
'#### Impuls ####
Public Class Puls
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Impulslänge

    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Or y(k - 1) = 1 Then
            'Bei Input = True oder wenn der Impuls noch nicht vollständig ausgegeben wurde
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch < Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T < Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 < Delay, 1, 0)
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else
            'Bei Input = False, aber nur wenn der Impuls schon vollständig ausgegeben wurde
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 0
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class
```

Verweise

Namensbereich „Zeitglieder“ → Seite 85

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.3.4 Stoppuhr (Timer.StopWatch)



Der Funktionsbaustein „StopWatch“ misst die zeitliche Länge eines Eingangsimpulses und gibt diese als Wert aus. Mit einem logischen „False“ am Eingang wird der Zeitzähler angehalten und mit dem nächsten logischen „True“ zurückgesetzt und neu gestartet. „NaN“ friert den Zeitzähler hingegen ein, ohne ihn mit dem nächsten logischen „True“ am Eingang zurückzusetzen. Je nach gewähltem Approximations-Typ lädt der Funktionsblock den Zeitzähler beim Neustarten auf den Wert „0“, auf die übergebene Zykluszeit „DeltaT“ oder deren Hälfte vor.

Konvertierungen

Eingang (Double → Boolean):	1 und ±∞	→ *True*	
	0 und *NaN*	→ *False*	
Ausgang (Double → Boolean):	1 und ±∞	→ *True*	
	0	→ *False*	
	NaN	→ *NaN*	(zum Pausieren des Zeitzählers)

Funktionsblock-Testergebnis

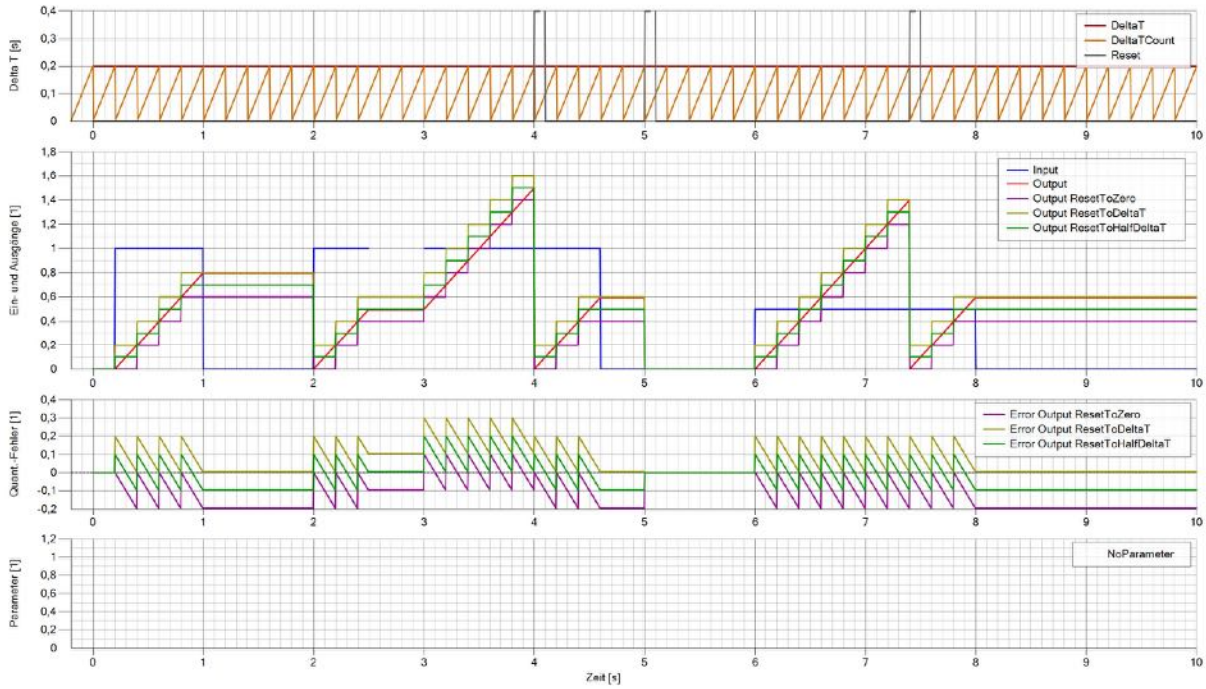


Abb. 3-33: Funktionsblock-Testergebnis: Stoppuhr

0 - 2 mit $Input = True$ starten und mit $Input = False$ anhalten

2 - 4 mit $Input = True$ starten, mit $Input = NaN$ pausieren und weiterlaufen lassen

4 - 6 Zurücksetzen durch Aufrufen der Funktion *„Reset“*

6 - 10 Mit $Input = True$ starten und durch Aufrufen der Funktion *„Reset“* zurücksetzen ($Input = 0{,}5$ wird beispielsweise auch als $Input = True$ interpretiert)

Programmcode

```
'#### Stoppuhr ####
Public Class StopWatch
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.ResetToZero
    Public Enum ApproxTypeEnum
        ResetToZero 'Auf 0 zurücksetzen
        ResetToDeltaT 'Auf Zykluszeit vorladen
        ResetToHalfDeltaT 'Auf halbe Zykluszeit vorladen
    End Enum

    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            StopWatch += T 'Zeit hochzählen
            If u(k - 1) = 0 Then 'Bei Flanke auf True
                StopWatch = 0 'Zeitzähler zurücksetzen
                Select Case DirectCast(ApproxType, ApproxTypeEnum)
                    Case ApproxTypeEnum.ResetToZero
                        StopWatch += 0 'Zeitzähler zurücksetzen
                    Case ApproxTypeEnum.ResetToDeltaT
                        StopWatch += T 'Zeitzähler auf Zykluszeit vorladen
                    Case ApproxTypeEnum.ResetToHalfDeltaT
                        StopWatch += T / 2 'Zeitzähler auf halbe Zykluszeit vorladen
                    Case Else
                        StopWatch += Double.NaN
                End Select
            End If
        End If
        Return StopWatch
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class
```

Verweise

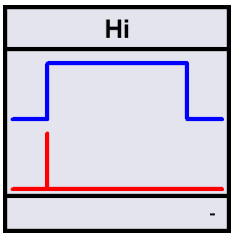
Namensbereich „Zeitglieder“ → Seite 85

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

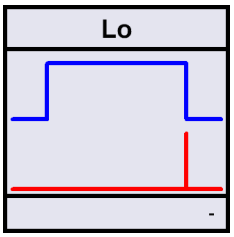
3.4 Flankenerkennungen (Flank)

Die Funktionsblöcke des Namensbereichs „Flank“ können positive und negative Flanken in Signalen erkennen. Somit ist es zum Beispiel mit den ersten drei Flankenerkennungen „Hi“, „Lo“ und „Any“ möglich, bei zyklischer Abarbeitung des Programmcodes bestimmte Ereignisse nur bei einer Flanke auszuführen. Der vierte Funktionsblock „Relay“ fungiert hingegen als Toggle-Flipflop und ermöglicht das Ein- und Wiederausschalten eines Zustandes durch zwei aufeinanderfolgende Impulse.

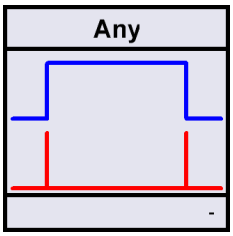
Wie bereits auf Seite 36 in den Hinweisen zur Implementierung eines Funktionsblock erklärt, ist zu beachten, dass auch Wertänderungen des Eingangssignals in positiver Richtung als positive Flanke und Wertänderungen in negativer Richtung als negative Flanke interpretiert werden. Dies gilt auch dann, wenn die Gleitkommazahl den Wert „+∞“ oder „-∞“ einnimmt oder verlässt. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird hingegen nicht als Flanke erkannt.



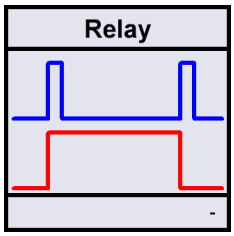
→ Seite 97



→ Seite 99



→ Seite 101

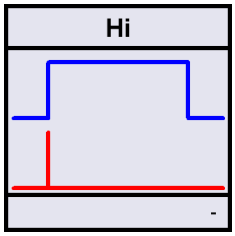


→ Seite 103

Verweise

Basisklasse „Rückblickender Funktionsblock“	→ Seite 32
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38

3.4.1 Positive Flankenerkennung (Flank.Hi)



Der Funktionsbaustein „Hi“ gibt bei einer positiven Flanke am Eingang für einen Zyklus lang den Wert „1“ und somit ein logisches „True“ aus. Eine positive Flanke wird dann erkannt, wenn sich der Eingangswert im Vergleich zum letzten Zyklus vergrößert hat. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird nicht als Flanke erkannt.

Ausgangsverhalten

Flankenerkennung:	$u(k) > u(k-1)$	$\rightarrow True$
	$u(k) \overset{\text{Equals}}{=} NaN$ oder $u(k-1) \overset{\text{Equals}}{=} NaN$	$\rightarrow False$
	$Sonst$	$\rightarrow False$

Funktionsblock-Testergebnis

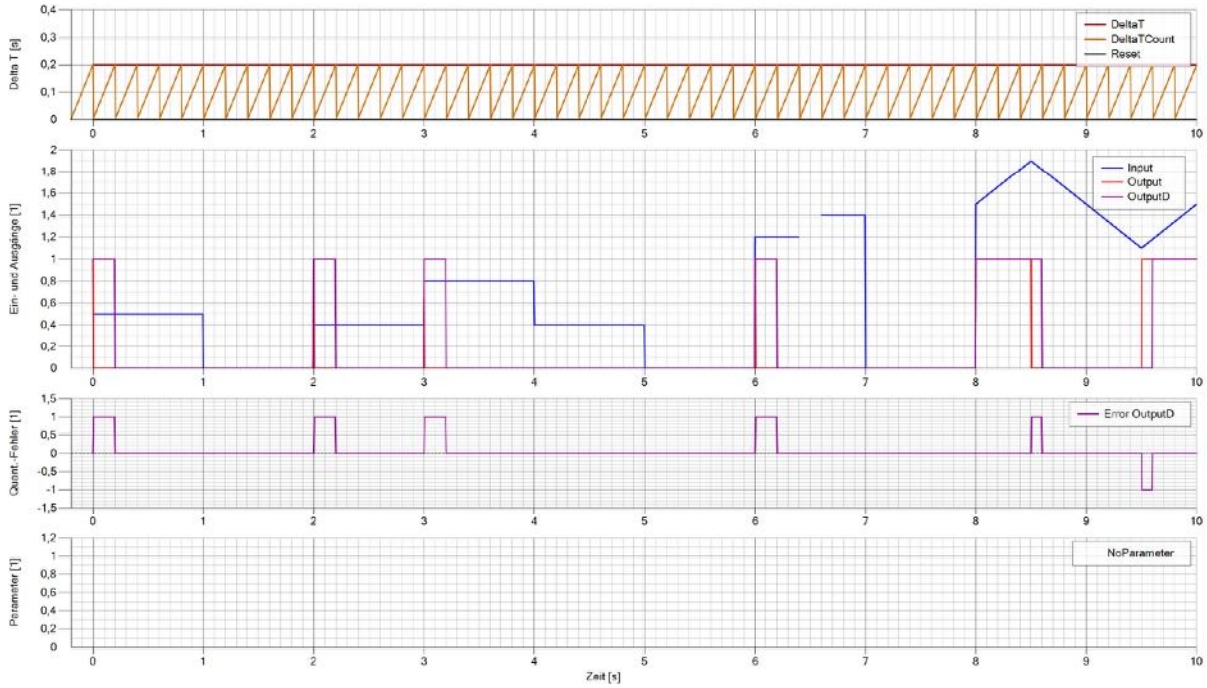


Abb. 3-34: Funktionsblock-Testergebnis: Positive Flankenerkennung

0 - 2 Flankenerkennung an einem Impuls am Eingang

2 - 6 Flankenerkennung bei Wertesprüngen am Eingang

6 - 8 Flankenerkennung bei $Input = NaN$ (wird nicht als Flanke erkannt)

8 - 10 Flankenerkennung bei Werteänderungen am Eingang

Programmcode

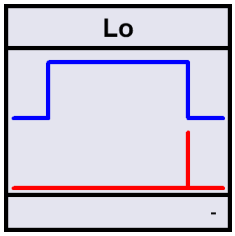
```
'#### Positive Flankenerkennung ####
Public Class Hi
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1), 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Flankenerkennungen“ → Seite 95

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.4.2 Negative Flankenerkennung (Flank.Lo)



Der Funktionsbaustein „Lo“ gibt bei einer negativen Flanke am Eingang für einen Zyklus lang den Wert „1“ und somit ein logisches „True“ aus. Eine negative Flanke wird dann erkannt, wenn sich der Eingangswert im Vergleich zum letzten Zyklus verkleinert hat. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird nicht als Flanke erkannt.

Ausgangsverhalten

Flankenerkennung:	$u(k) < u(k-1)$	$\rightarrow True$
	$u(k) \overset{\text{Equals}}{=} NaN$ oder $u(k-1) \overset{\text{Equals}}{=} NaN$	$\rightarrow False$
	$Sonst$	$\rightarrow False$

Funktionsblock-Testergebnis

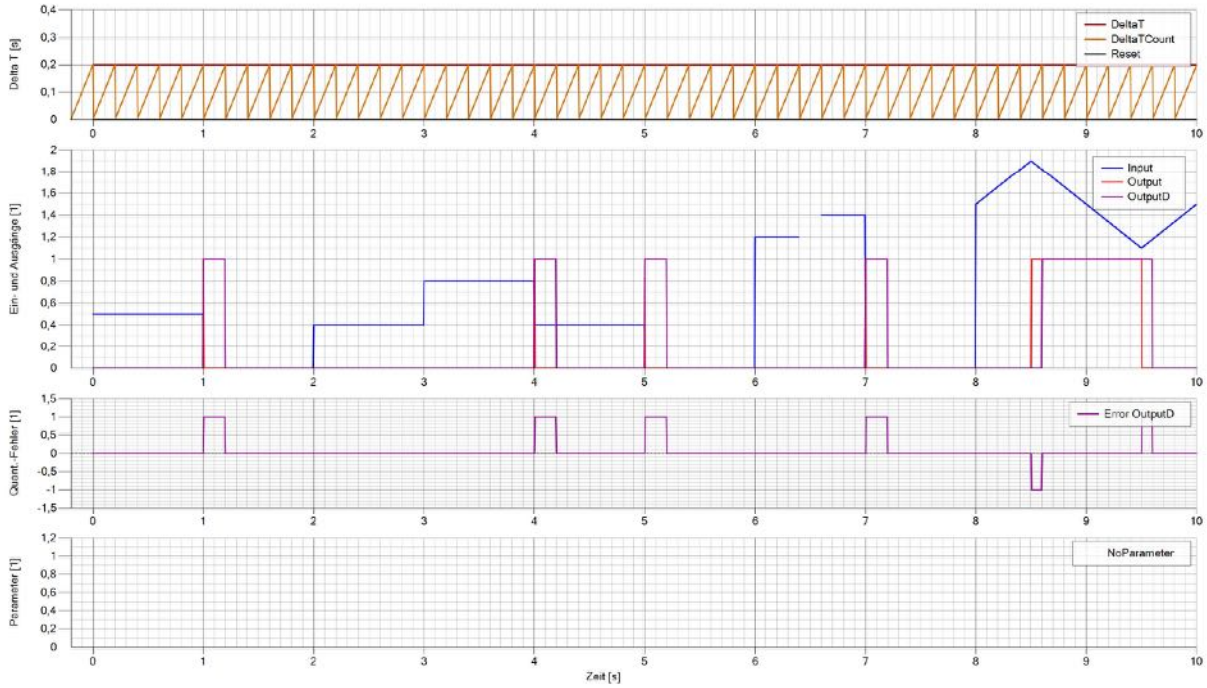


Abb. 3-35: Funktionsblock-Testergebnis: Negative Flankenerkennung

0 - 2 Flankenerkennung an einem Impuls am Eingang

2 - 6 Flankenerkennung bei Wertesprüngen am Eingang

6 - 8 Flankenerkennung bei $Input = NaN$ (wird nicht als Flanke erkannt)

8 - 10 Flankenerkennung bei Werteänderungen am Eingang

Programmcode

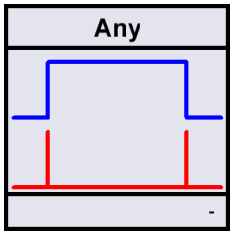
```
'#### Negative Flankenerkennung ####
Public Class Lo
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) < u(k - 1), 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Flankenerkennungen“ → Seite 95

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.4.3 Flankenerkennung (Flank.Any)



Der Funktionsbaustein „Any“ gibt bei einer Flanke am Eingang für einen Zyklus lang den Wert „1“ und somit ein logisches „True“ aus. Eine Flanke wird dann erkannt, wenn sich der Eingangswert im Vergleich zum letzten Zyklus vergrößert oder verkleinert hat. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird nicht als Flanke erkannt.

Ausgangsverhalten

Flankenerkennung:	$u(k) > u(k-1)$ oder $u(k) < u(k-1)$	$\rightarrow True$
	$u(k) \overset{\text{Equals}}{=} NaN$ oder $u(k-1) \overset{\text{Equals}}{=} NaN$	$\rightarrow False$
	$Sonst$	$\rightarrow False$

Funktionsblock-Testergebnis

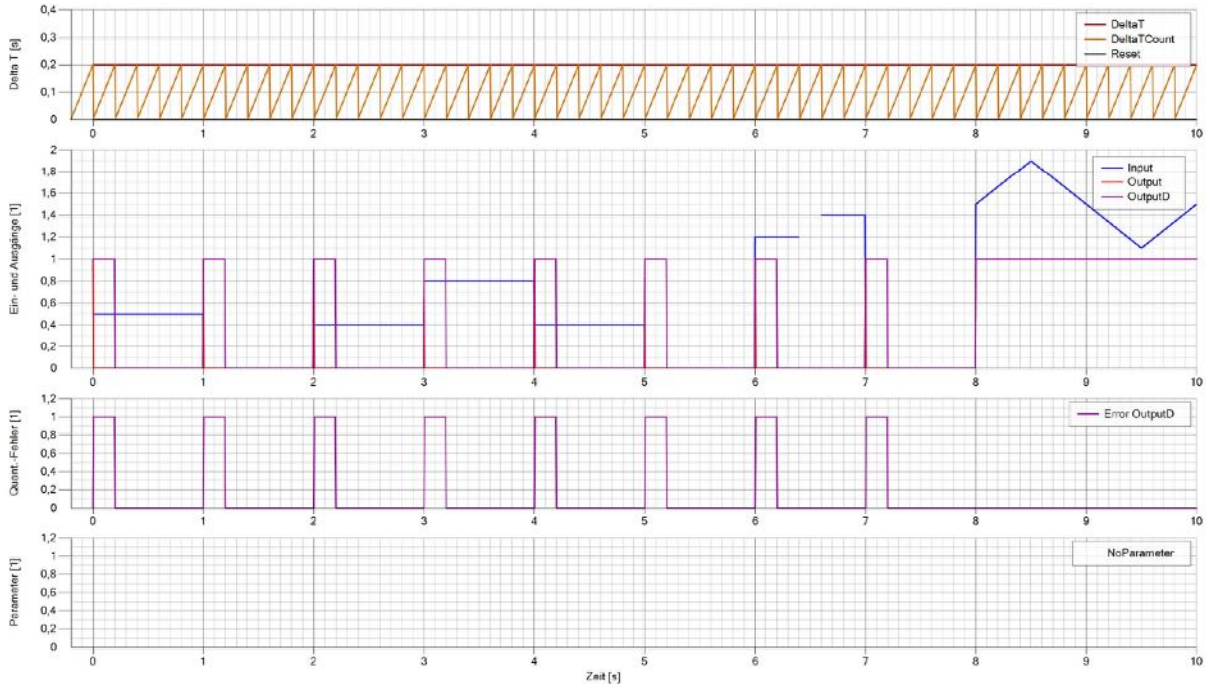


Abb. 3-36: Funktionsblock-Testergebnis: Flankenerkennung

0 - 2 Flankenerkennung an einem Impuls

2 - 6 Flankenerkennung bei Wertesprüngen am Eingang

6 - 8 Flankenerkennung bei $Input = NaN$ (wird nicht als Flanke erkannt)

8 - 10 Flankenerkennung bei Werteänderungen am Eingang

Programmcode

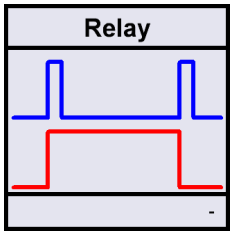
```
'#### Flankenerkennung ####
Public Class Any
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1) Or u(k - 0) < u(k - 1), 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Flankenerkennungen“ → Seite 95

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.4.4 Toggle-Flipflop (Flank.Relay)



Der Funktionsbaustein „Relay“ wechselt bei jeder positiven Flanke am Eingang seinen Ausgangswert zwischen „0“ und „1“. Eine positive Flanke wird dann erkannt, wenn sich der Eingangswert im Vergleich zum letzten Zyklus vergrößert hat. Ein Wertesprung auf „NaN“ oder von „NaN“ auf einen anderen Wert wird nicht als Flanke erkannt.

Ausgangsverhalten

Flankenerkennung:	$u(k) > u(k-1)$	→ Ausgang umschalten
	$u(k) \overset{\text{Equals}}{=} NaN$ oder $u(k-1) \overset{\text{Equals}}{=} NaN$	→ $False$
	$Sonst$	→ Ausgang halten

Funktionsblock-Testergebnis

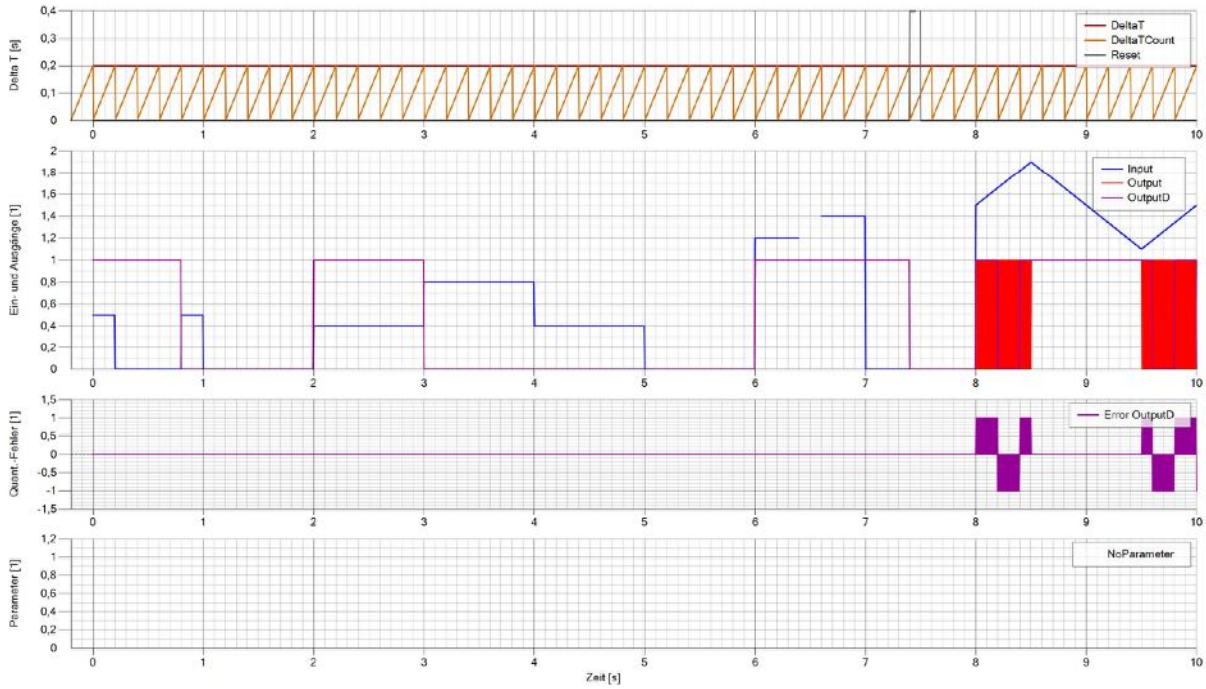


Abb. 3-37: Funktionsblock-Testergebnis: Toggle-Flipflop

0 - 2 Toggle-Flipflop an einem Impuls

2 - 6 Toggle-Flipflop bei Wertesprüngen am Eingang

6 - 7 Toggle-Flipflop bei $Input = NaN$ (wird nicht als Flanke erkannt)

7 - 8 Zurücksetzen durch Aufrufen der Funktion *„Reset“*

8 - 10 Toggle-Flipflop bei Werteänderungen am Eingang
(Die roten Flächen entstehen durch die ständige Wertänderung des Signals „Output“, weil der dazugehörige Funktionsblock in Zeitabständen der Simulations-Zykluszeit aufgerufen wird und jedes Mal das Ausgangssignal zwischen „0“ und „1“ umschaltet.)

Programmcode

```
'#### Toggle-Flipflop ####
Public Class Relay
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1), If(y(k - 1) = 0, 1, 0), y(k - 1))
    End Function

End Class
```

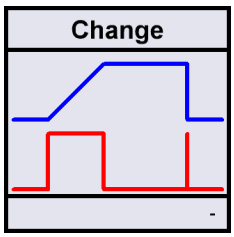
Verweise

Namensbereich „Flankenerkennungen“ → Seite 95

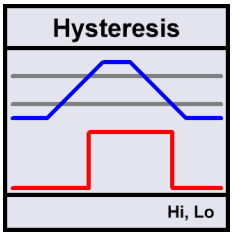
Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.5 Analoge Übertragungsglieder (Analog)

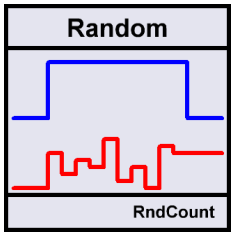
Die Funktionsblöcke des Namensbereichs „Analog“ verarbeiten analoge Signale und beziehen sich dabei auf Signalwerte der letzten Funktionsaufrufe. Somit kann der Funktionsbaustein „Change“ zum Beispiel erkennen, ob sich ein Signal seit dem letzten Funktionsaufruf geändert hat oder der Funktionsbaustein „Hysteresis“ kann ermitteln, welcher Grenzwert der Hysterese zuletzt überschritten wurde. Des Weiteren wird für den Zufallsgenerator „Random“ der letzte Ausgangswert gespeichert und als Anfangswert zum Generieren des nächsten Zufallswerts genutzt. Der letzte Funktionsblock „DeltaT“ errechnet hingegen mit einem Zeitstempel des letzten Funktionsaufrufs die verstrichene Zeitspanne und ermittelt so die Zykluszeit.



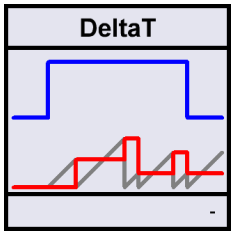
→ Seite 107



→ Seite 109



→ Seite 111

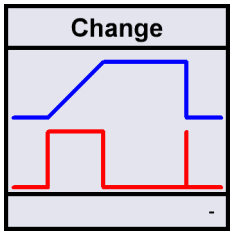


→ Seite 113

Verweise

Basisklasse „Rückblickender Funktionsblock“	→ Seite 32
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38

3.5.1 Wertänderung (Analog.Change)



Der Funktionsbaustein „Change“ gibt den Wert „1“ und somit ein logisches „True“ aus, wenn sich der Eingangswert seit dem letzten Funktionsaufruf geändert hat. Eine Wertänderung wird auch dann erkannt, wenn der Eingang den Wert „NaN“ oder „∞“ annimmt.

Eigenschaften

Ausgangsfunktion: $$y(k) = \begin{cases} False & \text{für } u(k) \overset{equals}{=} u(k-1) \\ True & sonst \end{cases}$$

Funktionsblock-Testergebnis

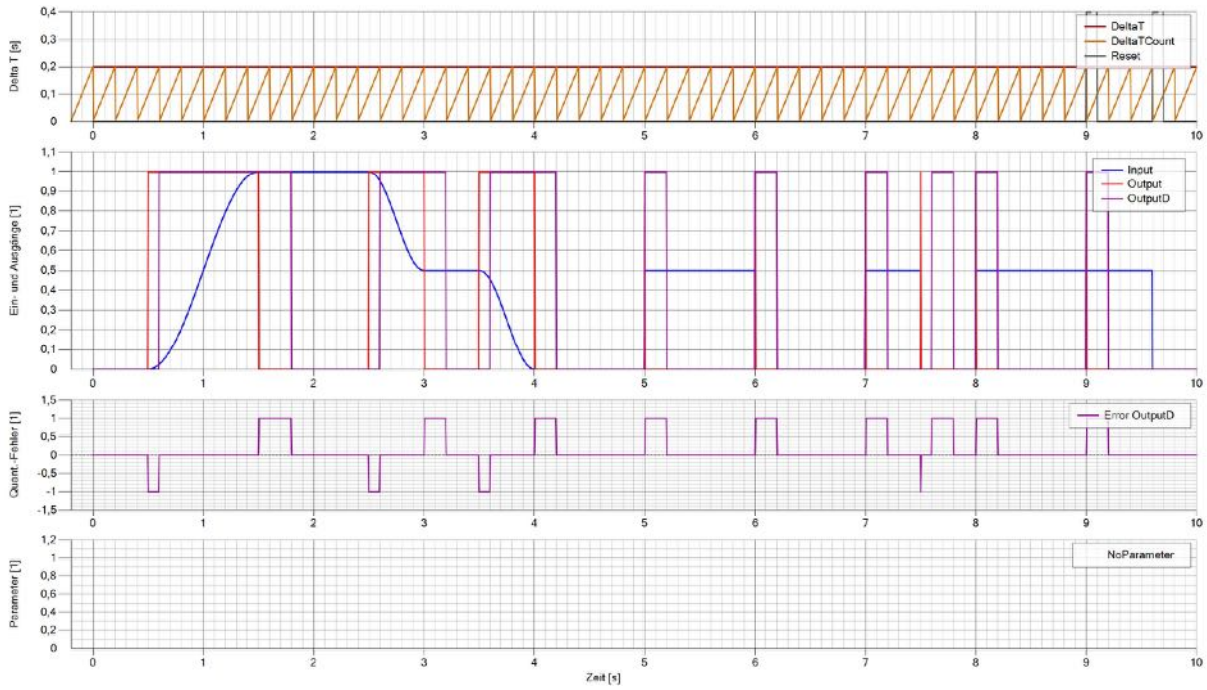


Abb. 3-38: Funktionsblock-Testergebnis: Wertänderung

0 - 5 Erkennen von Wertänderungen an Sinusprofilen

5 - 7 Erkennen von Wertänderungen an Flanken

7 - 9 Erkennen von Wertänderungen bei $Input = NaN$

8 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Programmcode

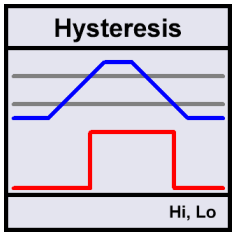
```
'#### Wertänderung ####
Public Class Change
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(Not u(k - 0).Equals(u(k - 1)), 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Analoge Übertragungsglieder“ → Seite 105

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.5.2 Hysterese (Analog.Hysteresis)



Der Funktionsblock „Hysteresis“ schaltet seinen Ausgangswert beim Erreichen des Schwellwerts „Hi“ auf den Wert „1“ bzw. „True“ und erst beim Unterschreiten des Schwellwerts „Lo“ wieder auf den Wert „0“ bzw. „False“. Ein Beispiel für dieses Verhalten ist aus der Regelungstechnik der Zweipunktregler einer Heizungssteuerung.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} False & \text{für } u(k) <= Lo \\ True & \text{für } u(k) >= Hi \\ y(k-1) & sonst \end{cases}$

Parameter: Hi = Schwellwert zum Einschalten

Lo = Schwellwert zum Ausschalten

Funktionsblock-Testergebnis

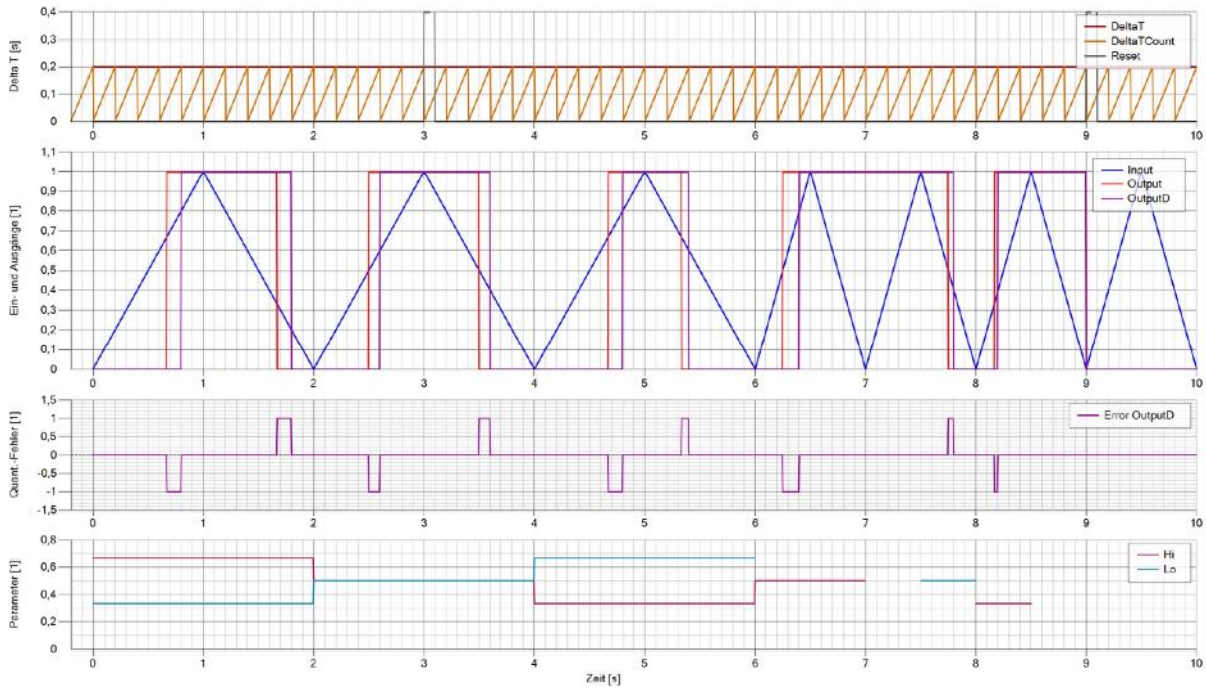


Abb. 3-39: Funktionsblock-Testergebnis: Hysterese

0 - 2 Verhalten bei $Hi = 0{,}66\ldots$ und $Lo = 0{,}33\ldots$

2 - 4 Verhalten bei $Hi = 0{,}5$ und $Lo = 0{,}5$ (Schwellwerte liegen aufeinander)

4 - 6 Verhalten bei $Hi = 0{,}33\ldots$ und $Lo = 0{,}66\ldots$ (Schwellwerte überschneiden sich)

6 - 9 Schwellwerte können mit $Hi = NaN$ oder $Lo = NaN$ deaktiviert werden

9 - 10 Zurücksetzen durch Aufrufen der Funktion „$Reset$“

Programmcode

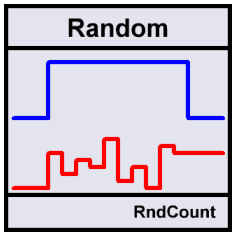
```
'#### Hysterese ####
Public Class Hysteresis
    Inherits BaseClass.FuncRetrospective

    Property Hi As Double = 2 / 3 'Schwellwert zum Einschalten
    Property Lo As Double = 1 / 3 'Schwellwert zum Ausschalten

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) <= Lo, 0, If(u(k - 0) >= Hi, 1, y(k - 1)))
    End Function

End Class
```

Verweise

Namensbereich „Analoge Übertragungsglieder“ → Seite 105

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.5.3 Zufallsgenerator (Analog.Random)



Der Funktionsblock „Random“ gibt mit jedem Aufruf der Ausgangsfunktion einen Zufallswert zurück. Wird als Eingangswert ein logisches „True“ übergeben, generiert der Funktionsblock einen neuen Zufallswert, sonst gibt er den letzten Wert zurück. Die Zufallswerte liegen dabei im Wertebereich $[0;1[$ und sind standardmäßig gleich verteilt. Jedoch kann auch eine andere Wahrscheinlichkeitsverteilung erreicht werden, indem man mit dem Parameter „RndCount“ bestimmt, aus wie vielen gleich verteilten Zufallswerten (ferner Würfeln) der Mittelwert gebildet werden soll, um eine Zufallszahl zu generieren.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} \frac{1}{RndCount} \cdot \sum_{i=1}^{RndCount} \mathrm{Rnd_i}(t) & \text{für } u(k) = True \\ y(k-1) & \text{für } u(k) = False \end{cases}$

Zufallsfunktion: $\mathrm{Rnd_i}(t)$ = gleich verteilte Zufallswerte des Wertebereichs $[0;1[$

Parameter: $RndCount$ = Anzahl der Würfel

Funktionsblock-Testergebnis

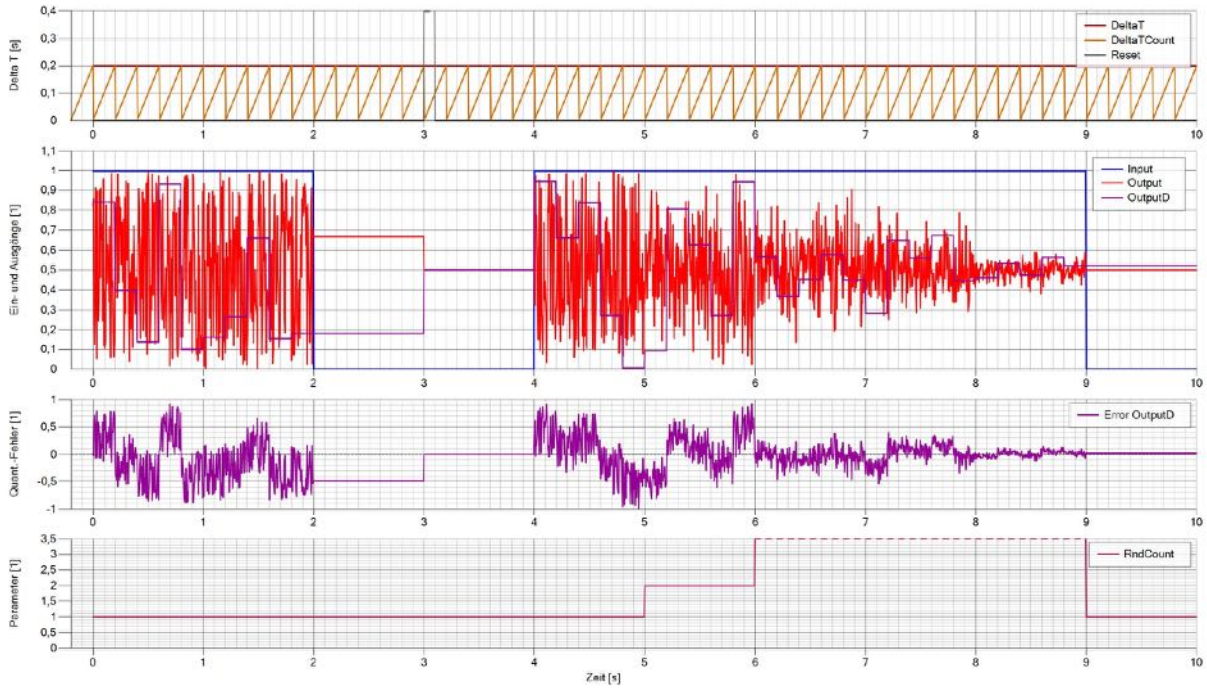


Abb. 3-40: Funktionsblock-Testergebnis: Zufallsgenerator

0 - 2 Verhalten bei $Input = 1$ und $RndCount = 1$ (gleich verteilte Zufallszahlen)

2 - 3 Verhalten bei $Input = 0$ (letzte Zufallszahl halten)

3 - 4 Zurücksetzen durch Aufrufen der Funktion „$Reset$“ (der Wert $0{,}5$ wird ausgegeben)

4 - 9 Verhalten bei $Input = 1$ und $RndCount = 1, 2, 4, 8, 64$
(Zufallszahlen verschiedener Wahrscheinlichkeitsverteilungen)

2 - 3 Verhalten bei $Input = 0$ (letzte Zufallszahl halten)

Mathematischer Hintergrund

Die folgenden Diagramme zeigen die Auswirkung des Parameters „RndCount“ auf die Wahrscheinlichkeitsverteilung $p(y)$ der Zufallszahlen.

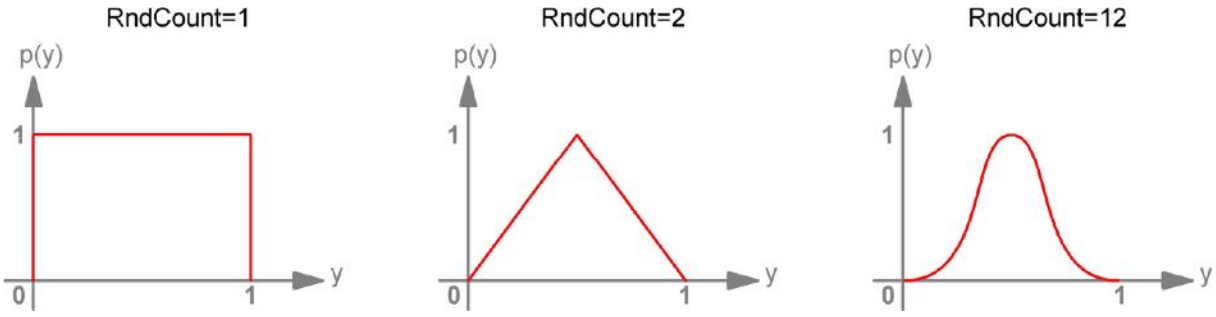


Abb. 3-41: Auswirkung des Parameters „RndCount“ auf die Wahrscheinlichkeitsverteilung

Programmcode

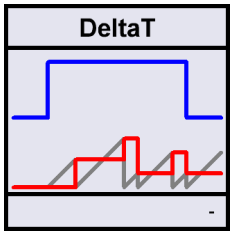
```
'#### Zufallsgenerator ####
Public Class Random
    Inherits BaseClass.FuncRetrospective

    Property RndCount As Double = 1 'Anzahl der Würfel (Beeinflusst die Wahrscheinlichkeitsverteilung)
    Shared Rnd As New System.Random(Convert.ToInt32(Now.Ticks Mod Integer.MaxValue)) 'Zufallsgenerator

    Protected Overrides Function Functionality() As Double
        If Not (RndCount >= 1 And RndCount <= Int32.MaxValue) Then
            Throw New Exception("RndCount is outside the allowed value range!")
        End If
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            Dim Value As Double
            'Alle Würfel werfen
            For i As Integer = 0 To Convert.ToInt32(RndCount) - 1
                Value += Rnd.NextDouble Mod 1 'Würfeln
            Next
            Value /= RndCount 'Mittelwert aller Würfel
            Return Value
        Else 'Bei Input = False
            Return y(k - 1) 'Letzten Zufallswert zurückgeben
        End If
    End Function

    Public Overrides Sub Reset()
        'Semaphore setzen, weil die Funktion "SetInitialConditions" selbst "Reset" aufruft.
        Static Semaphor As Integer = 0
        If Threading.Interlocked.Exchange(Semaphor, 1) = 1 Then Exit Sub
        Me.SetInitialConditions({}, {0, 0.5}) 'Letzten Zufallswert auf 0,5 setzen
        Semaphor = 0
    End Sub

End Class
```

Verweise

Namensbereich „Analoge Übertragungsglieder“ → Seite 105

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

3.5.4 Zykluszeit (Analog.DeltaT)



Der Funktionsblock „DeltaT“ gibt beim Aufrufen der Ausgangsfunktion, unabhängig von dem dabei übergebenen Eingangswert, die seit dem letzten Funktionsaufruf verstrichene Zeit zurück. Hierzu nutzt der Funktionsblock den von der Hardware und dem Betriebssystem unterstützten hochauflösenden Zeitgeber, der bei einer Intel® Core™ i7-4910MQ und Windows 8.1 mit einer Frequenz von 2.825.501 Hz getaktet ist. Um fortpflanzende Zeitfehler zu vermeiden, wird zum Bestimmen der Zykluszeit der Zeitgeber nicht neu gestartet, sondern lediglich der aktuelle Zeitwert ausgelesen und die Differenz zu dem Zeitwert des letzten Funktionsaufrufs berechnet.

Der Funktionsblock wurde entwickelt, um die Zykluszeit für zeitabhängige Funktionsblöcke zu bestimmen, wenn diese für Systeme eingesetzt werden, die von der realen Zeit abhängig sind.

Eigenschaften

Ausgangsfunktion: $y(k) = T_k - T_{k-1}$ T_x = Zeitpunkt des Funktionsblockaufrufs

Funktionsblock-Testergebnis

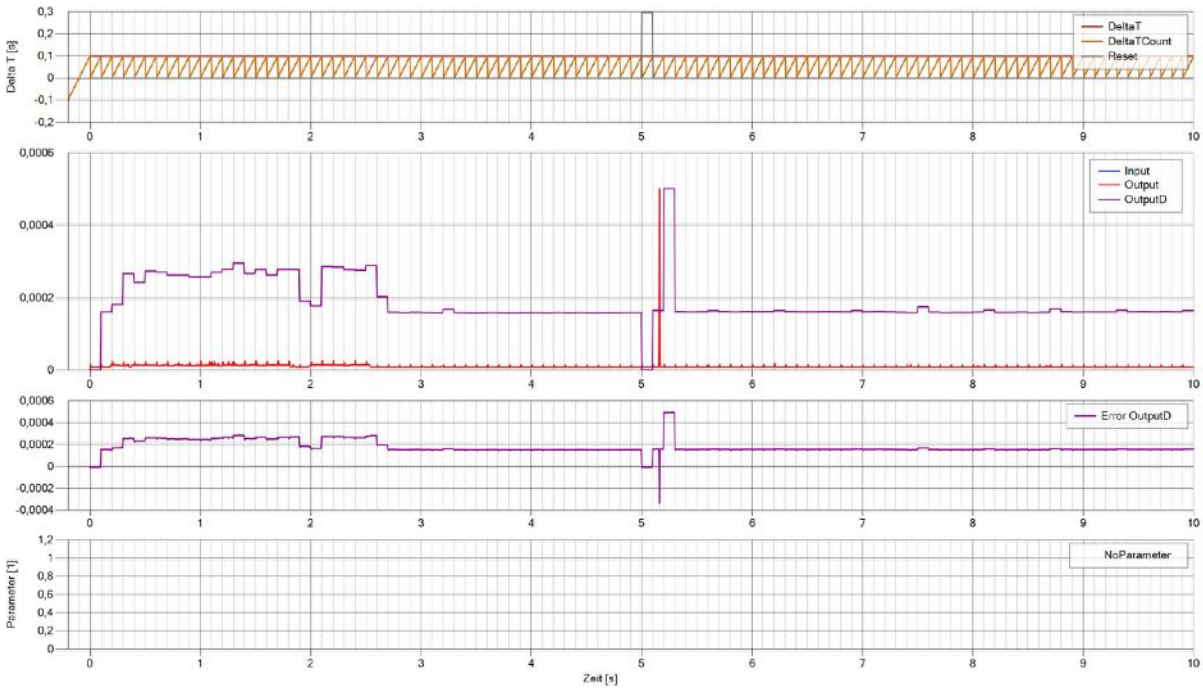


Abb. 3-42: Funktionsblock-Testergebnis: Zykluszeit

0 - 10 Reale Zykluszeiten, während der Funktionsblocktest ausgeführt wurde. Zu erkennen ist, dass das Berechnen des Simulations-Ergebnisses von einer Simulations-Zeitspanne von **0,1** Sekunden („DeltaT“ erstes Diagramm) gerade einmal durchschnittlich ca. **0,0002** Sekunden („OutputD“ zweites Diagramm) der realen Zeit in Anspruch nimmt. Die schwankenden Zykluszeiten sind auf das nicht hart-echtzeitfähige Betriebssystem Windows 8.1 zurückzuführen, dem die Testumgebung unterworfen ist.

Programmcode

```
'#### Zykluszeit ####
Public Class DeltaT
    Inherits BaseClass.FuncRetrospective

    Private StopWatch As New Stopwatch 'Hochauflösender Zeitgeber
    Private LastElapsedTicks As Long 'Verstrichene Zeit seit dem ersten Funktionsaufruf in Ticks

    Protected Overrides Function Functionality() As Double
        'Verstrichene Zeit seit erstem Funktionsaufruf ermitteln in Ticks
        Dim ElapsedTicks As Long = StopWatch.ElapsedTicks
        'Verstrichene Zeit seit letztem Funktionsaufruf in Sekunden ermitteln
        Dim DeltaT As Double = (ElapsedTicks - LastElapsedTicks) / Stopwatch.Frequency
        'Verstrichene Zeit seit erstem Funktionsaufruf merken
        LastElapsedTicks = ElapsedTicks
        'Zeitgeber beim ersten Durchlauf starten (danach nicht mehr zurücksetzen)
        If Not StopWatch.IsRunning Then StopWatch.Start()
        'Verstrichene Zeit seit dem letzten Funktionsaufruf zurückgeben
        Return DeltaT
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch.Reset() 'Zeitgeber zurücksetzen
        LastElapsedTicks = 0 'Seit erstem Funktionsaufruf verstrichene Zeit zurücksetzen
    End Sub

End Class
```

Verweise

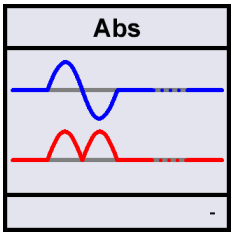
Namensbereich „Analoge Übertragungsglieder“ → Seite 105

Basisklasse „Rückblickender Funktionsblock“ → Seite 32

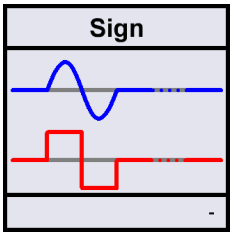
3.6 Grundlegende Funktionen (Generic)

Der Namensbereich „Generic“ enthält grundlegende mathematische Funktionen wie zum Beispiel die Betrags-, Vorzeichen- und die Modulo-Funktion. Der Ausgangswert aller Funktionsblöcke des Namensbereichs hängt ausschließlich vom Eingangswert und den Parametern ab. Es werden also keine Werte der letzten Funktionsaufrufe intern gespeichert.

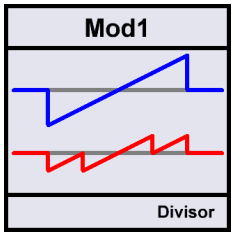
Der Funktionsblock „FuncOfSegments“ ist ein besonderer Funktionsblock des Namensbereichs „Generic“. Er ermöglicht das abschnittsweise Definieren einer mathematischen Funktion aus Funktionssegmenten, die im Namensbereich „FuncSegment“ enthalten sind. Eine Beispielanwendung für diesen Funktionsblock ist die elektronische Kurvenscheibe, mit der elektrische Antriebe von mechanischen Achsen über beliebige mathematische Funktonen miteinander gekoppelt werden können.



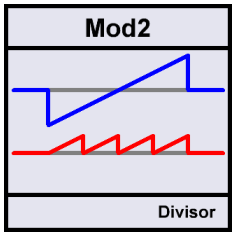
→ Seite 117



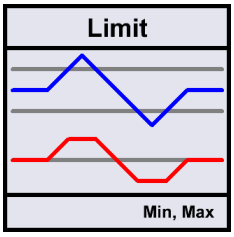
→ Seite 119



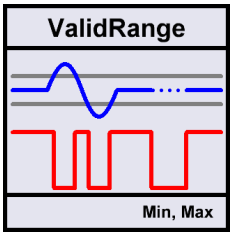
→ Seite 121



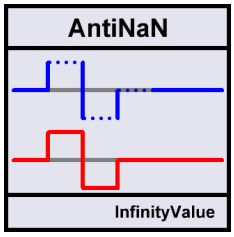
→ Seite 123



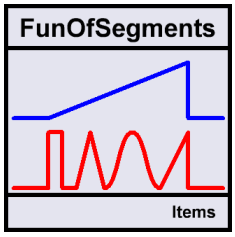
→ Seite 125



→ Seite 127



→ Seite 129

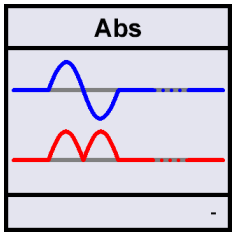


→ Seite 131

Verweise

Basisklasse „Funktionsblock“	→ Seite 31
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38

3.6.1 Betrag (Generic.Abs)



Der Funktionsblock „Abs“ berechnet den Betrag des Eingangswerts $u(k)$ und gibt ihn als Ausgangswert $y(k)$ zurück.

Eigenschaften

Ausgangsfunktion: $y(k) = |u(k)| = \begin{cases} u(k) & \text{für } u(k) \geq 0 \\ -u(k) & \text{für } u(k) < 0 \end{cases}$

Funktionsblock-Testergebnis

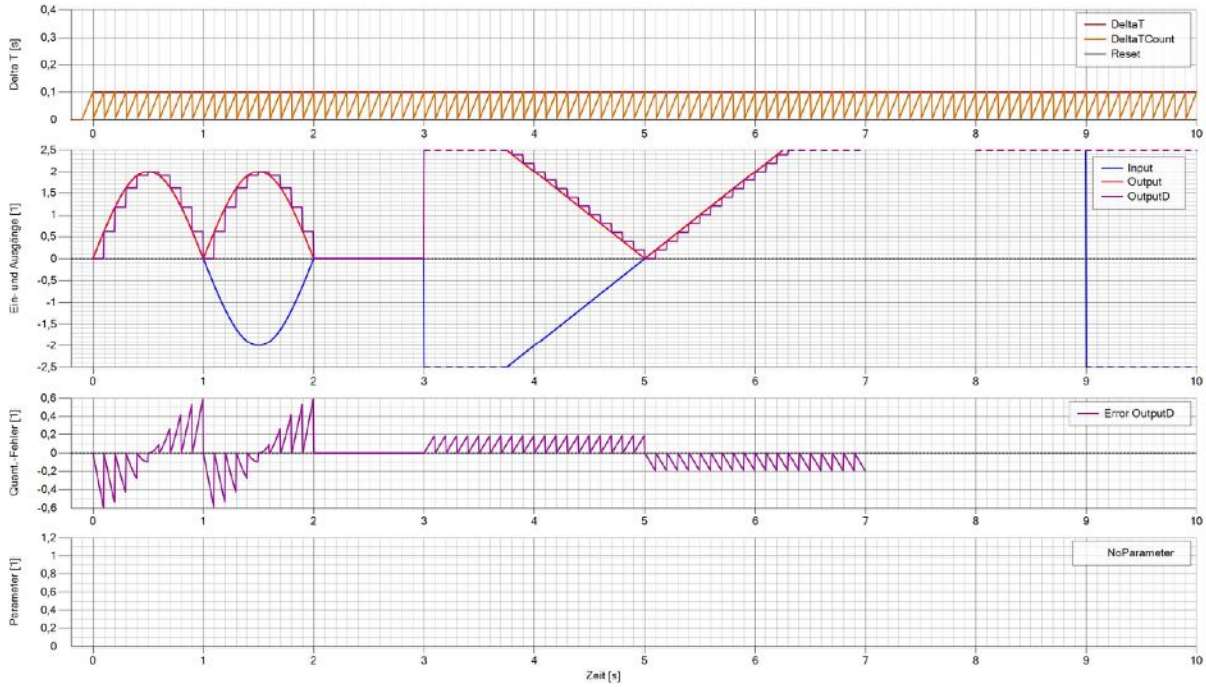


Abb. 3-43: Funktionsblock-Testergebnis: Betrag

0 - 2 Betrag einer Sinus-Funktion (blau verbirgt sich hinter rot)

2 - 3 Betrag von 0 ist 0

3 - 7 Betrag einer linearen Funktion (blau verbirgt sich hinter rot)

7 - 8 Betrag von NaN ist NaN

8 - 10 Betrag von $+\infty$ oder $-\infty$ ist $+\infty$

Programmcode

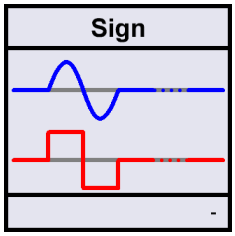
```
'#### Betrag ####
Public Class Abs
    Inherits BaseClass.Func

    Public Overrides Function Output(ByVal Input As Double) As Double
        Return Math.Abs(Input)
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.2 Vorzeichen (Generic.Sign)



Der Funktionsblock „Sign“ gibt beim Aufruf als Ausgangswert $y(k)$ das Vorzeichen des Eingangswerts $u(k)$ zurück.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} +1 & \text{für } u(k) > 0 \\ 0 & \text{für } u(k) = 0 \\ -1 & \text{für } u(k) < 0 \\ NaN & sonst \end{cases}$

Funktionsblock-Testergebnis

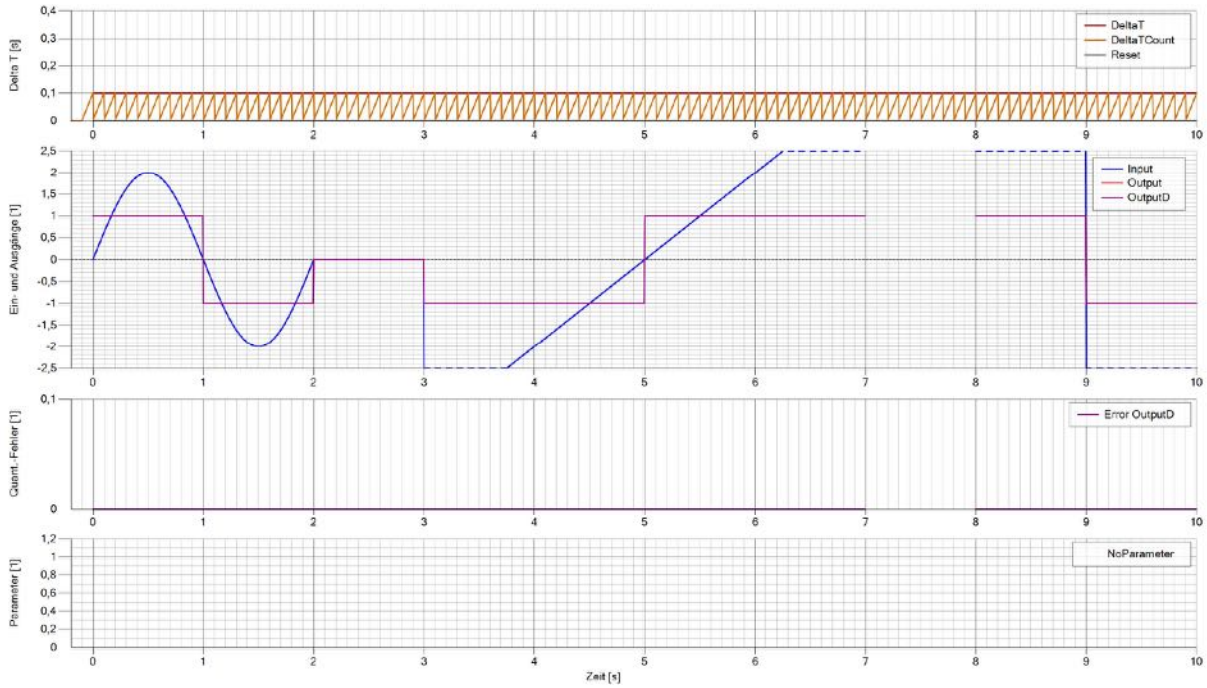


Abb. 3-44: Funktionsblock-Testergebnis: Vorzeichen

0 - 2 Vorzeichen bei einer Sinus-Funktion

2 - 3 Vorzeichen von 0 ist 0

3 - 7 Vorzeichen bei einer linearen Funktion

7 - 8 Betrag von NaN ist NaN

8 - 10 Vorzeichen von $+\infty$ und $-\infty$ ist $+1$ bzw. -1

Programmcode

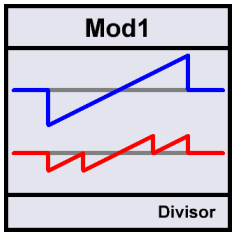
```
'#### Vorzeichen ####
Public Class Sign
    Inherits BaseClass.Func

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return Double.NaN
        Return Math.Sign(Input)
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.3 Modulo1 (Generic.Mod1)



Der Funktionsblock „Mod1“ gibt beim Aufruf der Ausgangsfunktion den Rest, der beim Dividieren des Eingangswerts $u(k)$ durch den Parameter $Divisor$ übrig bleibt, zurück.

Eigenschaften

Ausgangsfunktion: $y(k) = u(k)\ \text{Mod}\ |Divisor|$

Parameter: $Divisor$ = Divisor

Funktionsblock-Testergebnis

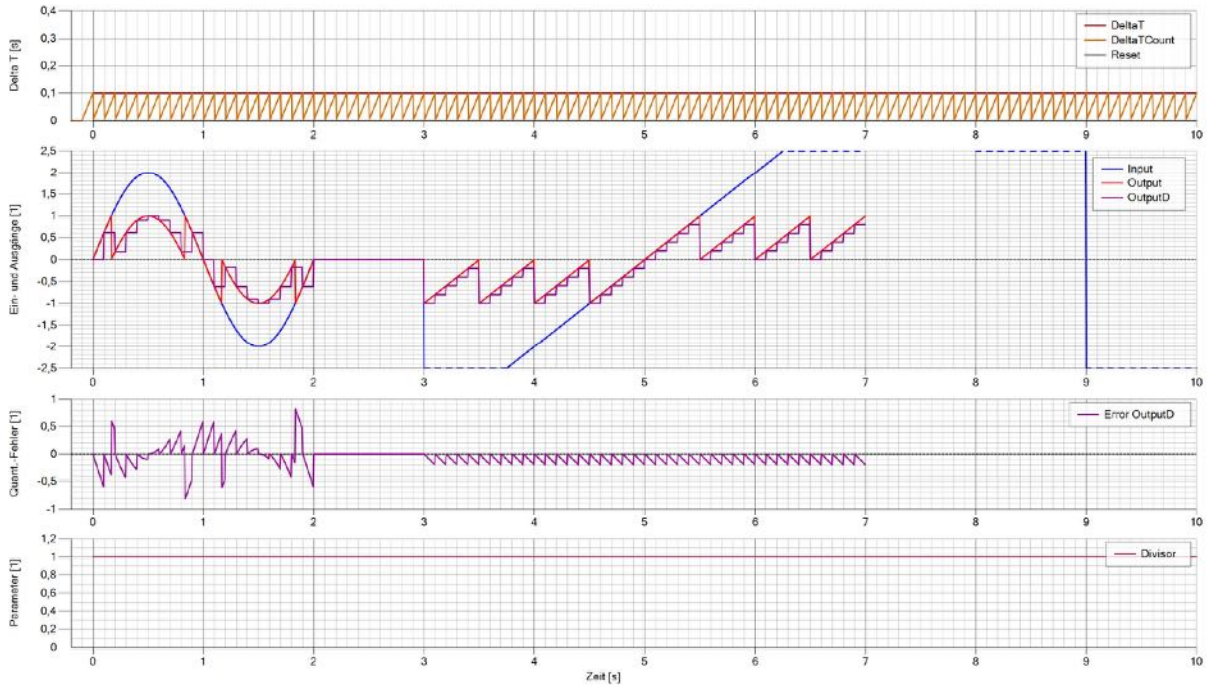


Abb. 3-45: Funktionsblock-Testergebnis: Modulo1

0 - 2 Modulo mit $Divisor = 1$ bei einer Sinus-Funktion

2 - 3 Modulo mit $Divisor = 1$ von 0 ist 0

3 - 7 Modulo mit $Divisor = 1$ bei einer linearen Funktion
(Eine Punktsymmetrie ist im Ausgangssignal zu erkennen)

7 - 8 Modulo mit $Divisor = 1$ von NaN ist NaN

8 - 10 Modulo mit $Divisor = 1$ von $\pm\infty$ ist NaN

Programmcode

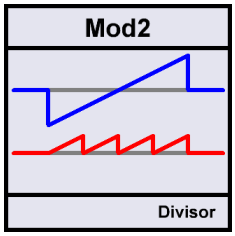
```
'#### Modulo1 ####
Public Class Mod1
    Inherits BaseClass.Func

    Property Divisor As Double = 1 'Divisor

    Public Overrides Function Output(ByVal Input As Double) As Double
        Return Input Mod Divisor
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.4 Modulo2 (Generic.Mod2)



Der Funktionsblock „Mod2“ gibt beim Aufruf der Ausgangsfunktion den Rest, der beim Dividieren des Eingangswerts $u(k)$ durch den Parameter $Divisor$ übrig bleibt, zurück. Im Gegensatz zum Funktionsblock „Mod1“ wird hier bei negativen Eingangswerten der Betrag des Divisors auf das Modulo-Ergebnis aufaddiert.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} u(k) \text{ Mod } |Divisor| & \text{für } u(k) \geq 0 \\ (u(k) \text{ Mod } |Divisor|) + |Divisor| & \text{für } u(k) < 0 \end{cases}$

Parameter: $Divisor$ = Divisor

Funktionsblock-Testergebnis

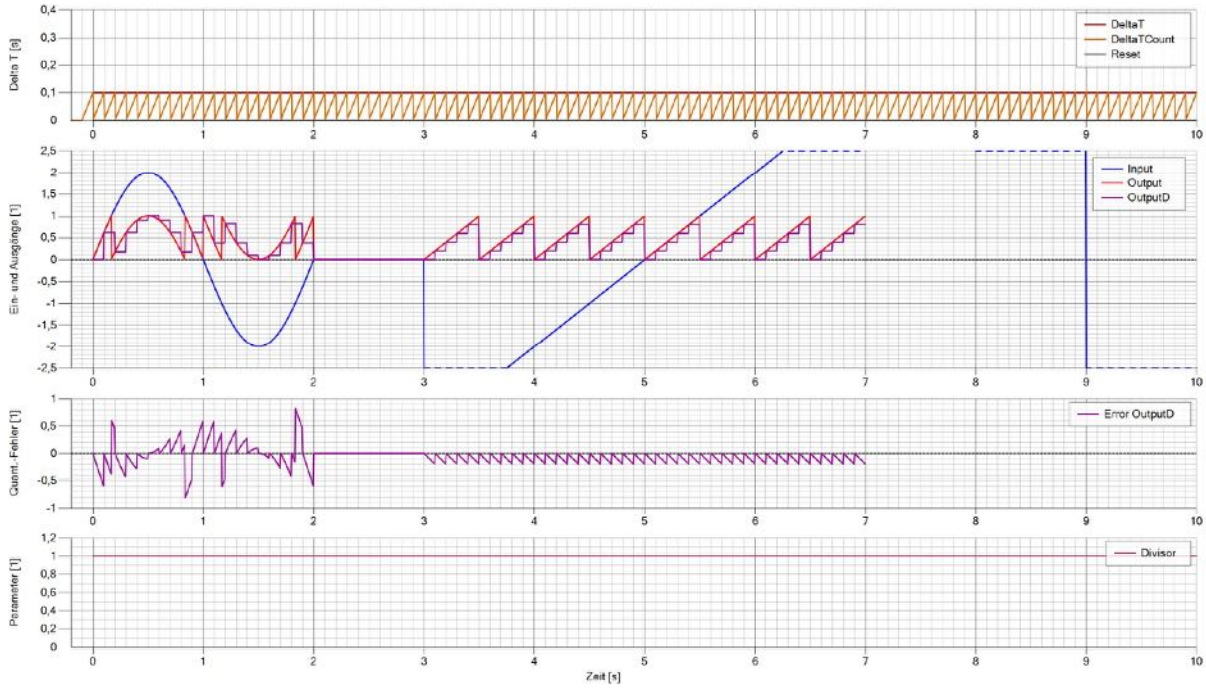


Abb. 3-46: Funktionsblock-Testergebnis: Modulo2

0 - 2 Modulo mit $Divisor = 1$ bei einer Sinus-Funktion

2 - 3 Modulo mit $Divisor = 1$ von 0 ist 0

3 - 7 Modulo mit $Divisor = 1$ bei einer Linearen Funktion
(Nulldurchgang ist im Ausgangssignal nicht zu erkennen)

7 - 8 Modulo mit $Divisor = 1$ von NaN ist NaN

8 - 10 Modulo mit $Divisor = 1$ von $\pm\infty$ ist NaN

Programmcode

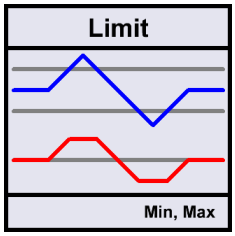
```
'#### Modulo2 ####
Public Class Mod2
    Inherits BaseClass.Func

    Property Divisor As Double = 1 'Divisor

    Public Overrides Function Output(ByVal Input As Double) As Double
        Input = Input Mod Divisor 'Eingangswert in den Bereich ]-1 to 1[ bringen
        Input += If(Input < 0, Math.Abs(Divisor), 0) 'Eingangswert in den Def.bereich [0 to 1[ bringen
        Return Input
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.5 Begrenzer (Generic.Limit)



Der Funktionsblock „Limit“ begrenzt die Eingangswerte auf einen definierbaren Wertebereich, der durch die Parameter „Max“ und „Min“ festgelegt wird. Falls sich die Bereichsgrenzen überschneiden, hat die Obergrenze höhere Priorität. Um eine Grenze zu deaktivieren, kann sie mit dem Wert „NaN“ belegt werden.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Max & \text{für } u(k) > Max \\ Min & \text{für } u(k) < Min \\ u(k) & sonst \end{cases}$

Parameter: Max = Wertebereich Obergrenze

Min = Wertebereich Untergrenze

Funktionsblock-Testergebnis

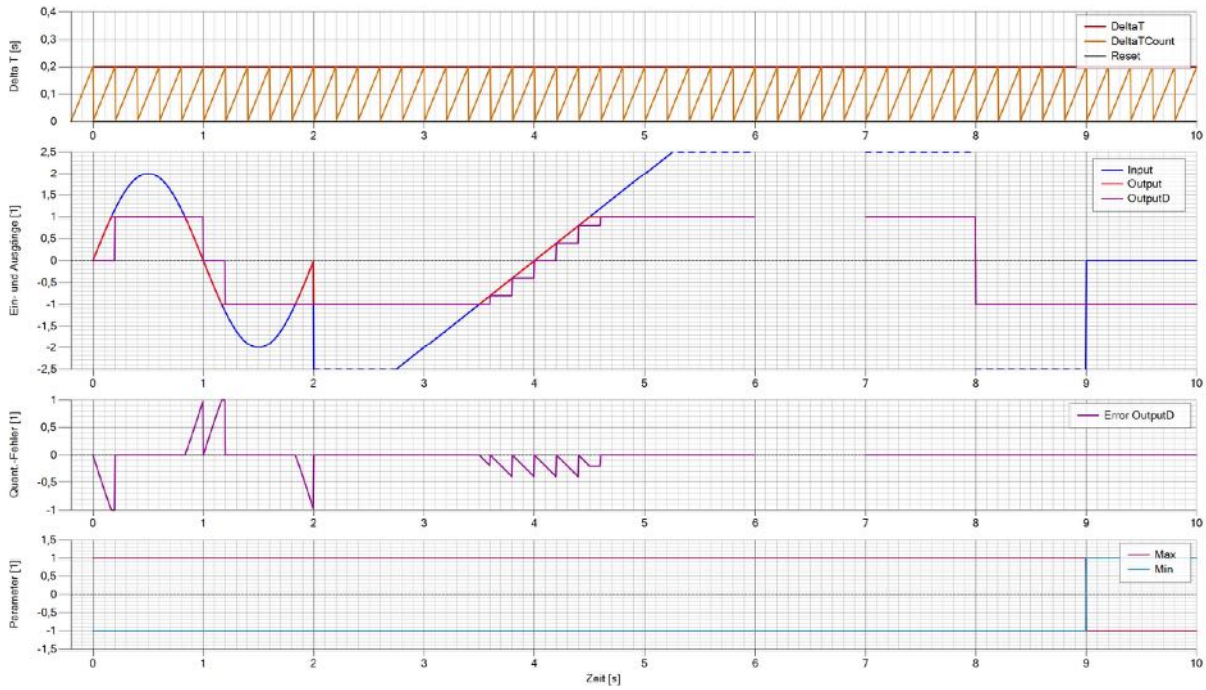


Abb. 3-47: Funktionsblock-Testergebnis: Begrenzer

0 - 2 Begrenzen einer Sinus-Funktion auf den Wertebereich $[-1; 1]$

2 - 6 Begrenzen einer linearen Funktion auf den Wertebereich $[-1; 1]$

6 - 7 Der Eingangswert NaN hat den Ausgangswert NaN zur Folge

7 - 9 Die Eingangswerte $+\infty$ und $-\infty$ werden auch begrenzt

9 - 10 Begrenzen auf den Wertebereich $[1; -1]$ (die Obergrenze hat höhere Priorität)

Programmcode

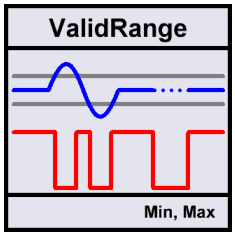
```
'#### Begrenzer ####
Public Class Limit
    Inherits BaseClass.Func

    Property Max As Double = +1 'Wertebereich Obergrenze
    Property Min As Double = -1 'Wertebereich Untergrenze

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Input > Max Then Return Max 'Auf Obergrenze begrenzen
        If Input < Min Then Return Min 'Auf Untergrenze begrenzen
        Return Input
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.6 Gültigkeitsbereich (Generic.ValidRange)



Der Funktionsblock „ValidRange“ überprüft, ob sich der Eingangswert in einem definierbaren Wertebereich befindet, der durch die Parameter „Max“ und „Min“ mit $[Min; Max]$ festgelegt wird. Um eine Grenze zu deaktivieren, kann sie mit dem Wert „$+\infty$“ bzw. „$-\infty$“ belegt werden.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} True & \text{für } Max \geq u(k) \geq Min \\ False & sonst \end{cases}$

Parameter:
- Max = Wertebereich Obergrenze
- Min = Wertebereich Untergrenze

Funktionsblock-Testergebnis

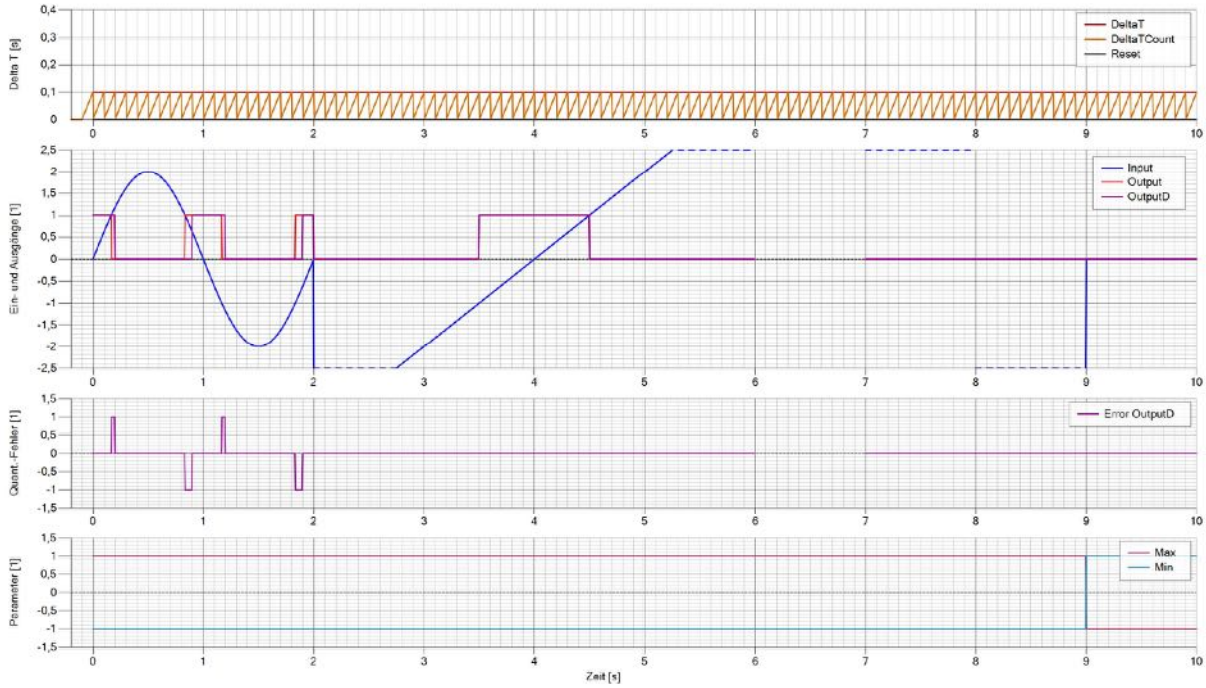


Abb. 3-48: Funktionsblock-Testergebnis: Gültigkeitsbereich

0 - 2 Sinus-Funktion auf den Wertebereich $[-1; 1]$ prüfen

2 - 6 Lineare Funktion auf den Wertebereich $[-1; 1]$ prüfen

6 - 7 Der Eingangswert NaN hat den Ausgangswert NaN zur Folge

7 - 9 Die Eingangswerte $+\infty$ und $-\infty$ liegen außerhalb des Wertebereichs $[-1; 1]$

9 - 10 Der Wertebereich $[-1; 1]$ überschneidet sich und wird niemals $True$ liefern

Programmcode

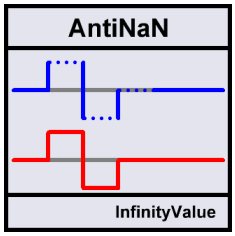
```
'#### Gültigkeitsbereich ####
Public Class ValidRange
    Inherits BaseClass.Func

    Property Min As Double = -1
    Property Max As Double = +1

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return Double.NaN
        Return If(Input >= Min And Input <= Max, 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.7 AntiNaN (Generic.AntiNaN)



Der Funktionsblock „AntiNaN“ ersetzt den besonderen „Double“-Wert „NaN“ durch „0“ und begrenzt den Eingangswert auf den Wertebereich $[-InfinityValue;\ InfinityValue]$, um auch die besonderen „Double“-Werte „+∞“ und „−∞“ zu unterdrücken. Der Parameter „InfinityValue“ ist standardmäßig mit dem Konstantwert „Single.MaxValue“ bzw. $3{,}402823 \cdot 10^{38}$ vorbelegt.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} 0 & \text{für } u(k) \overset{equals}{=} NaN \\ InfinityValue & \text{für } u(k) > InfinityValue \\ -InfinityValue & \text{für } u(k) < -InfinityValue \\ u(k) & sonst \end{cases}$

Parameter: $InfinityValue$ = Ersatzwert für den Wert „unendlich“

Funktionsblock-Testergebnis

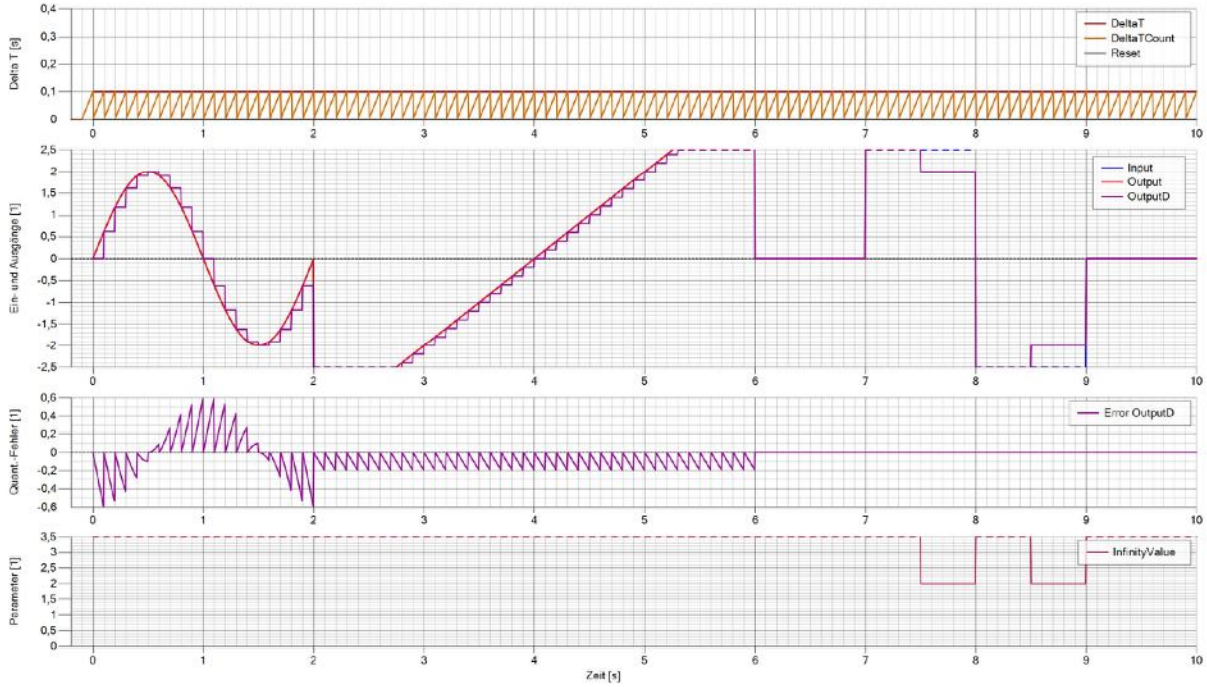


Abb. 3-49: Funktionsblock-Testergebnis: AntiNaN

0 - 2 Sinus-Funktion liegt im Wertebereich $[-InfinityValue; InfinityValue]$

2 - 6 Lineare Funktion liegt im Wertebereich $[-InfinityValue; InfinityValue]$

6 - 7 Der Eingangswert NaN hat den Ausgangswert 0 zur Folge (AntiNaN)

7 - 9 Die Eingangswerte +∞ und −∞ werden auf den Wertebereich $[-InfinityValue; InfinityValue]$ begrenzt

9 - 10 Der Eingangswert 0 liegt im Wertebereich $[-InfinityValue; InfinityValue]$

Programmcode

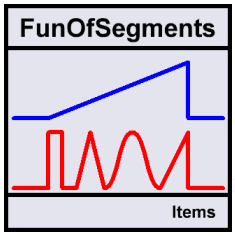
```
'#### AntiNaN ####
Public Class AntiNaN
    Inherits BaseClass.Func

    Property InfinityValue As Double = Single.MaxValue 'Ersatzwert für den Wert "unendlich"

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return 0
        If Input > InfinityValue Then Return InfinityValue
        If Input < -InfinityValue Then Return -InfinityValue
        Return Input
    End Function

End Class
```

Verweise

Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

3.6.8 Abschnittsweise definierte Funktion (FuncOfSegments)



Mit dem Funktionsblock „FuncOfSegments“ kann eine mathematische Funktion abschnittsweise definiert werden. Hierzu stellt der Funktionsblock eine öffentliche Liste als Parameter zur Verfügung, der Funktionssegmente des Namensbereichs „FuncSegment“ (Seite 133) hinzugefügt werden können. Diese bilden dann die abschnittsweise definierte Ausgangsfunktion. Der Parameter „Length“ eines jeden Funktionssegments legt dabei die Länge dessen Definitionsbereichs fest.

Das folgende Beispiel zeigt, wie die Funktionssegmente aneinandergehängt werden und welche Werte die Ausgangsfunktion $y(x)$ an den Segmentübergängen zurückgibt.

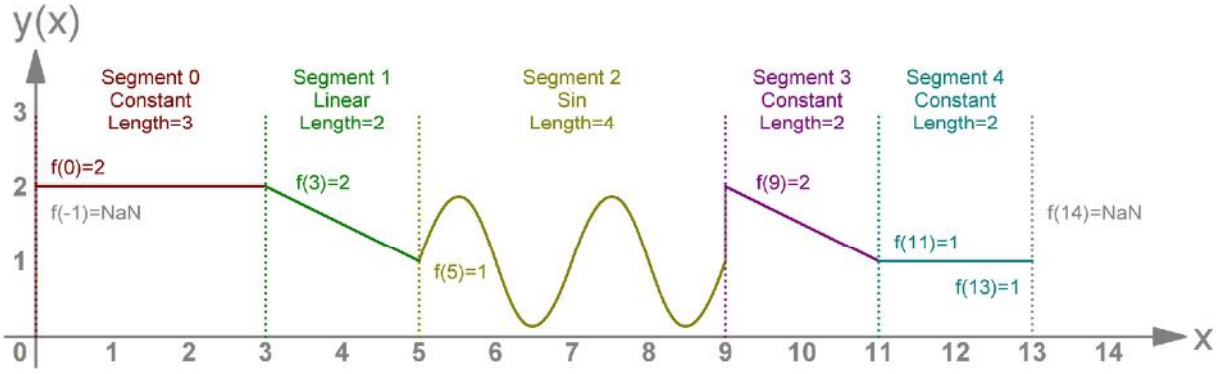


Abb. 3-50: Beispiel für eine abschnittsweise definierte Funktion

Um den Funktionsblock „FuncOfSegments“ zu testen, wurden mit der Testumgebung einige Einfangsfunktionen abschnittsweise definiert und deren Werteverlauf mit Hilfe der Wertetabelle der Testumgebung überprüft.

Funktionsblock-Testergebnis

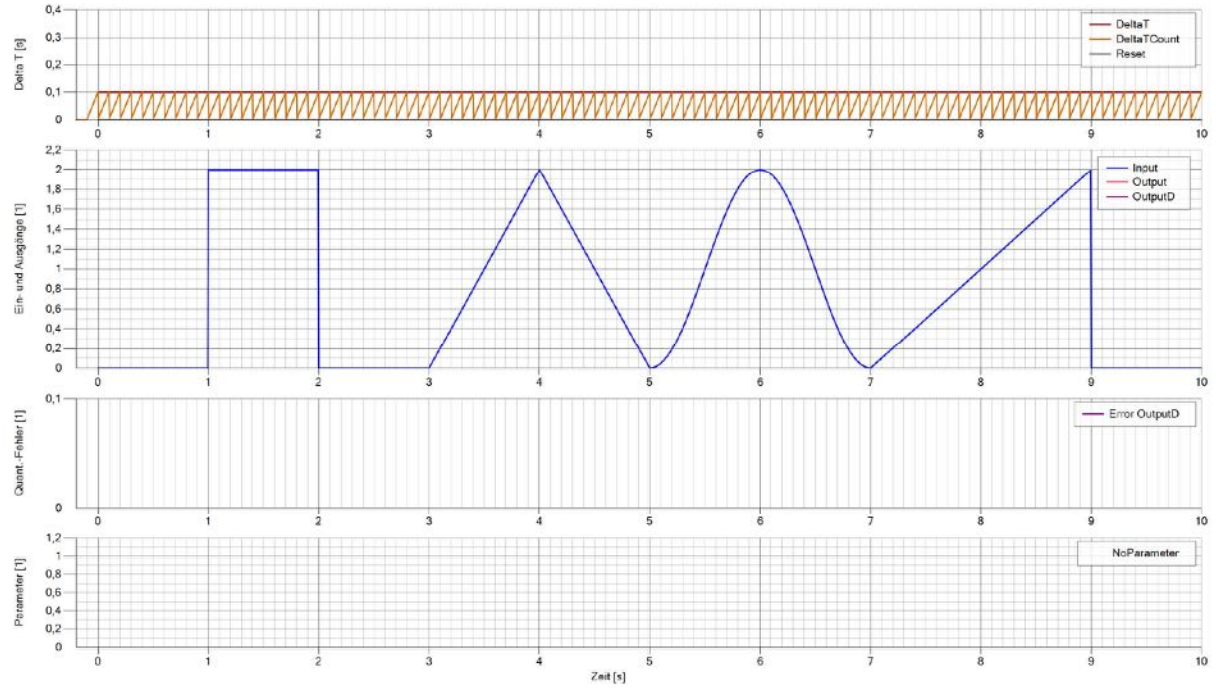


Abb. 3-51: Funktionsblock-Testergebnis: Abschnittsweise definierte Funktion

Hinweise

Bei der Berechnung des Ausgangssignals wird das Eingangssignal des Typs „Double“ vorübergehend auf die Genauigkeit „Single“ gerundet, um numerische Fehler zu unterdrücken.

Programmcode

```
'#### Abschnittsweise definierte Funktion ####
Public Class FuncOfSegments
    Inherits BaseClass.Func

    'Die Funktions-Segmente werden wie im folgenden Beispiel aneinandergereiht:
    'Segment a(x) mit der Länge 1 beschreibt den Bereich [0;1[
    'Segment b(x) mit der Länge 2 beschreibt den Bereich [1;3[
    'Segment c(x) mit der Länge 2 beschreibt den Bereich [3;5]
    'Die Aufrufe der Ausgangsfunktion geben dann folgende Werte zurück:
    'f(-0.1) => NaN
    'f(0.0) => a(0.0)
    'f(1.0) => b(0.0)
    'f(3.0) => c(0.0)
    'f(4.0) => c(1.0)
    'f(5.0) => c(2.0)
    'f(6.0) => NaN

    'Liste der Funktions-Segmente
    Property Items As New List(Of BaseClass.FuncSegment)

    Public Overrides Function Output(ByVal Input As Double) As Double
        'Numerische Fehler werden auf Kosten der Gleitkommazahl-Genauigkeit unterdrückt
        Dim Length As Double = GetLength() 'Länge zwischenspeichern für bessere Laufzeit
        If Items.Count = 0 Then Return Double.NaN 'Keine Funktions-Segmente
        If Input < 0 Then Return Double.NaN 'Außerhalb der unteren Definitionsgrenze
        If Convert.ToDouble(Convert.ToSingle(Input)) = Convert.ToDouble(Convert.ToSingle(Length)) _
        AndAlso Items.Count > 0 Then 'Auf der oberen Definitionsgrenze (mit Toleranz Single.Epsilon)
            Return Items(Items.Count - 1).Output(Items(Items.Count - 1).Length())
        End If
        If Input > Length Then Return Double.NaN 'Außerhalb der oberen Definitionsgrenze
        'Funktions-Segmente bis zur gesuchten X-Position durchlaufen und den Funktions-Wert zurückgeben
        Dim PosX As Double = 0 'X-Position
        For Each FunSegment As BaseClass.FuncSegment In Items 'Alle Funktionssegmente durchlaufen
            'Numerische Fehler unterdrücken, indem auf Single-Genauigkeit gerundet wird
            Dim PosInSegRounded As Double = Convert.ToDouble(Convert.ToSingle(Input - PosX))
            Dim FunSegLengthRounded As Double = Convert.ToDouble(Convert.ToSingle(FunSegment.Length))
            'Ist die Position im Definitionsbereich des Funktionssegments?
            If PosInSegRounded < FunSegLengthRounded Then
                'Durch die Konvertierung in Single kann PosInSegment minimal kleiner als 0 oder minimal
                'größer als FunSegment.Length() sein. Deshalb Wert wieder in den Def.-Bereich bringen.
                If PosInSegRounded < 0 Then PosInSegRounded = 0
                If PosInSegRounded > FunSegment.Length() Then PosInSegRounded = FunSegment.Length()
                Return FunSegment.Output(PosInSegRounded) 'Gebe den Funktionswert zur X-Position zurück
            End If
            PosX += FunSegment.Length() 'Verschiebe die aktuelle X-Position um die Segment-Länge weiter
        Next
        Return Double.NaN
    End Function
    Public Function GetLength() As Double 'Gesamtlänge aller Funktionssegmente
        Dim LengthDouble As Double 'Summe aller Funktionssegment-Längen
        For Each Segment As BaseClass.FuncSegment In Items 'Alle Funktionssegmente durchlaufen
            LengthDouble += Segment.Length() 'Längen aufaddieren
        Next
        Return Convert.ToDouble(Convert.ToSingle(LengthDouble)) 'Gesamtlänge zurückgeben
    End Function

End Class
```

Verweise

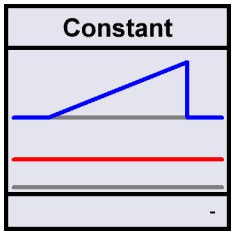
Namensbereich „Grundlegende Funktionen“ → Seite 115

Basisklasse „Funktionsblock“ → Seite 31

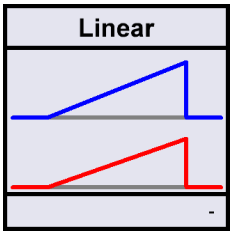
3.7 Funktions-Segmente (FuncSegment)

Unter dem Namensbereich „FuncSegments“ enthält die Bibliothek Funktionssegmente, die mathematische Funktionen in einem Definitionsbereich beschreiben. Der Parameter *Length* der Funktionssegmente legt diesen Definitionsbereich mit $[0; Length]$ fest. Zudem stellen die Funktionssegmente die Eigenschaften Offset, Amplitude, Frequenz, Phase und weitere funktionsspezifische Parameter zur Verfügung, mit denen die Funktionsform des Segments eingestellt werden kann.

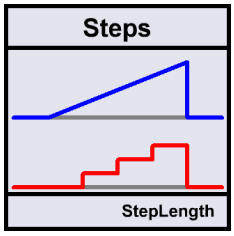
Mit dem Funktionsblock „FuncOfSegments“, der auf Seite 131 dokumentiert ist, können beliebig viele Funktionssegmente zusammengefügt werden. Der Funktionsblock beschreibt dann mit seiner Ausgangsfunktion eine mit den Segmenten abschnittsweise definierte Funktion.



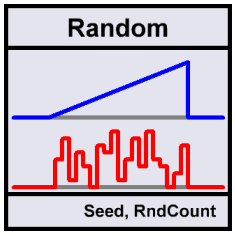
→ Seite 135



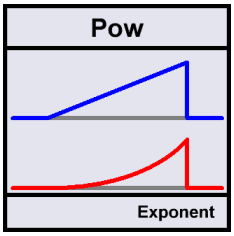
→ Seite 137



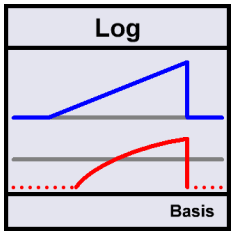
→ Seite 139



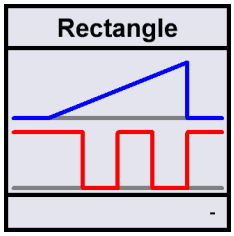
→ Seite 141



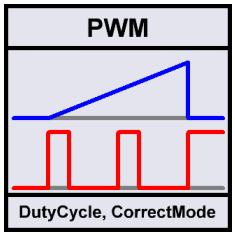
→ Seite 143



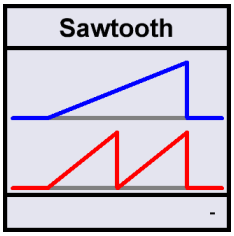
→ Seite 145



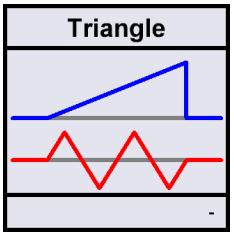
→ Seite 147



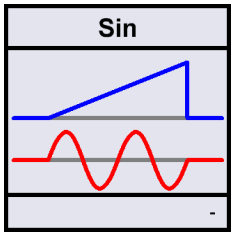
→ Seite 149



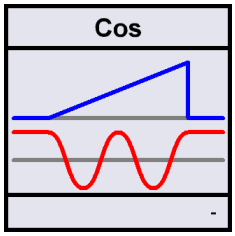
→ Seite 151



→ Seite 153



→ Seite 155

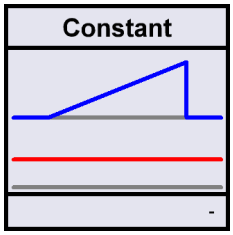


→ Seite 157

Verweise

Basisklasse „Funktionssegment“	→ Seite 34
Basisklasse „Periodisches Funktionssegment“	→ Seite 35
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38
Funktionsblock „FuncOfSegments“	→ Seite 131

3.7.1 Konstante Funktion (FuncSegment.Constant)



Das Funktionssegment „Constant“ beschreibt mit seiner Ausgangsfunktion eine konstante Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \leq u(k) \leq Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = 1$

Parameter: $Offset$ = Offset (Verschiebung in Y-Richtung)

$Factor$ = Verstärkung (wird von Basisklasse geerbt, aber ignoriert)

Funktionsblock-Testergebnis

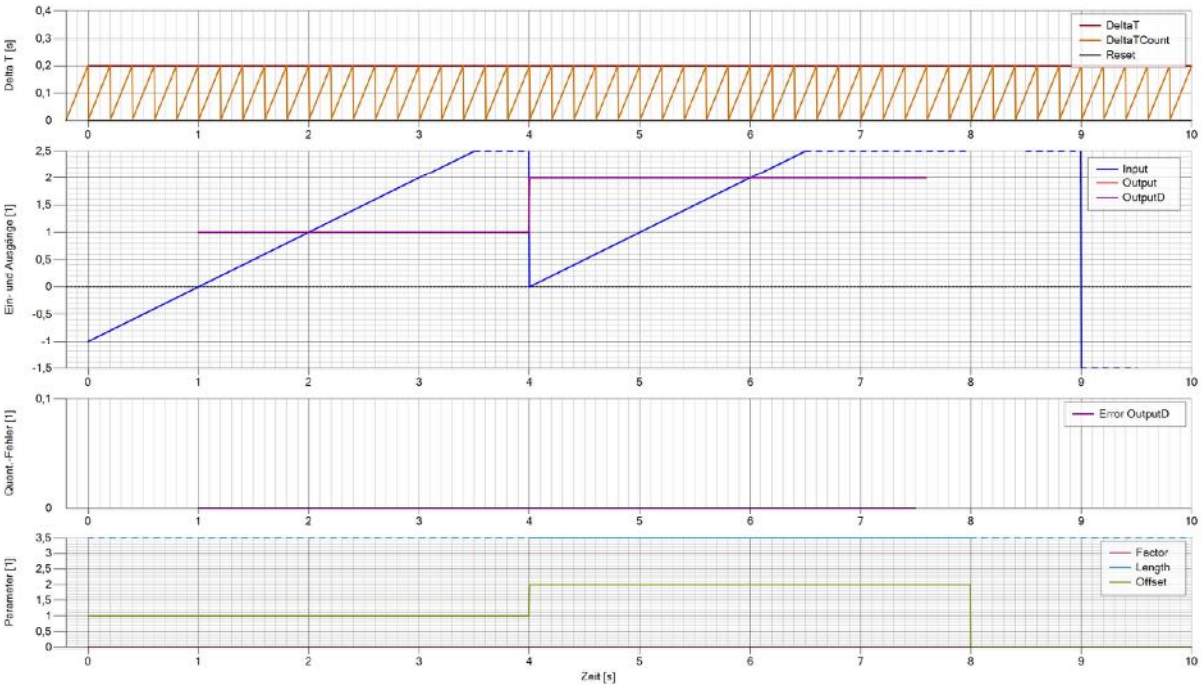


Abb. 3-52: Funktionsblock-Testergebnis: Konstante Funktion

0 - 4 Verhalten bei $Factor = 0$ $Offset = 1$ $Length = 10$

4 - 8 Verhalten bei $Factor = 0$ $Offset = 2$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

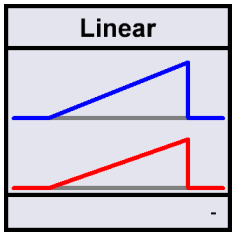
```
'#### Konstante Funktion ####
Public Class Constant
    Inherits BaseClass.FuncSegment

    Sub New()
        Factor = 0 'Standardwert überschreiben
    End Sub

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return 0
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

3.7.2 Lineare Funktion (FuncSegment.Linear)



Das Funktionssegment „Linear“ beschreibt mit seiner Ausgangsfunktion eine lineare Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \leq u(k) \leq Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = x$

Parameter: $Offset$ = Offset (Verschiebung in Y-Richtung)

$Factor$ = Verstärkung (Skalierung in Y-Richtung)

Funktionsblock-Testergebnis

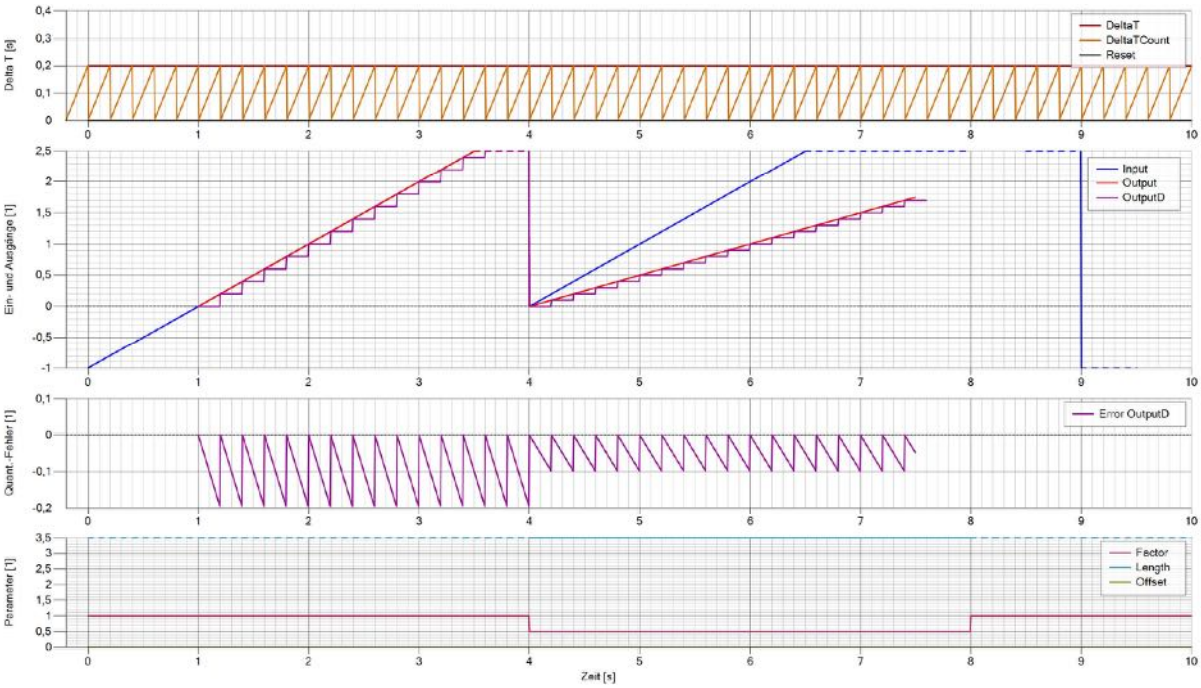


Abb. 3-53: Funktionsblock-Testergebnis: Lineare Funktion

0 - 4 Verhalten bei $Factor = 1$ $Offset = 0$ $Length = 10$

4 - 8 Verhalten bei $Factor = 0{,}5$ $Offset = 0$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

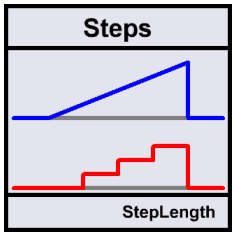
```
'#### Lineare Funktion ####
Public Class Linear
    Inherits BaseClass.FuncSegment

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return Input
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

3.7.3 Stufenfunktion (FuncSegment.Steps)



Das Funktionssegment „Steps“ beschreibt mit seiner Ausgangsfunktion eine Stufenfunktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \leq u(k) \leq Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \sum_{i=1}^{\infty} \mathcal{H}(x - i \cdot StepLength)$ $\mathcal{H}$ =Heaviside-Funktion

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $StepLength$ = Stufenlänge

Funktionsblock-Testergebnis

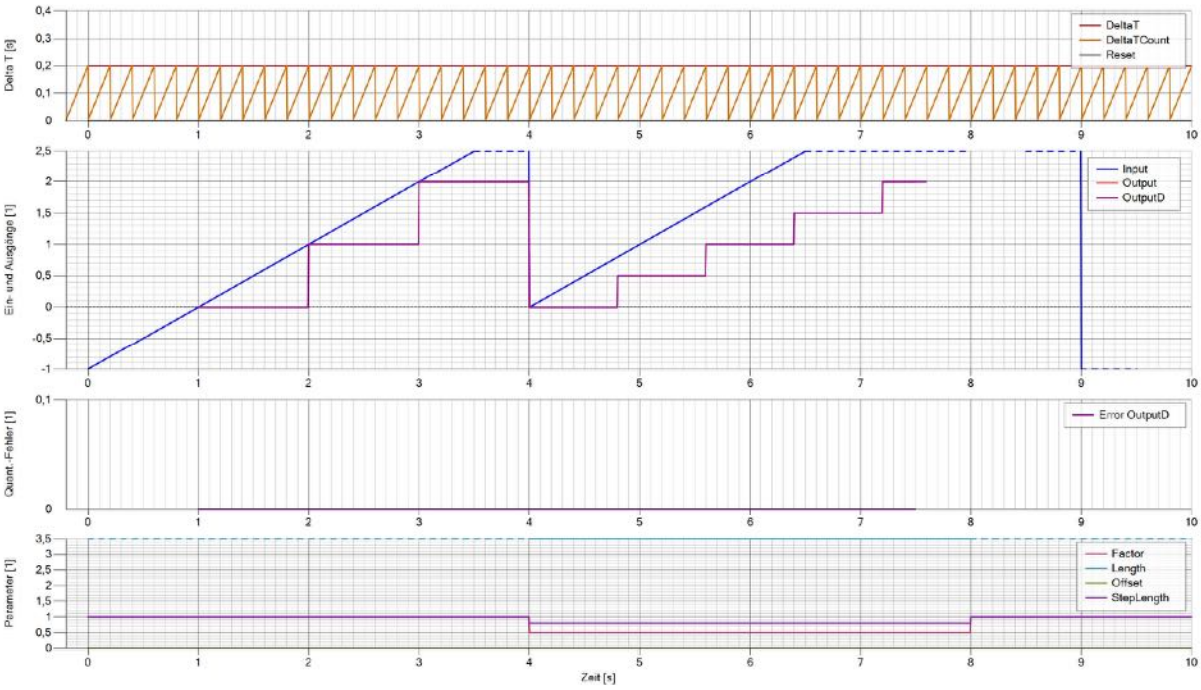


Abb. 3-54: Funktionsblock-Testergebnis: Stufenfunktion

0 - 4 Verhalten bei $Factor = 1$ $Offset = 0$ $StepLength = 1$ $Length = 10$

4 - 8 Verhalten bei $Factor = 0{,}5$ $Offset = 0$ $StepLength = 0{,}75$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

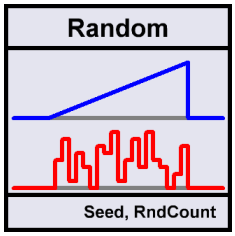
```
'#### Stufenfunktion ####
Public Class Steps
    Inherits BaseClass.FuncSegment

    Property StepLength As Double = 1 'Stufenlänge

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return Math.Truncate(Input / StepLength)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

3.7.4 Zufallsfunktion (FuncSegment.Random)



Das Funktionssegment „Random“ beschreibt mit seiner Ausgangsfunktion eine Zufallsfunktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann. Die Zufallsfunktion bildet jeden möglichen Eingangswert fest auf einen Zufallswert ab. Das bedeutet, dass Funktionsaufrufe mit dem gleichen Eingangswert den gleichen Ausgangswert liefern. Die Zufallswerte liegen dabei im Wertebereich $[0; 1]$ und sind standardmäßig gleich verteilt. Jedoch kann auch eine andere Wahrscheinlichkeitsverteilung erreicht werden, indem man mit dem Parameter „RndCount“ bestimmt, aus wie vielen gleich verteilten Zufallswerten der Mittelwert gebildet werden soll, um eine Zufallszahl zu generieren. Zudem kann mit dem Parameter „Seed“ den Eingangswerten ein anderer Satz an Zufallszahlen als Ausgangswerte zugeordnet werden.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \leq u(k) \leq Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \mathrm{Rnd}(x)$ Rnd bildet x auf die Wertemenge $[0; 1]$ ab

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Seed$ = Startwert (ordnet den Eingangswerten andere Ausgangswerte zu)
- $RndCount$ = Anzahl der Würfel (beeinflusst die Wahrscheinlichkeitsverteilung)

Funktionsblock-Testergebnis

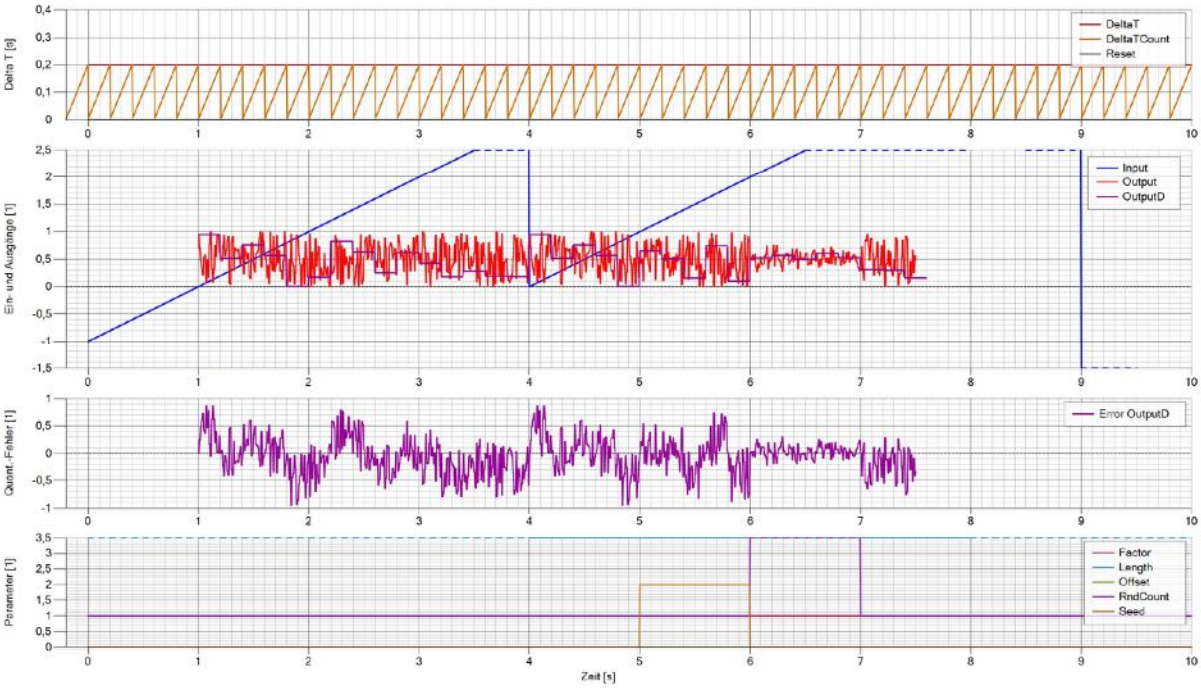


Abb. 3-55: Funktionsblock-Testergebnis: Zufallsfunktion

0 - 1 Der Definitionsbereich beginnt erst bei $Input = 0$

1 - 4 Zufallswerte bei $Factor = 1$ $Offset = 0$ $Seed = 0$ $RndCount = 1$ $Length = 10$

4 - 5 Bei gleichen Eingangswerten gleiche Zufallswerte, weil $Seed = 0$

5 - 6 Bei gleichen Eingangswerten andere Zufallswerte wegen $Seed = 1$

6 - 7 Andere Wahrscheinlichkeitsverteilung wegen $RndCount = 8$

7 - 8 Ende des Definitionsbereichs wegen $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

```
'#### Zufallsfunktion ####
Public Class Random
    Inherits BaseClass.FuncSegment

    Property Seed As Double = 0 'Startwert (ordnet den Eingangswerten andere Ausgangswerte zu)
    Property RndCount As Double = 1 'Anzahl der Würfel (beeinflusst die Wahrscheinlichkeitsverteilung)

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        If Not (RndCount >= 1 And RndCount <= Int32.MaxValue) Then
            Throw New Exception("RndCount is outside the allowed value range!")
        End If
        'Startwert des Zufallsgenerators aus Input und Seed generieren.
        Dim V As Byte() = BitConverter.GetBytes(Input) 'Bits des Double-Werts als Byte-Array darstellen
        Dim S As Byte() = BitConverter.GetBytes(Seed) 'Bits des Double-Werts als Byte-Array darstellen
        Dim IntegerSeed As Byte() = { 'Bits beider Double-Werte zusammenmischen
    CByte((CInt(V(1)) + V(4) + S(1) + S(4)) Mod 255),
    CByte((CInt(V(2)) + V(5) + S(2) + S(5)) Mod 255),
    CByte((CInt(V(3)) + V(6) + S(3) + S(6)) Mod 255),
    CByte((CInt(V(4)) + V(7) + S(4) + S(7)) Mod 255)}
        'Byte-Array in einen Integer-Wert zurück konvertieren
        Dim GeneratorSeed As Integer = BitConverter.ToInt32(IntegerSeed, 0) 'Startwert
        'Zufallsgenerator mit Startwert (abhängig von Input und Seed) erstellen
        Dim RandomGenerator As New System.Random(GeneratorSeed)
        'Den Durchschnitt mehrerer Zufallswerte berechnen, um den Verteilungsgrad zu erreichen
        Dim ValueSum As Double 'Zufallswert
        For i As UInt32 = 0 To Convert.ToUInt32(RndCount - 1) 'Alle Würfel durchlaufen
            'Zufallswert für einen Würfel errechnen [-∞; ∞]
            Dim InputByteArray As Byte() = BitConverter.GetBytes(Input)
            '8 zufällige Bytes errechnen
            For j As Integer = 0 To 7
                InputByteArray(j) = CByte(RandomGenerator.Next() Mod 255)
            Next
            '8 Bytes zu einem Integer-Wert zusammenführen
            Dim IntegerValue As UInt64 = BitConverter.ToUInt64(InputByteArray, 0)
            'Integer-Wert zu einem Double-Wert [0; 1]
            Dim Value As Double = IntegerValue / UInt64.MaxValue
            'Würfelergebnis aufaddieren
            ValueSum += Value
        Next
        Return ValueSum / RndCount 'Mittelwert aller Würfel bestimmen
    End Function

End Class
```

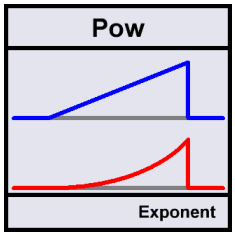
Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

Funktionsblock „Zufallsgenerator“ → Seite 111

3.7.5 Potenzfunktion (FuncSegment.Pow)



Das Funktionssegment „Pow“ beschreibt mit seiner Ausgangsfunktion eine Potenzfunktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \left(u(k)\right)^{Exponent}$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Exponent$ = Exponent der Potenzfunktion

Funktionsblock-Testergebnis

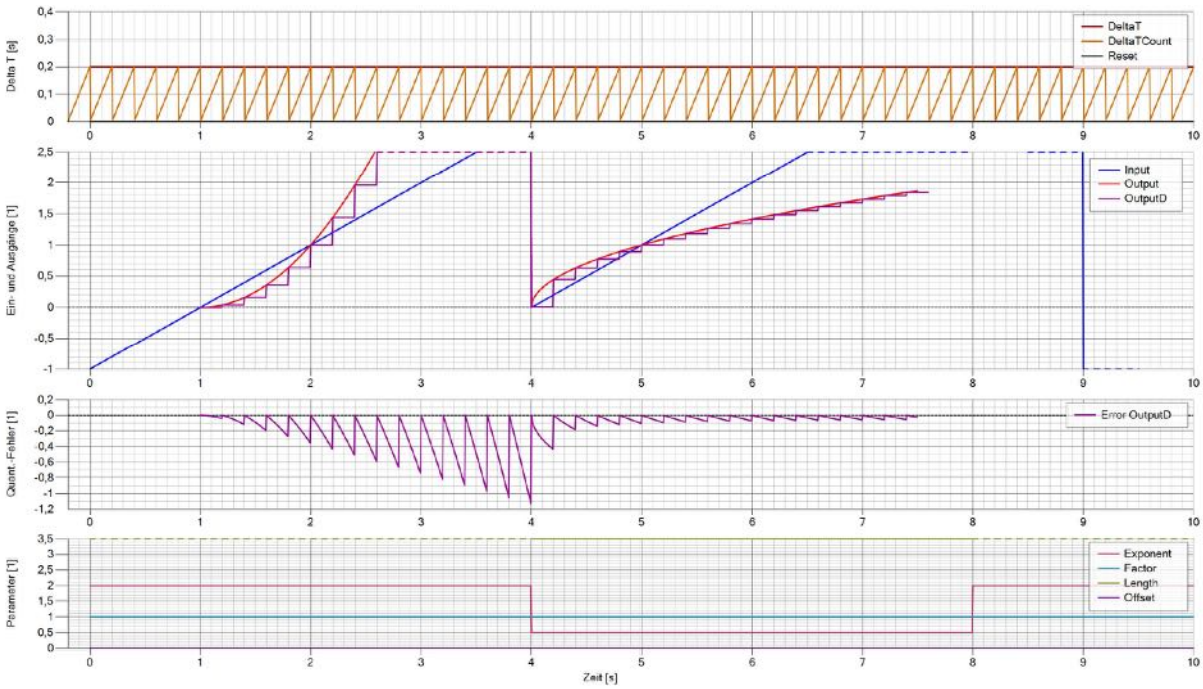


Abb. 3-56: Funktionsblock-Testergebnis: Potenzfunktion

0 - 4 Verhalten bei $Factor = 1$ $Offset = 0$ $Exponent = 2$ $Length = 10$

4 - 8 Verhalten bei $Factor = 1$ $Offset = 0$ $Exponent = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

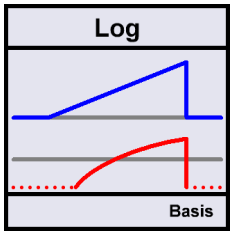
```
'#### Potenzfunktion ####
Public Class Pow
    Inherits BaseClass.FuncSegment

    Property Exponent As Double = 2 'Exponent der Potenz-Funktion

    Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0; ∞])
        Return Math.Pow(Input, Exponent)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

3.7.6 Logarithmische Funktion (FuncSegment.Log)



Das Funktionssegment „Log“ beschreibt mit seiner Ausgangsfunktion eine logarithmische Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(x) & \text{für } 0 \leq u(k) \leq Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \log_{Basis} u(k)$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Basis$ = Basis der logarithmischen Funktion

Funktionsblock-Testergebnis

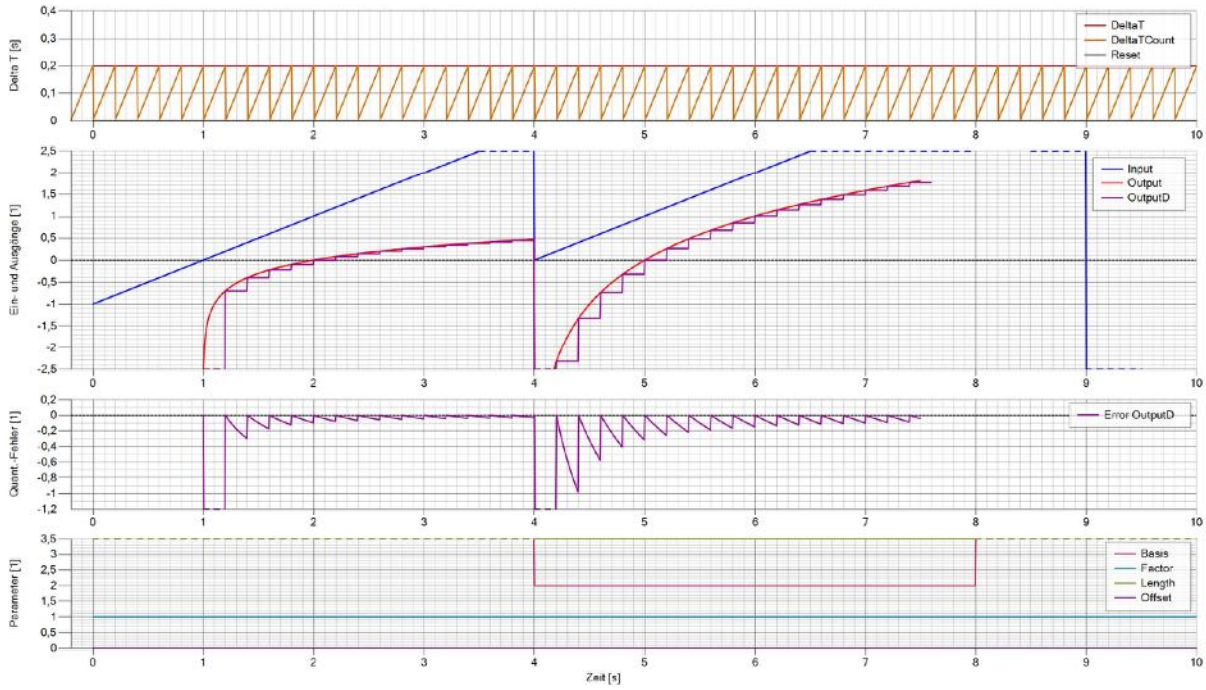


Abb. 3-57: Funktionsblock-Testergebnis: Logarithmische Funktion

0 - 4 Verhalten bei $Factor = 1$ $Offset = 0$ $Basis = 10$ $Length = 10$

4 - 8 Verhalten bei $Factor = 1$ $Offset = 0$ $Basis = 2$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

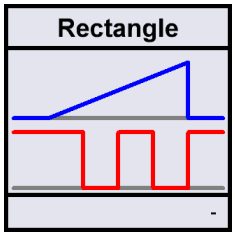
```
'#### Logarithmische Funktion ####
Public Class Log
    Inherits BaseClass.FuncSegment

    Property Basis As Double = 10 'Basis der logarithmischen Funktion

    Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0; ∞])
        Return Math.Log(Input) / Math.Log(Basis)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Funktionssegment“ → Seite 34

3.7.7 Rechteck-Funktion (FuncSegment.Rectangle)



Das Funktionssegment „Rectangle“ beschreibt mit seiner Ausgangsfunktion eine periodische Rechteck-Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot \\ f(\mathrm{Mod}\,2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \begin{cases} 1 & \text{für } x < 0{,}5 \\ 0 & \text{für } x \ge 0{,}5 \end{cases} \quad x \in [0; 1[$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

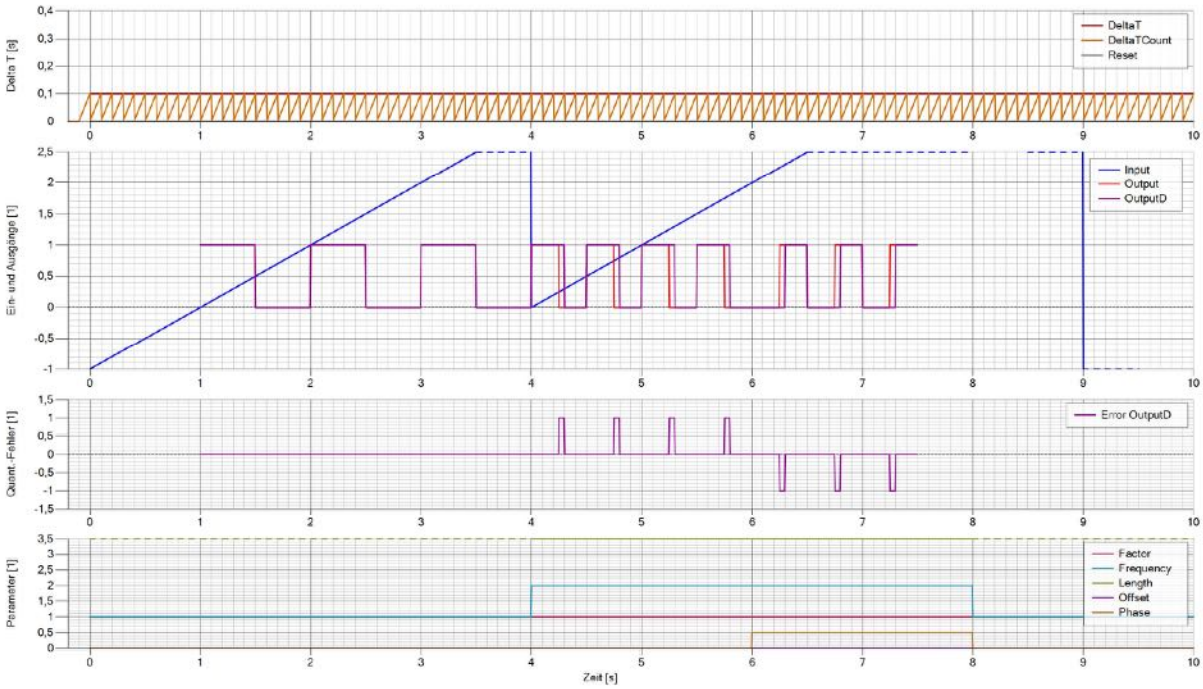


Abb. 3-58: Funktionsblock-Testergebnis: Rechteck-Funktion

0 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$ $Length = 10$

4 - 6 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0$ $Length = 3{,}5$

6 - 8 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

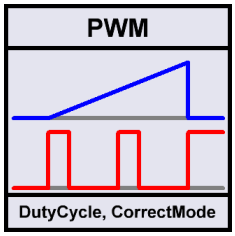
```
'#### Rechteck-Funktion ####
Public Class Rectangle

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return If(Input < 0.5, 1, 0)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.7.8 PWM-Funktion (FuncSegment.PWM)



Das Funktionssegment „PWM“ beschreibt mit seiner Ausgangsfunktion eine pulsweitenmodulierte Rechteckfunktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(\mathrm{Mod}\,2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \begin{cases} 1 & \text{für } x < DutyCycle \\ 0 & \text{für } x \ge DutyCycle \end{cases} \quad x \in [0; 1[$

Parameter:

$Offset$ = Offset (Verschiebung in Y-Richtung)

$Factor$ = Verstärkung (Skalierung in Y-Richtung)

$Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)

$Frequency$ = Frequenz (Skalierung in X-Richtung)

$DutyCycle$ = Tastgrad (Verhältnis Impulsdauer zu Periodendauer)

Funktionsblock-Testergebnis

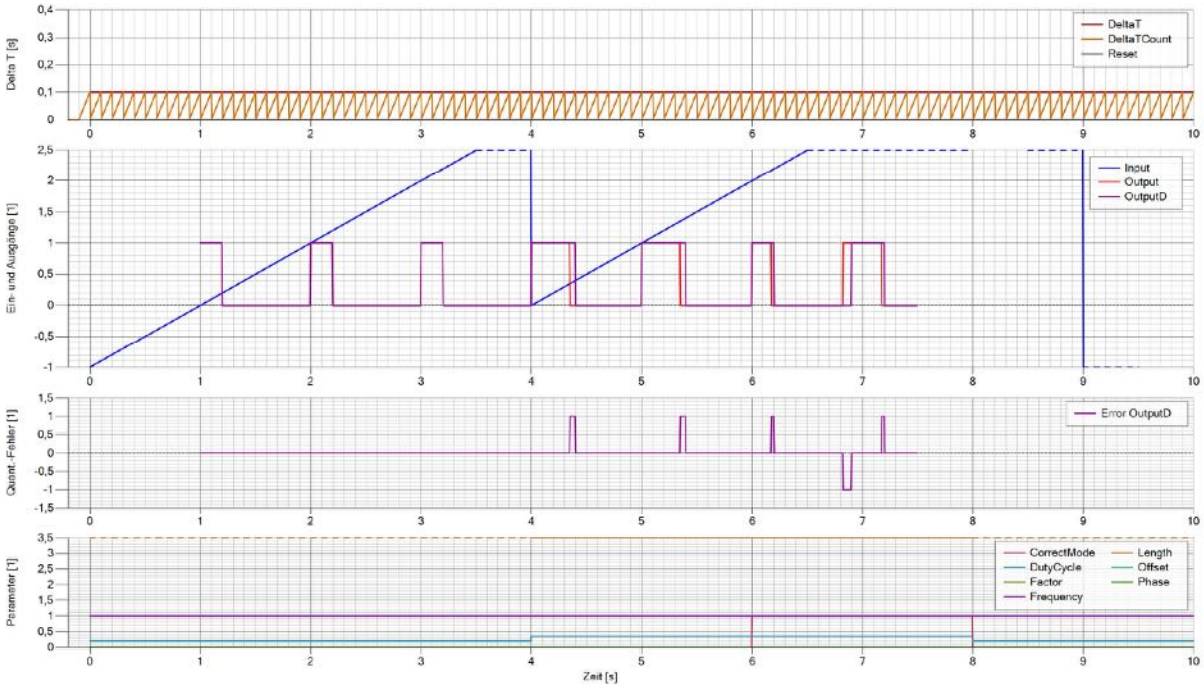


Abb. 3-59: Funktionsblock-Testergebnis: PWM-Funktion

0 - 4 Verhalten bei $DutyCycle = 0{,}2$

4 - 6 Verhalten bei $DutyCycle = 0{,}4$ (Tastgrad erhöht)

6 - 8 Verhalten bei $CorrectMode = 1$ (frequenz- und phasenrichtiger Modus aktiviert)

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Hinweise

Bei der Berechnung des Ausgangssignals wird das Eingangssignal des Typs „Double“ vorübergehend auf die Genauigkeit „Single“ gerundet, um numerische Fehler zu unterdrücken.

Frequenz- und phasenrichtige PWM

Die frequenz- und phasenrichtige Pulsweitenmodulation kann über den öffentlichen Parameter *CorrectMode* mit dem booleschen Wert „True“ aktiviert werden.

Programmcode

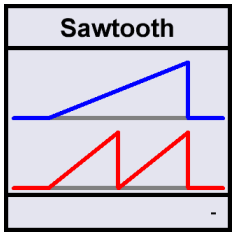
```
'#### PWM-Funktion ####
Public Class PWM
    Inherits BaseClass.FuncSegmentPeriodic 'Vererbung

    Property DutyCycle As Double = 0.2 'Tastgrad (Verhältnis Impulsdauer zu Periodendauer)
    Property CorrectMode As Boolean = False 'frequenz- und phasenrichtig

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        If CorrectMode Then 'Frequenz- und phasenrichtige Ausgabe?
            Return If((Convert.ToSingle(Input) < Convert.ToSingle(DutyCycle / 2)) Or
                      (Convert.ToSingle(Input) >= Convert.ToSingle(1 - DutyCycle / 2)), 1, 0)
        Else
            Return If(Input < DutyCycle, 1, 0)
        End If
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.7.9 Sägezahn-Funktion (FuncSegment.Sawtooth)



Das Funktionssegment „Sawtooth“ beschreibt mit seiner Ausgangsfunktion eine periodische Sägezahn-Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot f(\mathrm{Mod}\, 2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = x \qquad x \in [0; 1[$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

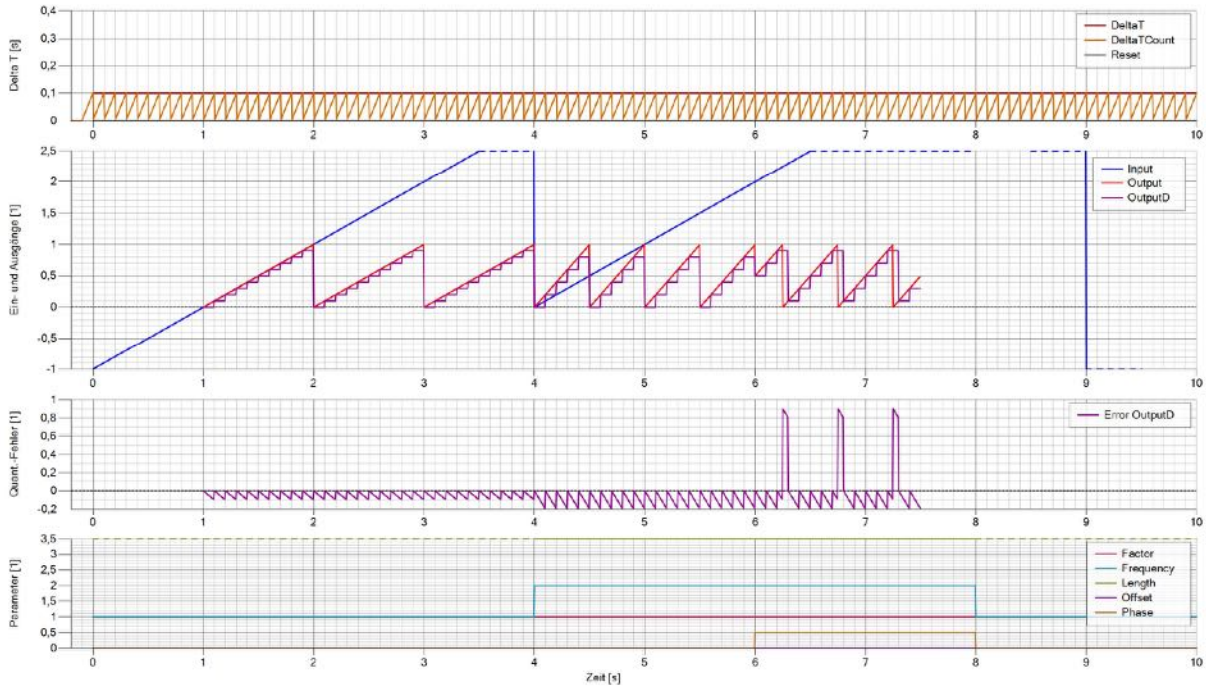


Abb. 3-60: Funktionsblock-Testergebnis: Sägezahn-Funktion

0 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$ $Length = 10$

4 - 6 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0$ $Length = 3{,}5$

6 - 8 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

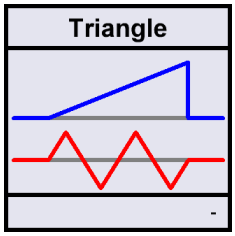
```
'#### Sägezahn-Funktion ####
Public Class Sawtooth

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return Input
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.7.10 Dreieck-Funktion (FuncSegment.Triangle)



Das Funktionssegment „Triangle“ beschreibt mit seiner Ausgangsfunktion eine periodische Dreieck-Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot \\ f(\mathrm{Mod}\,2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \begin{cases} 4 \cdot x & \text{für } \frac{0}{4} \le x < \frac{1}{4} \\ -4 \cdot x + 2 & \text{für } \frac{1}{4} \le x < \frac{3}{4} \\ 4 \cdot x - 4 & \text{für } \frac{3}{4} \le x < \frac{4}{4} \end{cases} \quad x \in [0; 1[$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

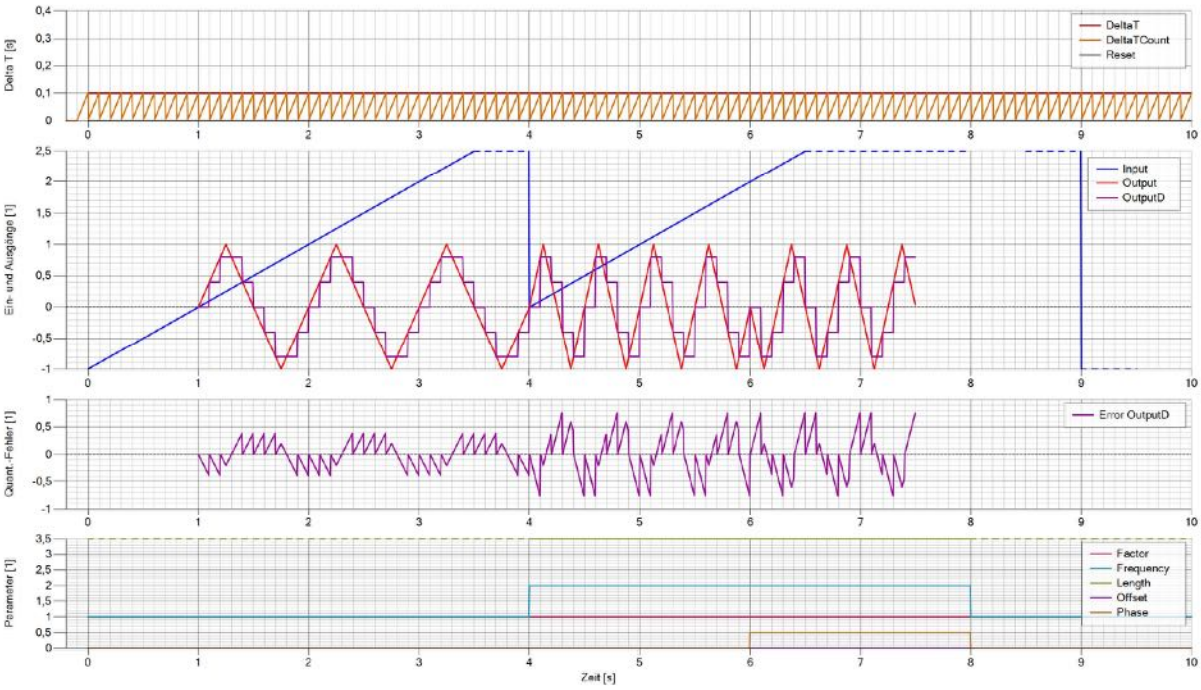


Abb. 3-61: Funktionsblock-Testergebnis: Dreieck-Funktion

0 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$ $Length = 10$

4 - 6 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0$ $Length = 3{,}5$

6 - 8 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

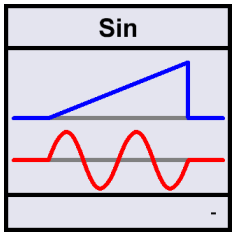
```
'#### Dreieck-Funktion ####
Public Class Triangle

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return 4 * If(Input >= 1 / 4 And Input < 3 / 4, -Input + 2 / 4, Input) _
                 + If(Input >= 3 / 4, -4, 0)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.7.11 Sinus-Funktion (FuncSegment.Sin)



Das Funktionssegment „Sin“ beschreibt mit seiner Ausgangsfunktion eine periodische Sinus-Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot \\ f(\mathrm{Mod}\, 2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \sin(x) \quad x \in [0; 1[$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

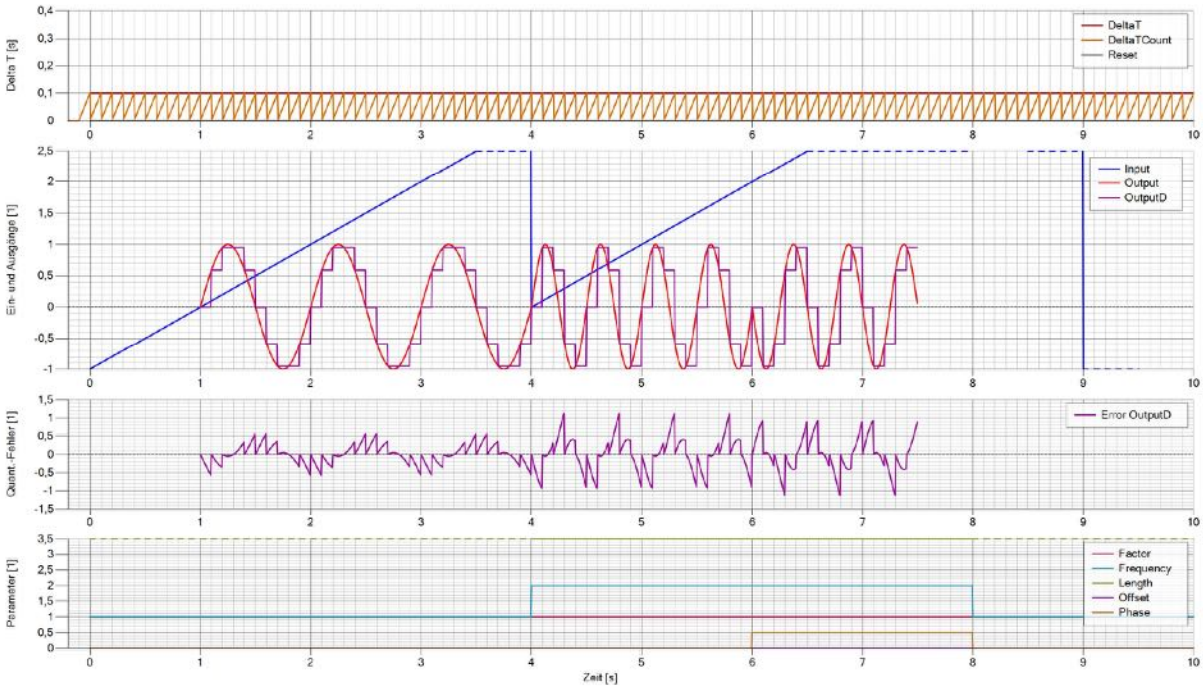


Abb. 3-62: Funktionsblock-Testergebnis: Sinus-Funktion

0 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$ $Length = 10$

4 - 6 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0$ $Length = 3{,}5$

6 - 8 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

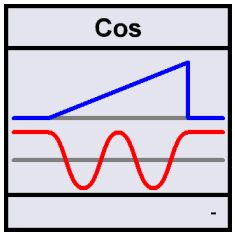
```
'#### Sinus-Funktion ####
Public Class Sin

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return System.Math.Sin(Input * 2 * System.Math.PI)
    End Function

End Class
```

Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.7.12 Cosinus-Funktion (FuncSegment.Cos)



Das Funktionssegment „Cos“ beschreibt mit seiner Ausgangsfunktion eine periodische Cosinus-Funktion in einem Definitionsbereich, der über den Parameter „Length“ mit $[0; Length]$ festgelegt werden kann.

Eigenschaften

Ausgangsfunktion: $y(k) = \begin{cases} Offset + Factor \cdot \\ f(\mathrm{Mod}\, 2(Freq. \cdot u(k) + Ph.)) & \text{für } 0 \le u(k) \le Length \\ NaN & sonst \end{cases}$

Funktionsform: $f(x) = \cos(x) \quad x \in [0; 1[$

Parameter:
- $Offset$ = Offset (Verschiebung in Y-Richtung)
- $Factor$ = Verstärkung (Skalierung in Y-Richtung)
- $Phase$ = Phase ($1 \mathrel{\widehat{=}} 360°$) (Verschiebung in X-Richtung)
- $Frequency$ = Frequenz (Skalierung in X-Richtung)

Funktionsblock-Testergebnis

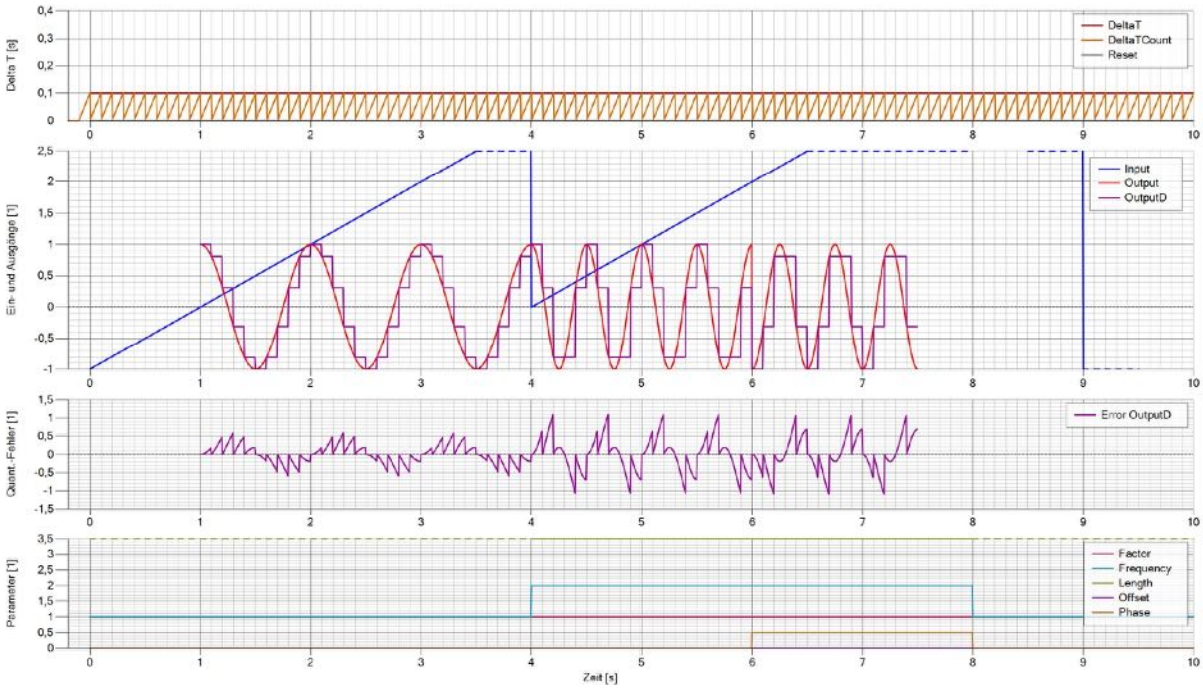


Abb. 3-63: Funktionsblock-Testergebnis: Cosinus-Funktion

0 - 4 Verhalten bei $Factor = 1$ $Frequency = 1$ $Offset = 0$ $Phase = 0$ $Length = 10$

4 - 6 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0$ $Length = 3{,}5$

6 - 8 Verhalten bei $Factor = 1$ $Frequency = 2$ $Offset = 0$ $Phase = 0{,}5$ $Length = 3{,}5$

8 - 10 Reaktion auf $Input = NaN$ und $Input = \pm\infty$

Programmcode

```
'#### Cosinus-Funktion ####
Public Class Cos

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return System.Math.Cos(Input * 2 * System.Math.PI)
    End Function

End Class
```

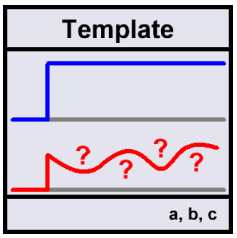
Verweise

Namensbereich „Funktions-Segmente“ → Seite 133

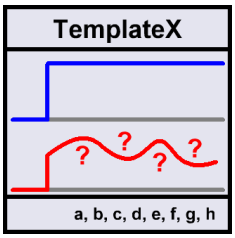
Basisklasse „Periodisches Funktionssegment“ → Seite 35

3.8 Kompilierung zur Laufzeit (JustInTime)

Die Funktionsblöcke des Namensbereichs „JustInTime“ dienen als Funktionsblock-Vorlage für die in der Testumgebung integrierte Entwicklungsumgebung und sind nicht für das Einbinden in weitere Projekte vorgesehen. Die Vorlage „Template“ ermöglicht dem Entwickler das Erstellen eines Funktionsblocks, der über einen Approximations-Typ und drei Parameter verfügt. Die zweite Vorlage „TemplateX“ stellt hingegen fünf Approximations-Typen und acht Parameter zur Verfügung. Eine Anleitung zum Erstellen von Funktionsblöcken mit der Testumgebung ist auf Seite 183 zu finden.



→ Seite 161

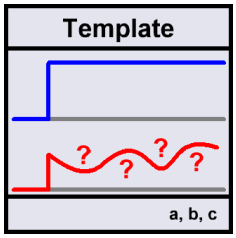


→ Seite 163

Verweise

Basisklasse „Zeitabhängiger Funktionsblock“	→ Seite 33
Aufbau der Bibliothek	→ Seite 29
Implementierung eines Funktionsblocks	→ Seite 36
Hinweise zur Implementierung	→ Seite 38
Integrierte Entwicklungsumgebung	→ Seite 183

3.8.1 Funktionsblock-Vorlage (JustInTime.Template)



Der Funktionsblock „Template“ dient als Funktionsblock-Vorlage für die in der Testumgebung integrierte Entwicklungsumgebung und ist nicht für das Einbinden in weitere Projekte vorgesehen. Mit dieser Vorlage kann der Entwickler Funktionsblöcke erstellen, die über einen Approximations-Typ und drei Parameter verfügen. Eine Anleitung zum Erstellen von Funktionsblöcken mit der Testumgebung ist auf Seite 183 zu finden.

Eigenschaften

Approximations-Typen: *Approx*

Parameter: a, b, c

Funktionsblock-Testergebnis

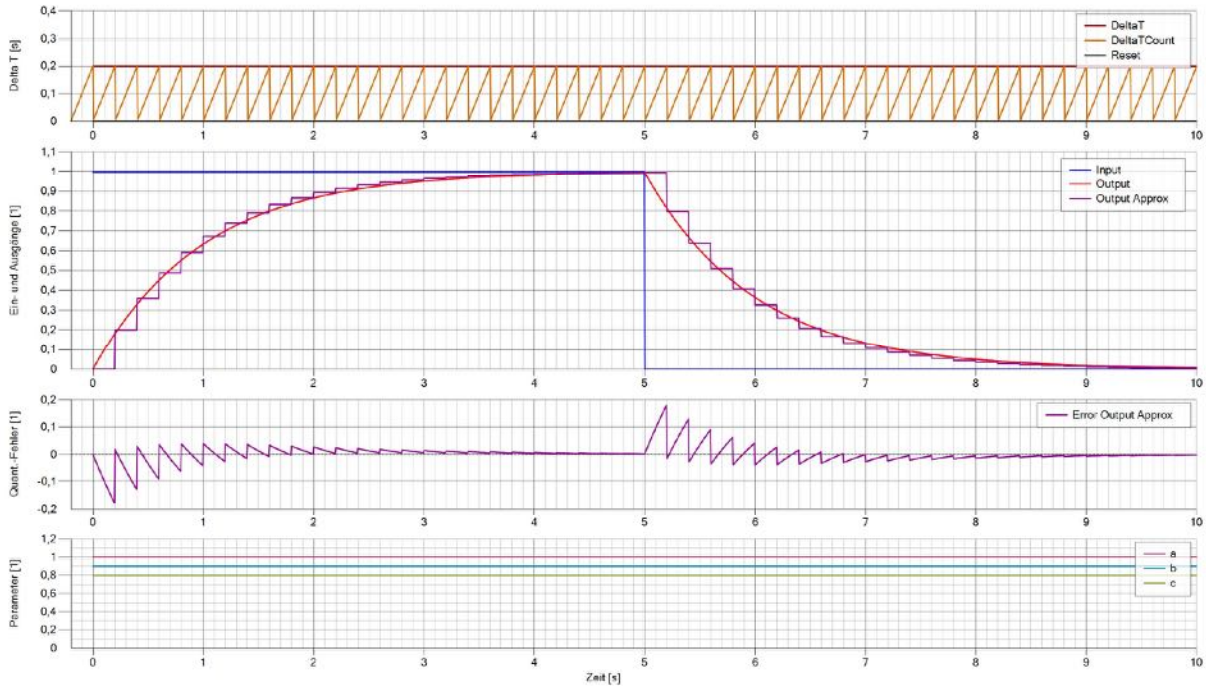


Abb. 3-64: Funktionsblock-Testergebnis: Funktionsblock-Vorlage

0 - 10 Beispiel-Funktionalität, die mit der Funktionsblock-Vorlage erstellt wurde.

Programmcode

```
'#### Funktionsblock-Vorlage ####
Public Class Template
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx
    Public Enum ApproxTypeEnum
        Approx
    End Enum

    Public Property a As Double = 1
    Public Property b As Double = 1
    Public Property c As Double = 1
```

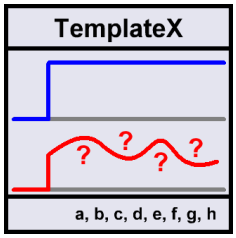
```
    Protected Overrides Function Functionality() As Double
        Return Double.NaN
    End Function

    Public Shared Function GetDefaultVbNetCode() As String
        Return <![CDATA[
    'Funktionsblock-Vorlage, die zur Laufzeit der Testumgebung kompiliert wird.
    Class Template
    'Durch Vererbung stehen folgende Parameter und Eigenschaften zur Verfügung:
    'Eingangswerte:             u(k-0), u(k-1), u(k-2)
    'Ausgangswerte:             y(k-0), y(k-1), y(k-2)
    'DeltaT:                    T
    'Approximations-Typ:ApproxType
    Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

    'Approximationstyp dieses Templates
    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx
    Public Enum ApproxTypeEnum
        Approx
    End Enum

    'Parameter dieses Templates
    Public Property a As Double
    Public Property b As Double
    Public Property c As Double

    'Konstruktor
    Sub New()
    End Sub

    'Funktion, zum Berechnen der Ausgangswerte.
    Protected Overrides Function Functionality() As Double
        'Je nach gewähltem Approximations-Typ werden die dazugehörigen Gleichungen aufgerufen.
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.Approx 'T1-Glied approximiert nach "Euler Vorwärts"
                Return T / a * (u(k - 1) - y(k - 1)) + y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

    End Class]]>.Value()
    End Function

End Class
```

Verweise

Namensbereich „Kompilierung zur Laufzeit“ → Seite 159

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

3.8.2 Erweiterte Funktionsblock-Vorlage (JustInTime.TemplateX)



Der Funktionsblock „TemplateX“ dient als Funktionsblock-Vorlage für die in der Testumgebung integrierte Entwicklungsumgebung und ist nicht für das Einbinden in weitere Projekte vorgesehen. Mit dieser Vorlage kann der Entwickler Funktionsblöcke erstellen, die über fünf Approximations-Typen und acht Parameter verfügen. Eine Anleitung zum Erstellen von Funktionsblöcken mit der Testumgebung ist auf Seite 183 zu finden.

Eigenschaften

Approximations-Typen: $Approx0$, $Approx1$, $Approx2$, $Approx3$, $Approx4$

Parameter: a, b, c, d, e, f, g, h

Funktionsblock-Testergebnis

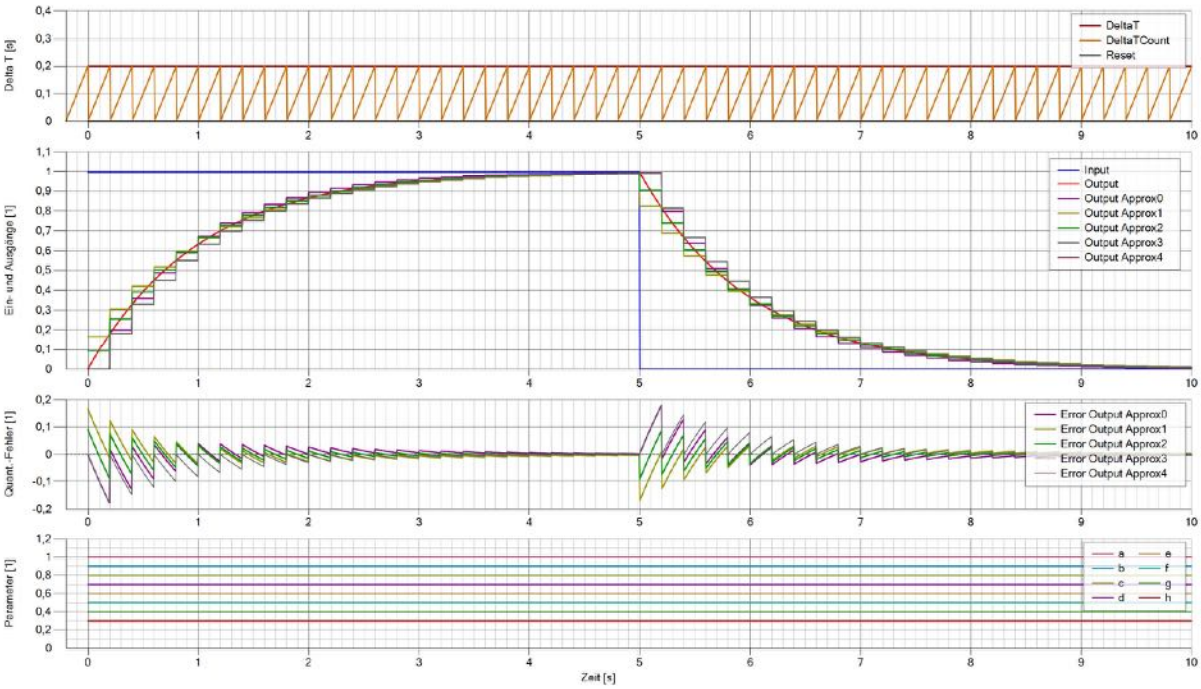


Abb. 3-65: Funktionsblock-Testergebnis: Erweiterte Funktionsblock-Vorlage

0 - 10 Beispiel-Funktionalität, die mit der Funktionsblock-Vorlage erstellt wurde.

Programmcode

```
'#### Erweiterte Funktionsblock-Vorlage ####
Public Class TemplateX
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx0
    Public Enum ApproxTypeEnum
        Approx0
        Approx1
        Approx2
        Approx3
        Approx4
    End Enum
```

```
    Public Property a As Double = 1
    Public Property b As Double = 1
    Public Property c As Double = 1
    Public Property d As Double = 1
    Public Property e As Double = 1
    Public Property f As Double = 1
    Public Property g As Double = 1
    Public Property h As Double = 1

    Protected Overrides Function Functionality() As Double
        Return Double.NaN
    End Function

    Public Shared Function GetDefaultVbNetCode() As String
        Return <![CDATA[
    'Funktionsblock-Vorlage, die zur Laufzeit der Testumgebung kompiliert wird.
    Class TemplateX
        'Durch Vererbung stehen folgende Parameter und Eigenschaften zur Verfügung:
        'Eingangswerte:         u(k-0), u(k-1), u(k-2)
        'Ausgangswerte:         y(k-0), y(k-1), y(k-2)
        'DeltaT:                T
        'Approximations-Typ:    ApproxType
        Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

        'Approximationstyp dieses Templates
        Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx0
        Public Enum ApproxTypeEnum
            Approx0
            Approx1
            Approx2
            Approx3
            Approx4
        End Enum

        'Parameter dieses Templates
        Public Property a As Double = 1
        Public Property b As Double = 1
        Public Property c As Double = 1
        Public Property d As Double = 1
        Public Property e As Double = 1
        Public Property f As Double = 1
        Public Property g As Double = 1
        Public Property h As Double = 1

        'Funktion, zum Berechnen der Ausgangswerte.
        Protected Overrides Function Functionality() As Double
            'Je nach gewähltem Approximations-Typ werden die dazugehörigen Gleichungen aufgerufen.
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.Approx0 'T1-Glied approximiert nach "Euler Vorwärts"
                    Return T / a * (u(k - 1) - y(k - 1)) + y(k - 1)
                Case ApproxTypeEnum.Approx1 'T1-Glied approximiert nach "Euler Rückwärts"
                    Return 1 / (a + T) * (a * y(k - 1) + T * u(k))
                Case ApproxTypeEnum.Approx2 'T1-Glied approximiert nach "Tustin"
                    Return 1 / (2 * a + T) * ((2 * a - T) * y(k - 1) + T * u(k - 1) + T * u(k))
                Case ApproxTypeEnum.Approx3 'T1-Glied approximiert nach Pol-Nullstellen-Abbildung
                    Return (1 - Math.Exp(-1 / a * T)) * u(k - 1) + Math.Exp(-1 / a * T) * y(k - 1)
                Case ApproxTypeEnum.Approx4
                    Return Double.NaN
                Case Else
                    Return Double.NaN
            End Select
        End Function

    End Class
    ]]>.Value()
    End Function

End Class
```

Verweise

Namensbereich „Kompilierung zur Laufzeit“ → Seite 159

Basisklasse „Zeitabhängiger Funktionsblock“ → Seite 33

4 Testumgebung

Die Funktionsblock-Testumgebung wurde im Rahmen der Masterarbeit in VB.NET umgesetzt, um die Bausteine der Automatisierungsbibliothek schon während der Entwicklungsphase komfortabel testen zu können. Für Entwickler, die die Bibliothek in Zukunft einsetzen werden, stellt die Testumgebung umfangreiche Werkzeuge zum Erweitern der Bibliothek zur Verfügung. Zudem helfen die Schaubilder der Tests, die Funktionsweise der Funktionsbausteine schneller zu verstehen und verschiedene Approximationstypen miteinander zu vergleichen.

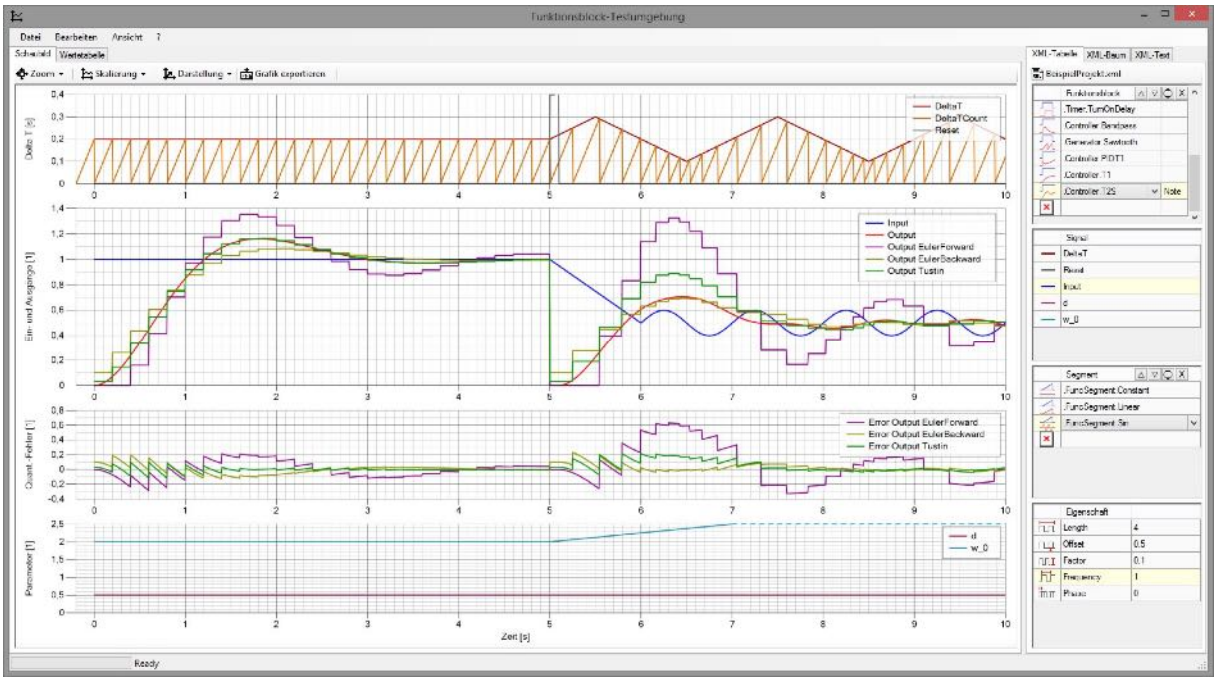


Abb. 4-1: Benutzeroberfläche der Testumgebung

4.1 Funktionsweise

Um einen Funktionsbaustein der Automatisierungsbibliothek zu testen, erstellt der Entwickler mit der Umgebung einen Funktionsblock-Test. Hierbei müssen für alle Eingänge und Parameter des Funktionsbausteins Testsignale durch das Aneinanderreihen von mathematischen Funktionen erstellt werden. Beim Ausführen des Funktionsblock-Tests ruft die Testumgebung dann den Funktionsblock mit den zuvor definierten Signalen auf und legt das Ausgangssignal zusammen mit den Testsignalen als Testergebnis ab. Das Ergebnis des Tests wird anschließend mit Diagrammen und Wertetabellen in der Benutzeroberfläche grafisch dargestellt und kann für die weitere Verarbeitung exportiert werden.

In den folgenden Abschnitten wird die Funktionsweise der Programmelemente kurz erklärt und es werden einige Funktionen des Programms vorgestellt. Anschließend folgt eine ausführliche Anleitung zur Testumgebung.

Funktionsblock-Test

Mit einem Funktionsblock-Test wird definiert, wie ein Funktionsbaustein der Bibliothek getestet werden soll. Dabei werden nicht nur das Eingangssignal und die Parameter des Funktionsblocks in Abhängigkeit der Zeit festgelegt, sondern auch bestimmt, zu welchen Zeitpunkten der Funktionsbaustein aufgerufen und zurückgesetzt werden soll. All dies geschieht durch das abschnittsweise Definieren von Testsignalen mit mathematischen Funktionen. Diese repräsentieren zum Beispiel für das Eingangssignal und die Parameter zeitabhängig Werte oder können durch ihren Signalverlauf Methoden- und Funktionsaufrufe zur Testlaufzeit auslösen.

Testlaufzeit

Zur Laufzeit eines Funktionsblock-Tests durchläuft das Programm die Zeitachse des Tests schrittweise in einem Zyklus. Bei jedem Durchlauf werden die von der Simulationszeit abhängigen Parameter in den zu testenden Funktionsbaustein geladen, die Ausgangsfunktion mit Übergabe des aktuellen Eingangswerts aufgerufen und der Rückgabewert als Testergebnis abgelegt. Dieser Zyklus wird für alle Approximations-Typen (Euler, Tustin, usw.), die der zu testende Funktionsbaustein unterstützt, einzeln ausgeführt. Als Testergebnis liegen danach alle am Funktionsblock-Test beteiligten Signale zeitlich quantisiert als Wertereihen vor.

Definieren der Testsignale

Der zeitliche Verlauf des Funktionsblock-Tests wird durch die benutzerdefinierten Testsignale festgelegt. Zum Erstellen eines solchen Signals stehen dem Entwickler einige Funktionsblöcke aus der Bibliothek zur Verfügung, die er aneinanderreihen kann, um Funktionen für komplexe Signale abschnittsweise zu definieren. Somit kann zum Beispiel während eines Funktionsblock-Tests der Wert des Eingangssignals so festgelegt werden, dass er zuerst konstant ist, danach linear abfällt und anschließend einer Sinusfunktion folgt. Während der Testlaufzeit werden alle definierten Testsignale, wie oben beschrieben, zeitlich quantisiert und zusammen mit den Ausgangssignalen des Funktionsbausteins als Testergebnis abgelegt.

Darstellung des Testergebnisses

Das Ergebnis eines Funktionsblock-Tests wird über die Benutzerfläche zum einen als Wertetabelle dargestellt und des Weiteren in einem Schaubild visualisiert. Das Schaubild kann als Grafik exportiert und die Wertetabelle für die weitere Verarbeitung zum Beispiel in ein Tabellenkalkulationsprogramm kopiert werden.

Speichern und Laden

Es ist möglich, alle erstellten Funktionsblock-Tests eines Projekts als XML-Datei zu speichern. Somit können bereits erstellte Tests zu einem späteren Zeitpunkt geladen und erweitert werden.

Rückgängig machen und Wiederholen

Alle Schritte, die der Benutzer beim Erstellen von Funktionsblock-Tests in der Testumgebung ausführt, können rückgängig gemacht und bei Bedarf wiederhergestellt werden. Dabei werden sogar die zum Zeitpunkt der Änderung gewählten Objektmarkierungen geladen.

Abstrakter Umgebungsaufbau

Die Testumgebung wurde sehr abstrakt und robust ausgelegt, um eine möglichst große Kompatibilität gegenüber Erweiterungen der Automatisierungsbibliothek zu bieten. Erst zur Laufzeit greift die Umgebung auf die Struktur der Bibliothekselemente zu und erkennt Funktionsblöcke mit ihren frei definierbaren Approximations-Typen und Parametern dynamisch. Die geringe typenabhängige Bindung ermöglicht es, alte Funktionsblock-Tests für weiterentwickelte Funktionsblöcke wiederzuverwenden. Zum Beispiel werden für neue Parameter eines Funktionsbausteins die dazu benötigten Testsignale automatisch erstellt und die aus Bibliotheksbausteinen entfernten Parameter auch aus den dazugehörigen Tests gelöscht.

Integrierte Entwicklungsumgebung

Ein für die Weiterentwicklung der Bibliothek sehr nützliches Programmelement ist die integrierte Entwicklungsumgebung. Sie ermöglicht die Programmierung neuer Funktionsblöcke direkt in einem Editor der Testumgebung. Der geschriebene Programmcode wird während der Laufzeit der Umgebung nach einer Änderung des Codes sofort von einem Just-In-Time-Compiler geprüft und übersetzt. Daraufhin aktualisiert die Umgebung auch das Testergebnis des dazugehörigen Funktionsblock-Tests automatisch und gibt bei Fehlern Meldungen aus. Somit können neue Funktionsblöcke schon während der Implementierung getestet werden, was die Entwicklung neuer Bibliotheksbausteine sehr zeiteffizient macht.

Bevor Schritt für Schritt gezeigt wird, wie man mit der Testumgebung einen Funktionsblock-Test erstellt, geht das folgende Unterkapitel detaillierter auf die Testsignale ein.

4.2 Testsignale

Wie bereits erwähnt, basiert ein Funktionsblock-Test auf benutzerdefinierten Testsignalen, die zusammen mit den zur Testlaufzeit ermittelten Ausgangssignalen das Testergebnis bilden. In den folgenden Abschnitten wird anhand eines Beispiels, bei dem der „T2S“-Funktionsblock getestet wird, die Bedeutung der einzelnen Signale detailliert erklärt.

„DeltaT“, „DeltaTCount“ und „Reset“

Die ersten drei Signale steuern die zeitliche Taktung der Funktionsaufrufe. Mit dem Signal „DeltaT“ kann der Benutzer festlegen, in welchen Zeitabständen die Ausgangsfunktion aufgerufen werden soll. Zur Testlaufzeit generiert die Umgebung hierzu das zweite Signal „DeltaTCount“. Der Wert dieses Signals wird mit jedem Zyklus um die Zykluszeit erhöht und beschreibt die seit dem letzten Funktionsaufruf verstrichene Zeit. Erreicht oder überschreitet der Zeitzähler „DeltaTCount“ das vom Benutzer vorgegebene Testsignal „DeltaT“, so wird noch im gleichen Zyklus die Ausgangsfunktion des Funktionsblocks aufgerufen und im darauf folgenden das Signal „DeltaTCount“ zurückgesetzt. Somit lässt sich mit dem Testsignal „DeltaT“ die Zykluszeit in Abhängigkeit der Simulationszeit variabel festlegen.

Mit dem Signal „Reset“ kann hingegen der zu testende Funktionsblock zur Laufzeit zurückgesetzt werden. Stellt das Signal zum Zeitpunkt eines Funktionsblockaufrufs einen Wert dar, der größer als 0 ist, wird in dem gleichen Zyklus noch vor dem Aufruf der Ausgangsfunktion die Reset-Funktion aufgerufen. Der Aufruf der Reset-Funktion des Funktionsblocks setzt alle intern gespeicherten Ein- und Ausgangswerte zurück.

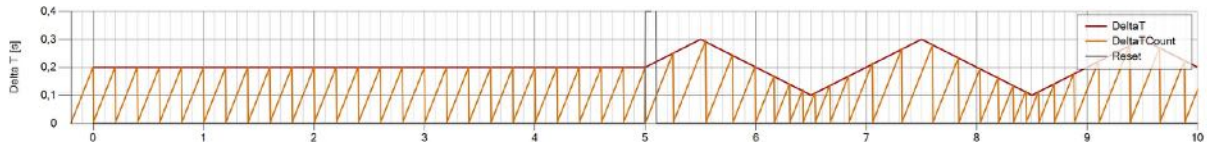


Abb. 4-2: Signale zur zeitlichen Taktung der Funktionsaufrufe

Die bisher vorgestellten Testsignale, die die zeitliche Taktung der Funktionsaufrufe steuern, werden im ersten Diagramm des Schaubilds dargestellt. Das obige Beispiel wurde einem Test entnommen, bei dem der Funktionsblock zunächst in äquidistanten Zeitabständen (im Beispiel 0,2 Sekunden) und nach halber Simulationszeit in nicht äquidistanten Zeitabständen aufgerufen wird, um eine schwankende Zykluszeit zu simulieren. Der Impuls, der im Signal „Reset“ zu finden ist, setzt dabei den Funktionsbaustein beim Erreichen der Simulationszeit von 5 Sekunden zurück.

„Input“, „Output“ und „Output <ApproximationType>“

Die Signale, die im zweiten Diagramm des Schaubildes dargestellt werden, beschreiben den zeitlichen Verlauf des Ein- und Ausgangswerts des zu testenden Funktionsblocks. Für jeden Approximations-Typ, den der Funktionsblock unterstützt (im Beispiel Euler Vorwärts, Euler Rückwärts und Tustin), wird ein eigenes Ausgangssignal erstellt, wobei das Eingangssignal „Input“ für alle Testfälle dasselbe ist.

Alle generierten Ausgangssignale beschreiben das Ausgangsverhalten des gleichen Funktionsbausteins bei unterschiedlich gewähltem Approximations-Typ und verschiedenen Zykluszeiten. Das Signal „Output“ wird mit dem Standard-Approximations-Typ des zu testenden Funktionsbausteins erstellt und kann als ideal angesehen werden, weil die Testumgebung zur Berechnung den Funktionsblock in sehr kleinen Zeitabständen von 0,05 Sekunden aufruft. Alle weiteren Ausgangssignale werden mit den vom Benutzer vorgegebenen Zykluszeiten generiert und stellen für alle unterstützten Approximation-Typen einzeln eine Näherung des idealen Ausgangssignals dar.

Während der Simulationszeit ruft die Testumgebung den Funktionsbaustein nur zu den mit dem Signal „DeltaT“ vorgegebenen Zeitpunkten auf. Deshalb sind in den Ausgangssignalen der Approximationen Stufen zu erkennen. Diese entstehen, weil sich das Signal nur zum Zeitpunkt eines Zyklus ändern kann, bei dem der Funktionsblock aufgerufen wird. Die Länge dieser Stufen entspricht demnach der zu diesem Zeitpunkt mit „DeltaT“ vorgegebenen Zykluszeit.

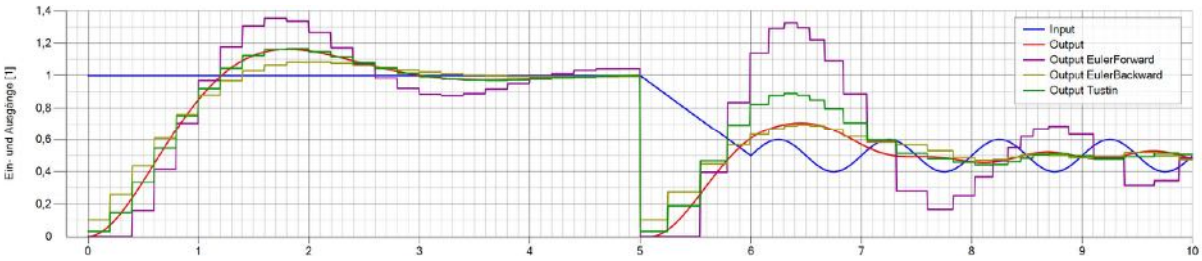


Abb. 4-3: Ein- und Ausgangssignale des „T2S“-Funktionsbausteins

Mit dem hier dargestellten Testergebnis wird das Verhalten des schwingungsfähigen „T2S“-Funktionsbausteins untersucht. Anhand des Diagramms ist beispielsweise zu erkennen, dass die Sprungantwort des Funktionsblocks eine gedämpfte Schwingung beschreibt und sich der Baustein auch noch nach dem Zurücksetzen richtig verhält. Des Weiteren kann man deutlich erkennen, dass alle Approximations-Typen des „T2S“-Bausteins unter den hier simulierten Bedingungen ein stabiles Ausgangssignal liefern; allerdings lassen sich daraus keine allgemein gültigen Stabilitätsbedingungen für den Funktionsblock ableiten.

„Error Output <ApproximationType>“

Zu jedem Approximations-Typ generiert die Testumgebung ein weiteres Signal, das die Abweichung des jeweiligen Ausgangssignals von dem idealen Ausgangssignal beschreibt. Diese durch die zeitliche Quantisierung entstandene Abweichung, nennt man Quantisierungsfehler. Das dritte Diagramm des Schaubildes stellt für jeden Approximations-Typ diesen Fehler in Abhängigkeit der Simulationszeit dar.

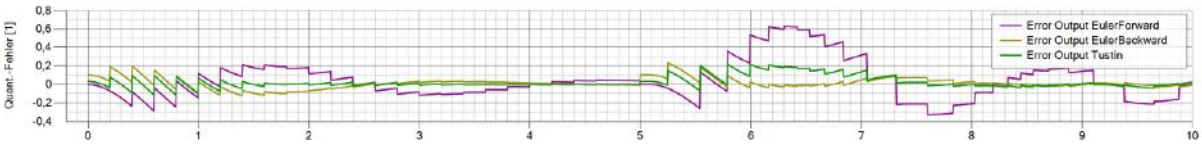


Abb. 4-4: Quantisierungsfehler der Ausgangssignale

Anhand des Diagramms lässt sich gut erkennen, dass der Funktionsblock „T2S“ während der Sprungantwort mit der Approximationsform nach „Tustin“ den kleinsten durchschnittlichen Approximationsfehler aufweist. Während der zweiten Hälfte der Simulationszeit, bei der die Zykluszeit schwankt, liefert hingegen die Approximation nach „Euler Rückwärts“ das bessere Ergebnis. Auch diese Aussagen sind in Hinblick auf diesen einen Funktionsblock-Test richtig. Dennoch sollte man dabei vorsichtig sein, daraus allgemeingültige Schlüsse zu ziehen.

Parameter

Die Testsignale der Parameter bestimmen die Werte, die vor jedem Funktionsblockaufruf in die öffentlichen Eigenschaften des Funktionsblocks geladen werden. Für den Test können die Parameter mittels Testsignalen von der Simulationslaufzeit abhängig vorgegeben werden.

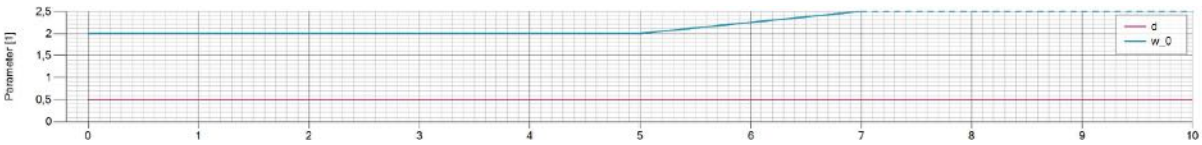


Abb. 4-5: Parametersignale

In diesem Beispiel verfügt der Funktionsblock „T2S“ über zwei Parameter, von denen auch seine Übertragungsfunktion $G(s)$ abhängig ist.

$$G(s) = \frac{1}{\frac{1}{\omega_0^2} \cdot s^2 + \frac{2 \cdot d}{\omega_0} \cdot s + 1}$$

Zum einen ist dies die Eigenkreisfrequenz ω_0 und zum anderen die Dämpfungskonstante d. Die dazugehörigen Testsignale wurden so definiert, dass die Dämpfung über die komplette Simulationszeit mit dem Wert 0,5 konstant bleibt, wobei die Eigenkreisfrequenz aber ab halber Simulationszeit kontinuierlich erhöht wird.

Die folgenden Unterkapitel erklären nun, wie man mit der Testumgebung Schritt für Schritt einen Funktionsblock-Test erstellt und ausführt.

4.3 Erstellen eines Funktionsblock-Tests

Das Erstellen eines Funktionsblocktests erfolgt über die Tabellenansichten, die sich im rechten Bereich der Benutzeroberfläche untereinander angeordnet befinden, aber hier nebeneinander dargestellt sind. Das Markieren eines Tabellenelements hat zur Folge, dass in der darauf folgenden Tabelle dessen Inhalt angezeigt wird.

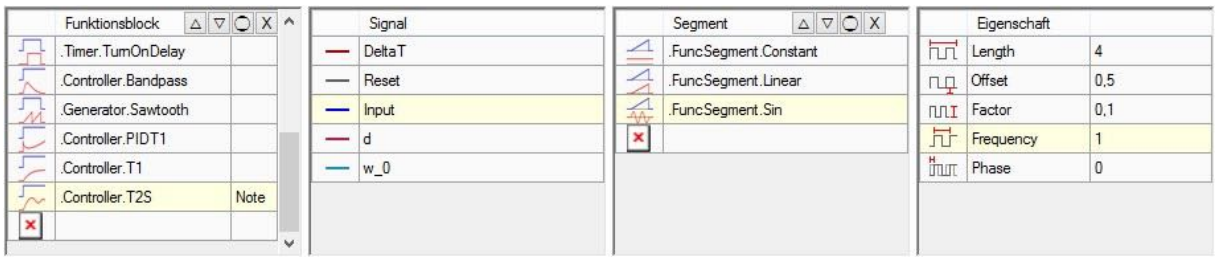


Abb. 4-6: Tabellenansichten zum Bearbeiten von Funktionsblock-Tests

Erstellen eines Testumgebungs-Projekts

Bevor mit der Testumgebung Funktionsblock-Tests erstellt werden können, muss ein Projekt angelegt werden. Die Anwendung lädt beim Starten - wenn möglich - automatisch das zuletzt geöffnete Projekt. Ansonsten wird ein leeres Projekt erstellt und geöffnet. Möchte man während einer Session manuell ein neues Projekt anlegen, kann man dies jederzeit über das Dropdown-Listenfeld „Datei“ tun.

Abb. 4-7: Dropdown-Listenfeld „Datei“

Über das gleiche Listenfeld ist es ebenfalls möglich, eine Projektdatei zu laden und als XML-Datei zu speichern oder das Programm zu beenden. Wurden während einer Session Änderungen am Projekt vorgenommen, fragt die Umgebung beim Beenden, ob diese gespeichert werden sollen. Hierbei ist zu beachten, dass das Auswählen eines anderen Elements einer Tabelle bereits eine Änderung darstellt, weil die zuletzt gesetzten Markierungen mit abgespeichert werden.

Der Dateiname des aktuellen Testumgebungs-Projekts wird oberhalb der Tabellenansichten im rechten Bereich der Testumgebung angezeigt.

Abb. 4-8: Angezeigter Name der Projektdatei „BeispielProjekt.xml“

Der Inhalt der XML-Datei kann über die Registerkarten im rechten Bereich der Benutzeroberfläche in drei verschiedenen Darstellungen als Tabelle, Baumstruktur oder Text angezeigt werden. Das Ändern von Werten ist aber nur über die Tabellenansicht möglich.

Abb. 4-9: Darstellung der XML-Datei als Tabelle, Baumstruktur oder Text

Die XML-Datenstruktur ist so aufgebaut, dass ein Testumgebungs-Projekt beliebig viele Funktionsblock-Tests enthalten kann. Jeder dieser Tests verfügt über Testsignale, die sich aus Funktions-Segmenten zusammensetzen, die wiederum über ihre Parameter eingestellt werden. Die folgende Abbildung stellt diese Datenstruktur mit Beispielen dar.

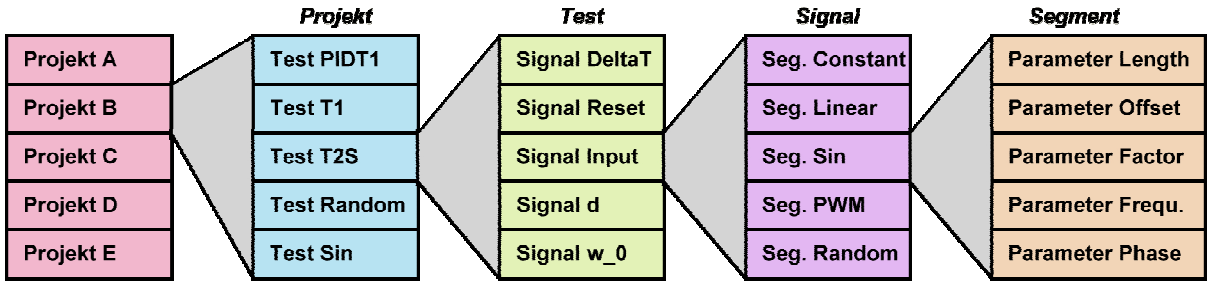


Abb. 4-10: Grober Aufbau der XML-Struktur

Führt der Benutzer einen Befehl aus, der den Inhalt der XML-Struktur verändert, erstellt die Testumgebung vor dem Übernehmen einen Wiederherstellungspunkt. Hierdurch kann der Entwickler über das Dropdown-Listenfeld „Bearbeiten“ jederzeit bis zu 100 Änderungen rückgängig machen und diese bei Bedarf wiederholen. Dabei ist zu beachten, dass, während ein Wiederherstellungspunkt geladen ist, das Ändern eines Wertes die Löschung aller jüngeren Wiederherstellungspunkte zur Folge hat.

Abb. 4-11: Dropdown-Listenfeld „Bearbeiten“

Erstellen eines Funktionsblock-Tests

Alle im Projekt erstellten Funktionsblock-Tests werden in der oberen Tabellenansicht angezeigt. Über einen Klick in die letzte, leere Zeile kann ein neuer Test hinzugefügt und anschließend über die eingeblendete Auswahl der zu testende Funktionsbaustein ausgewählt werden. Die vier Schaltflächen oben rechts ermöglichen einen bereits erstellten Funktionsblock-Test zu klonen (⧉), zu löschen (X) oder in der Liste zu verschieben (△▽). Die erste Spalte der Tabelle zeigt die Icons der Funktionsblöcke, die aus dem Verzeichnis „Images“ geladen und auf das Funktionsblock-Symbol zugeschnitten werden. In der zweiten Spalte steht der Namensbereich und der Klassenname des ausgewählten Funktionsblocks. Über die letzte Spalte kann der Benutzer den Funktionsblock-Tests kurze Notizen und Bemerkungen hinzufügen.

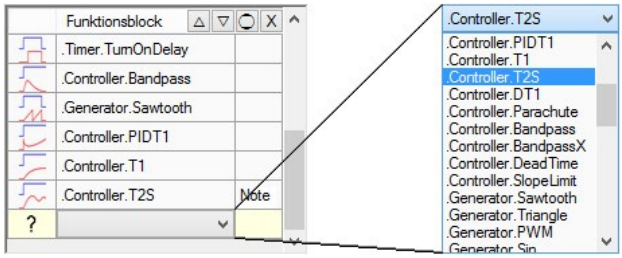


Abb. 4-12: Tabelle der Funktionsblock-Tests

In dem dargestellten Projekt wird zum Beispiel der markierte Test für den Funktionsblock „T2S“ aus dem Namensbereich „Controller“ der Bibliothek erstellt. Das hierzu angezeigte Icon ist unter dem Dateinamen „.Controller.T2S.png“ zu finden, der sich aus dem Namensbereich und Funktionsblock-Typ ableitet. Die Datei wird für die Anzeige zugeschnitten und freigestellt.

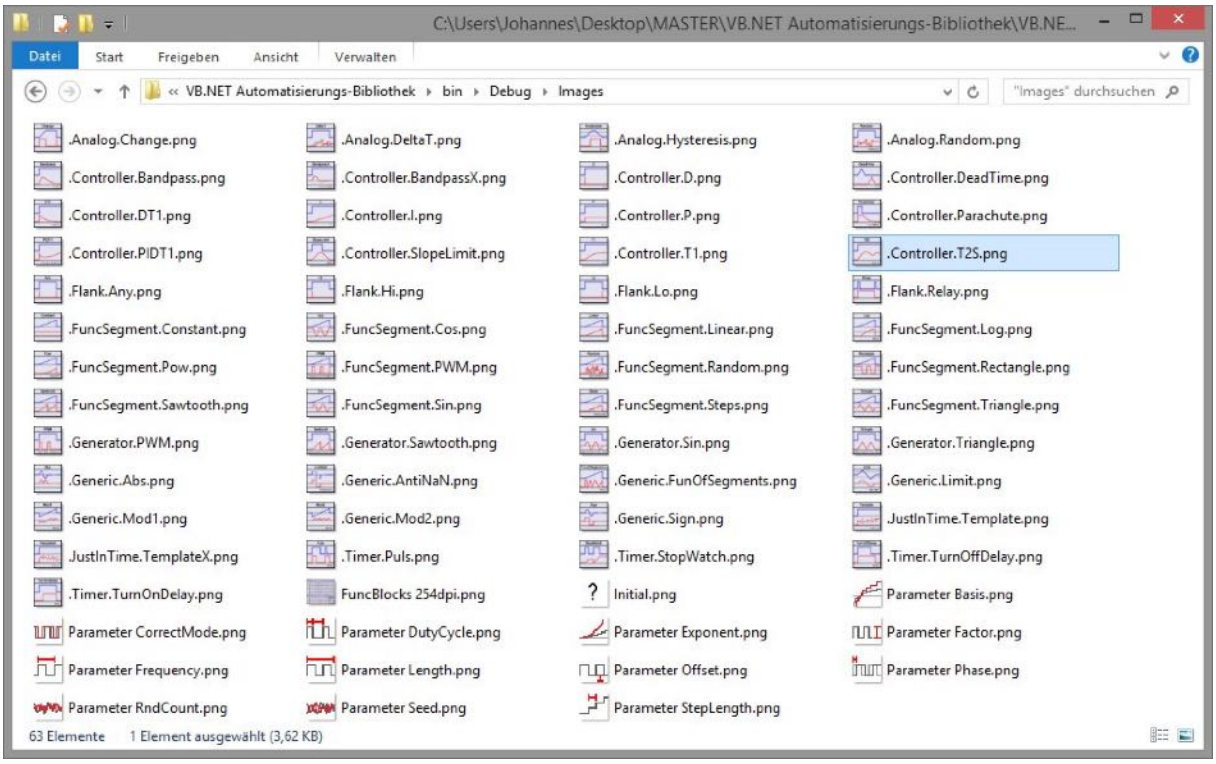


Abb. 4-13: Explorer-Ansicht des Unterverzeichnisses „Images“

Definieren der Testsignale

Wird ein Funktionsblock-Test ausgewählt, erscheinen in der folgenden Tabellenansicht die dazugehörigen Testsignale „DeltaT“, „Reset“, „Input“ und je nach verfügbaren Parametern des Funktionsblocks einige weitere. Wie bereits erklärt, bestimmt das Signal „DeltaT“, in welchen Zeitabständen der Funktionsblock aufgerufen wird, das Signal „Reset“, zu welchen Zeitpunkten der Baustein zurückgesetzt werden soll und das Signal „Input“, mit welchem Eingangswert der Funktionsblock aufgerufen wird. Alle weiteren Signale repräsentieren hingegen den zeitlichen Verlauf der Parameterwerte und werden abhängig von den Parameternamen des ausgewählten Funktionsblocks erstellt. Dabei belegt die Testumgebung die Signale mit einem konstanten Funktionssegment vor, das sich über die komplette Simulationszeit von 10 Sekunden erstreckt und als Offset den Standardwert des Funktionsbaustein-Parameters hat.

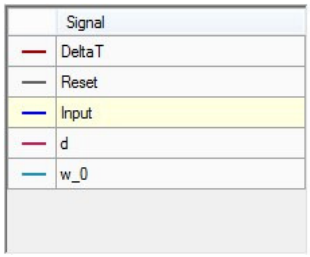


Abb. 4-14: Tabelle der Testsignale

Die in diesem Funktionsblock-Test verfügbaren Testsignale „DeltaT“, „Reset“ und „Input“ werden unabhängig von dem Typ des ausgewählten Funktionsblocks generiert und in der Tabelle als erstes angezeigt. Die Signale „d“ und „w_0“ wurden hingegen auf Basis der Parameternamen des zu testenden Funktionsblocks „T2S“ erstellt und sind alphabetisch sortiert angefügt. Die jeweilige farbliche Kennzeichnung der Signale richtet sich nach dem in der Tabelle zugewiesenen Zeilenindex.

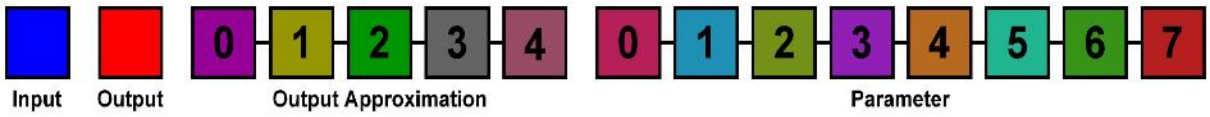


Abb. 4-15: Signal-Farbpalette

Die Testsignale eines Funktionsblock-Tests können auf neue Tests übertragen werden. Hierzu klont man den bereits erstellten Test und wählt nachträglich einen anderen Funktionsblock-Typen aus. Verfügt dieser nun teilweise über andere Parameter, so werden automatisch die bereits erstellten Signalverläufe gleichnamiger Parameter übernommen und die Signalverläufe nicht mehr vorhandener Parameter verworfen. Dies verhält sich genauso, wenn zum Beispiel durch die Weiterentwicklung eines Funktionsblocks neue Parameter hinzugefügt oder entfernt werden. Ändert man aber den Klassennamen eines Funktionsblocks in der Bibliothek, muss der dazu entworfene Test neu erstellt oder der Name auch in der XML-Datei des Test-Projekts durch den neuen ersetzt werden.

Hinzufügen von Funktions-Segmenten

Mit der nächsten Tabellenansicht werden die Testsignale abschnittsweise definiert. Ähnlich wie bei dem Erstellen eines Funktionsblock-Tests kann auch hier mit einem Klick in die letzte Zeile ein neues Element hinzugefügt und über die vier Schaltflächen das markierte Funktions-Segment geklont (⧉), gelöscht (X) oder in der Liste verschoben (△▽) werden. Auch hier wird in der ersten Spalte der Tabelle das Icon des Segments aus dem Verzeichnis „Images“ dargestellt und über die zweite Spalte der Namensbereich und Klassenname angezeigt. Die auswählbaren Funktions-Segmente sind nämlich auch Funktionsblöcke der Bibliothek, die unter dem Namensbereich „FuncSegments“ zu finden sind.

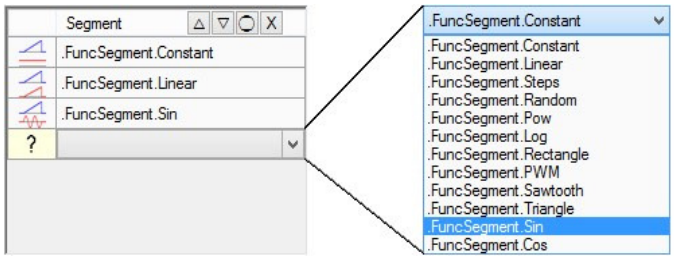


Abb. 4-16: Tabelle der Funktions-Segmente

Das in diesem Beispiel definierte Testsignal „Input“ (blau) beschreibt das Eingangssignal des zu testenden Funktionsblocks abschnittsweise. Die zum Erstellen des Signals verwendeten Funktionssegmente „Constant“, „Linear“ und „Sin“ erzeugen einen Signalverlauf, der sich erst konstant, dann linear fallend und anschließend nach einer Sinus-Funktion verhält.

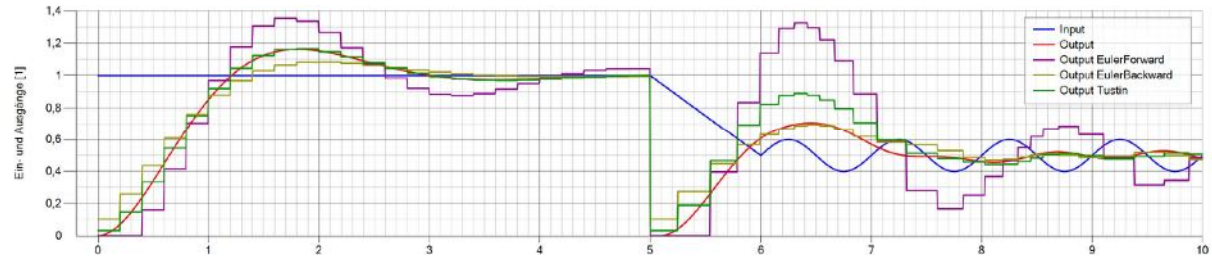


Abb. 4-17: Beispiel eines Eingangssignals „Input“ und dazugehörige Ausgangssignale

Die aktuell in der Bibliothek enthaltenen Funktionssegmente sind im Folgenden abgebildet.

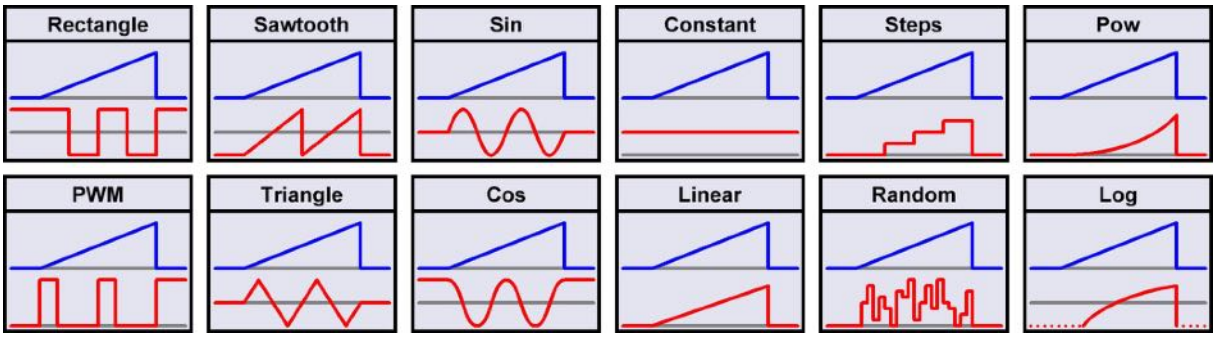


Abb. 4-18: Funktionssegmente zum Erstellen von Testsignalen

Ändern der Funktionssegment-Eigenschaften

Die genaue Signalform eines Funktionssegments wird durch seine Eigenschaften definiert, die zum aktuell ausgewählten Segment in der untersten Tabellenansicht angezeigt werden. Alle Funktions-Segmente verfügen über die Eigenschaften „Length“, „Offset“ und „Factor“, die in dieser Reihenfolge immer als erstes in der Tabelle angezeigt werden. Je nach gewähltem Funktionssegment werden noch zusätzliche Eigenschaften mit angezeigt, wie zum Beispiel bei dem Funktionssegment „Sin“ die Parameter „Frequency“ und „Phase“. Jeder Parameter kann mit einem Wert des Typs Double belegt werden. Hierzu gehören auch „NaN“, „$+unendlich$“ sowie „$-unendlich$“. Die Icons aus der ersten Spalte der Tabellenansicht werden auch aus dem Verzeichnis „Images“ geladen, wie z. B. „Parameter Frequency.png“.

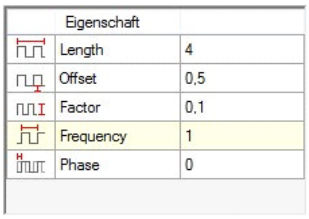


Abb. 4-19: Tabelle der Parameter

In der folgenden Tabelle sind die Parameter aller auswählbaren Funktions-Segmente erklärt.

Icon	Property-Name	Erklärung
	Length	Zeitliche Definitionslänge des Funktions-Segments
	Offset	Nullpunktverschiebung der Funktion
	Factor	Verstärkung der Funktion
	Frequency	Wiederholungen einer Funktionsperiode pro Sekunde
	Phase	Phasenverschiebung mit Wichtung ($1 \triangleq 360°$)
	StepLength	Stufenlänge (nur beim Funktionssegment Steps)
	DutyCycle	Tastgrad (Verhältnis von Impulsdauer zu Periodendauer) (nur beim Funktionssegment PWM)
	CorrectMode	frequenz- und phasenkorrekter Modus (wenn $\neq 0$) (nur beim Funktionssegment PWM)
	Exponent	Exponent $Input^{Exponent}$ (nur beim Funktionssegment Pow)
	Basis	Basis $\log_{Basis} Input$ (nur beim Funktionssegment Log)
	Seed	Startwert der deterministischen Zufallsfunktion (nur beim Funktionssegment Random)
	RndCount	Faktor zur Wahrscheinlichkeitsverteilung der Zufallswerte (nur beim Funktionssegment Random)

4.4 Schaubild

Das Schaubild der Benutzeroberfläche setzt sich aus vier Diagrammen zusammen, die den Werteverlauf aller Testsignale abhängig von der Simulationszeit darstellen. Im ersten Diagramm werden die Signale zur Steuerung der zeitlichen Taktung der Funktionsaufrufe abgebildet. Das nächste Diagramm zeigt alle Ein- und Ausgangssignale und das darauf folgende die dazugehörigen Quantisierungsfehler. Die Parameter des Funktionsblocks werden im letzten Diagramm des Schaubilds aufgezeigt.

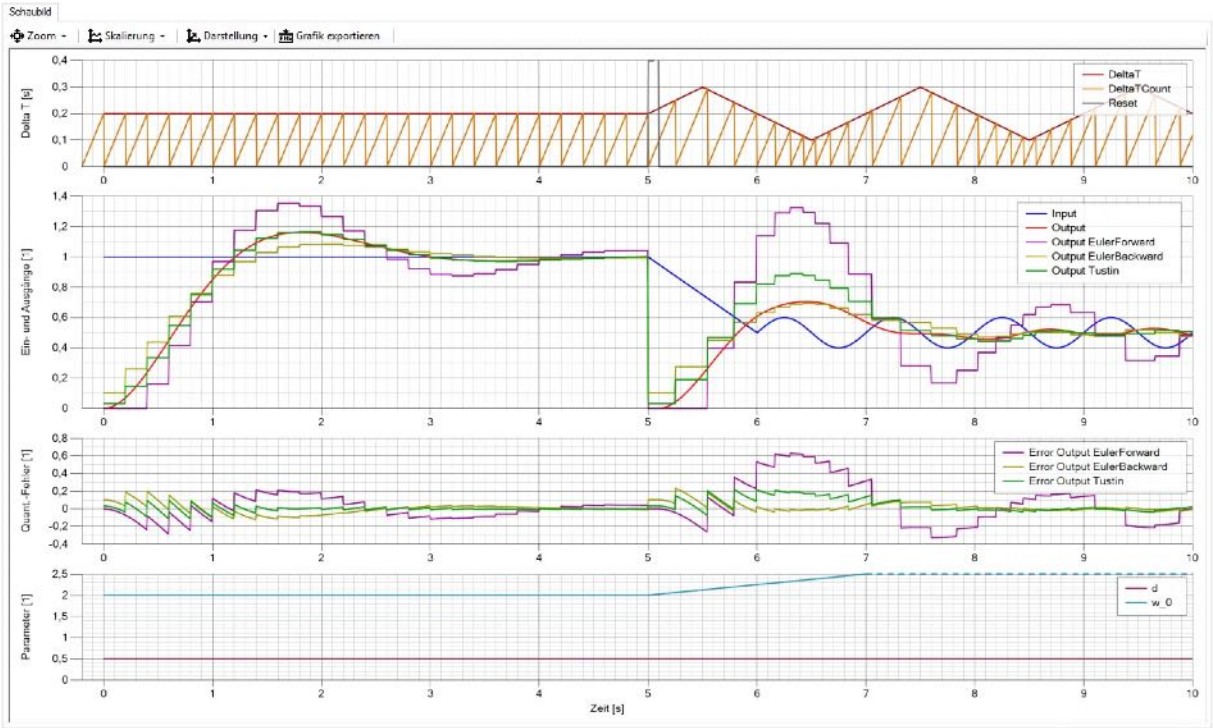


Abb. 4-20: Registerkarte „Schaubild“

Beim Ablesen von Werten aus den Diagrammen ist zu beachten, dass bei einem Wertesprung eines Signals an einem Punkt der Zeitachse das Signal den von der rechten Seite aus genäherten Wert repräsentiert. Die folgende Abbildung liefert ein Beispiel für diese Regel.

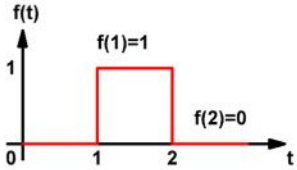


Abb. 4-21: Regel für Wertesprünge der Signale

Die Testumgebung passt die Darstellung der einzelnen Diagramme und deren Skalierung automatisch an das Testergebnis an. Je nach Funktionsblock-Typ werden die Graphen für die verschiedenen Approximations-Typen und Parameter automatisch hinzugefügt und in den Legenden mit den jeweiligen Signalnamen aufgeführt. Die Linienfarbe richtet sich dabei nach dem Signal-Index und entspricht somit der farblichen Markierung aus der Tabellenansicht.

Wie in der folgenden Abbildung zu sehen ist, befindet sich über dem Schaubild eine Symbolleiste, mit der der Benutzer den Zoom, die Skalierung und die Darstellung der Diagramme einstellen kann. Die Export-Funktion ist ebenfalls über diese Leiste zu erreichen.

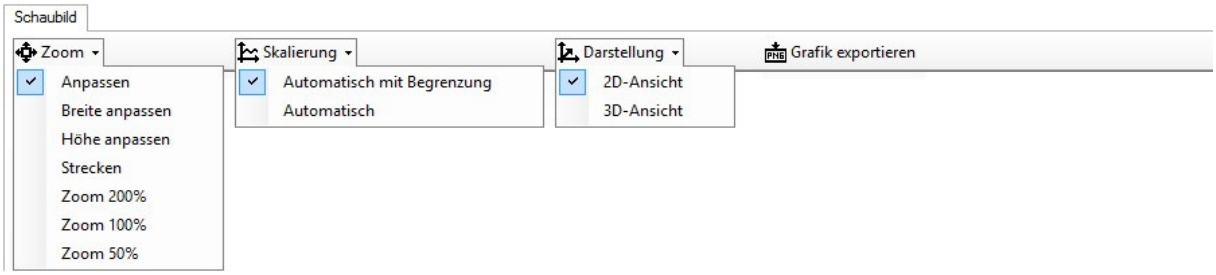


Abb. 4-22: Symbolleiste des Schaubilds

Alle Einstellungen und deren Optionen werden in den folgenden Abschnitten genauer erklärt.

Zoom-Modus

Mit der Einstellung „Zoom“ kann festgelegt werden, wie das Schaubild in die Benutzeroberfläche eingepasst wird. Auf die Exportfunktion der Grafik hat diese Einstellung keinen Einfluss.

Anpassen	Mit dieser Standardeinstellung wird das Schaubild so skaliert, dass es in den dafür vorgesehenen Container der Benutzeroberfläche passt. Das Seitenverhältnis wird dabei nicht verändert.
Breite anpassen	Nur die Breite des Schaubilds wird an die Größe des Anzeigecontainers angepasst. Das ursprüngliche Seitenverhältnis wird beibehalten.
Höhe anpassen	Nur die Höhe des Schaubilds wird an die Größe des Anzeigecontainers angepasst. Das ursprüngliche Seitenverhältnis wird beibehalten.
Strecken	Das Schaubild wird so gestreckt, dass es den Anzeigecontainer komplett ausfüllt. Das Seitenverhältnis wird möglicherweise verändert, so dass das Schaubild verzerrt erscheinen kann.
Zoom X%	Das Schaubild wird mit einem Vergrößerungsfaktor dargestellt und das ursprüngliche Seitenverhältnis dabei nicht verändert. Bei der Option von 100% nimmt die gesamte Grafik eine Pixelfläche von 2560 x 1440 Pixel ein.

Neben der Darstellung in der Benutzeroberfläche kann das Schaubild durch einen Doppelklick oder über das Kontextmenü, das mit der rechten Maustaste zu erreichen ist, auch im Vollbildmodus dargestellt werden.

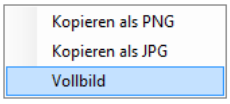


Abb. 4-23: Kontextmenü des Schaubilds

Skalier-Modus

Für die Skalierung der Diagramme stehen zwei Möglichkeiten zur Auswahl, die sich ausschließlich auf die Ordinatenachsen auswirken. Zum einen die Option „Automatisch mit Begrenzung“ und zum anderen „Automatisch“. Die erste Option mit Begrenzung ist speziell für das Testen von typischen Funktionsblöcken ausgelegt. Das bedeutet, dass dieser Skalier-Modus je nach Diagramm sinnvoll festgelegte Wertegrenzen nicht überschreitet. Reißt zum Beispiel ein Signalwert aus, werden große Betragswerte mit einer gestrichelten Linie angedeutet, anstatt die Ordinatenachse extrem groß zu skalieren.

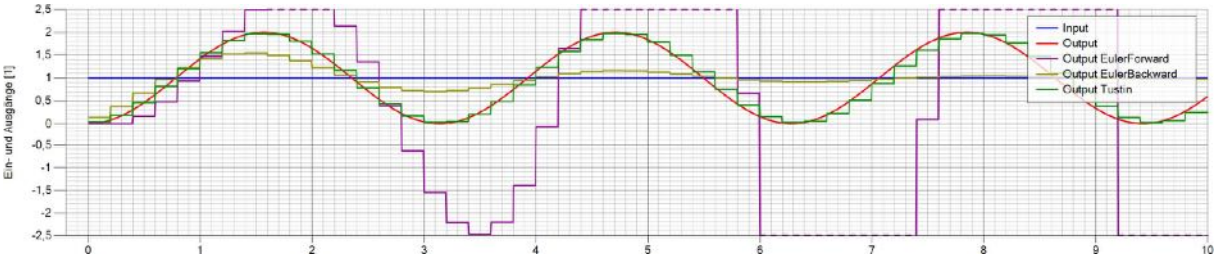


Abb. 4-24: Signalverläufe mit Skalier-Modus „Automatisch mit Begrenzung“

Die zweite Option „Automatisch“ überlässt das automatische Skalieren der Ordinatenachse komplett dem Steuerelement des .NET-Frameworks. Dies hat zur Folge, dass die Skalierung so groß gewählt wird, dass der komplette Signalverlauf sichtbar ist. In diesem Beispiel wird hierdurch der Verlauf des ausreißenden Graphen sichtbar, aber mit der Konsequenz, dass alle anderen Graphen mit verhältnismäßig kleinen Wertebereichen aufgrund der großen Skalierung extrem klein dargestellt werden.

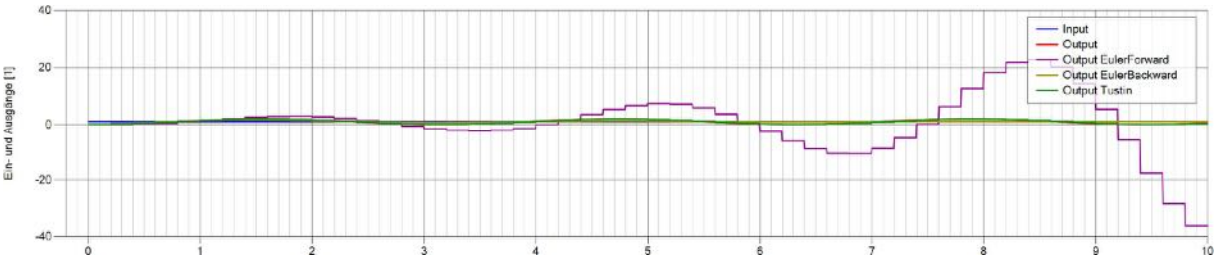


Abb. 4-25: Signalverläufe mit Skalier-Modus „Automatisch“

Somit kann je nach Bedarf der Fokus auf verschiedene Graphen gelegt werden. Die Skalierung der Abszissenachse ist fix auf den Zeitbereich der Simulationszeit von -0,2 bis 10,0 Sekunden festgelegt und kann nicht verändert werden.

Der spezielle Wert „Double.NaN“, den eine Gleitkommazahl und somit auch ein Signal darstellen kann, wird im Diagramm mittels einer Unterbrechung des Signals angezeigt. Die Werte „Double.PositiveInfinity“ und „Double.NegativeInfinity“ werden zudem im Skalier-Modus mit Begrenzung als gestrichelte Linie dargestellt und im Skalier-Modus ohne Begrenzung durch den Wert „Integer.MaxValue“ bzw. „Integer.MinValue“ ersetzt.

Darstellungs-Modus

Die Diagramme des Schaubilds können in einer 2D- und in einer 3D-Ansicht dargestellt werden. Bei der zweidimensionalen Darstellung werden Graphen mit gleichem Werteverlauf direkt übereinander gezeichnet. Dies hat zur Folge, dass bei der Darstellung von vielen Signalen in einem Diagramm der Verlauf von manchen Signalen nicht mehr eindeutig nachvollziehbar sein kann. Ein Beispiel hierfür liefert das blau eingefärbte Signal des folgenden Diagramms.

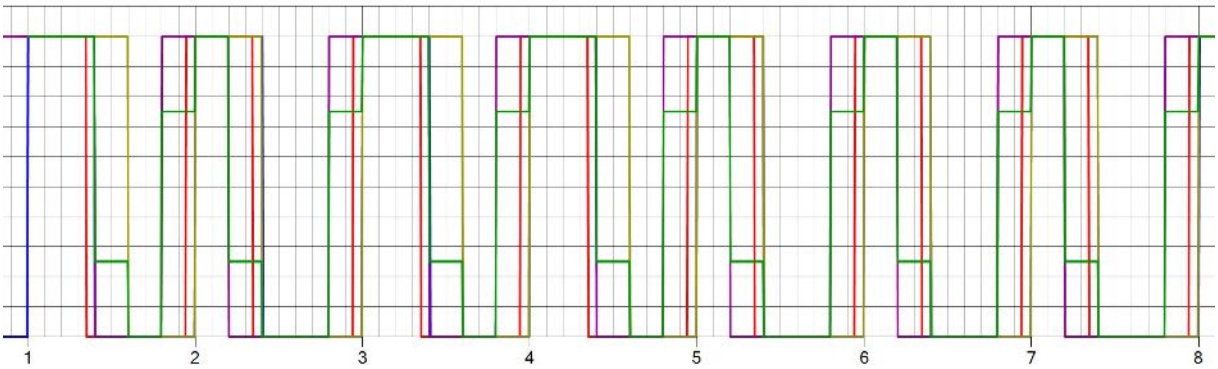


Abb. 4-26: Darstellungs-Modus „2D-Ansicht“

In solchen Fällen kann auf die dreidimensionale Ansicht umgeschaltet werden, um die Graphen optisch zueinander verschoben darzustellen. In dem Beispiel wird es somit möglich, den Verlauf des blauen Signals nachzuvollziehen.

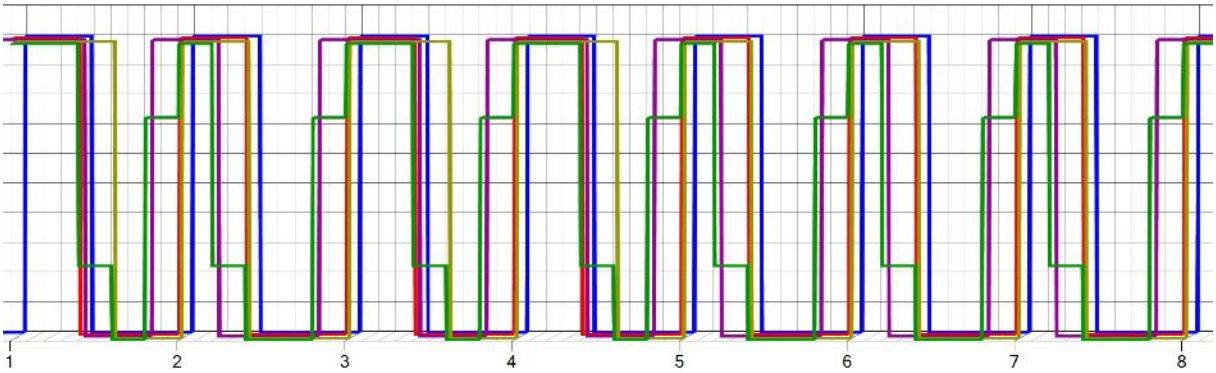


Abb. 4-27: Darstellungs-Modus „3D-Ansicht“

Die Reihenfolge, in der die Graphen gezeichnet werden, kann nicht verändert werden.

Schaubild exportieren

Das Schaubild kann über die Schaltfläche „Grafik exportieren“ der Symbolleiste als Bilddatei gespeichert oder über das Kontextmenü des Schaubilds in die Zwischenablage kopiert werden. Hierbei stehen zum einen das verlustfreie Dateiformat PNG oder das JPG-Format mit einer Komprimierungsqualität von 90% zur Auswahl. Die Auflösung der exportierten Grafik ist auf 2560 x 1440 Bildpunkte festgelegt, was bei einer Breite von 16 cm einer Punktdichte von ca. 400 dpi entspricht.

Schaubild drucken

Die Testumgebung stellt neben der Exportfunktion auch eine integrierte Druckfunktion für das Schaubild zur Verfügung, die über das Dropdown-Listenfeld „Datei“ aufgerufen werden kann.

Abb. 4-28: Dropdown-Listenfeld „Datei“

Als Ausgabegeräte können alle installierten Druckertreiber gewählt werden. Mit einem installierten „PDF-Creator“ ist somit zum Beispiel auch die Ausgabe als PDF-Datei möglich.

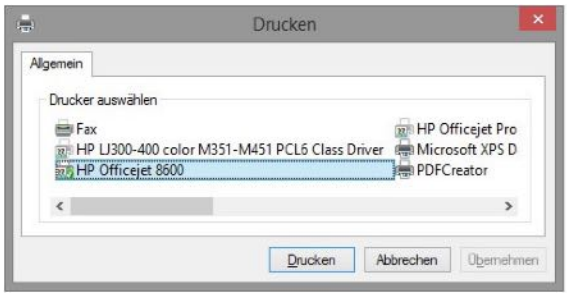


Abb. 4-29: Dialogfeld „Drucken“

Noch bevor der Druckauftrag ausgegeben wird, zeigt die Testumgebung das Schaubild zum Überprüfen in einer Seitenansicht an. Mit dem kleinen Druckersymbol in der Ecke links oben wird der Druckauftrag bestätigt und dem Spooler übergeben.

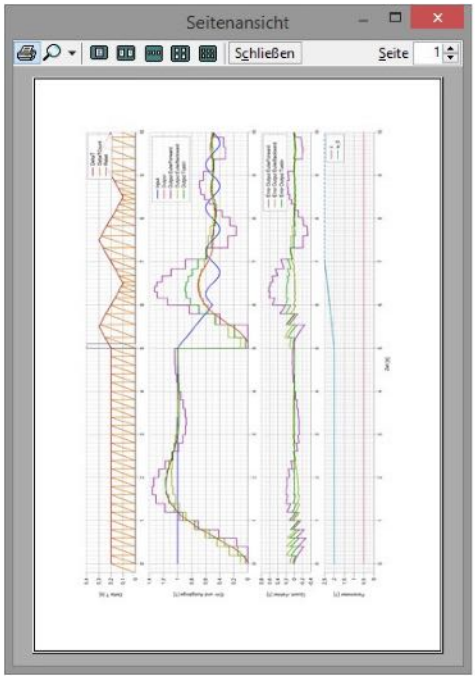


Abb. 4-30: Fenster „Seitenansicht“

4.5 Wertetabelle

Neben dem Schaubild wird das Testergebnis zusätzlich noch in einer Tabelle dargestellt, die über die zweite Registerkarte zu erreichen ist. Die Spalten der Wertetabelle entsprechen den Signalen des Testergebnisses und sind auch so eingefärbt. Jede Zeile stellt jeweils alle Signalwerte zu einem Zeitpunkt eines Simulationszyklus dar.

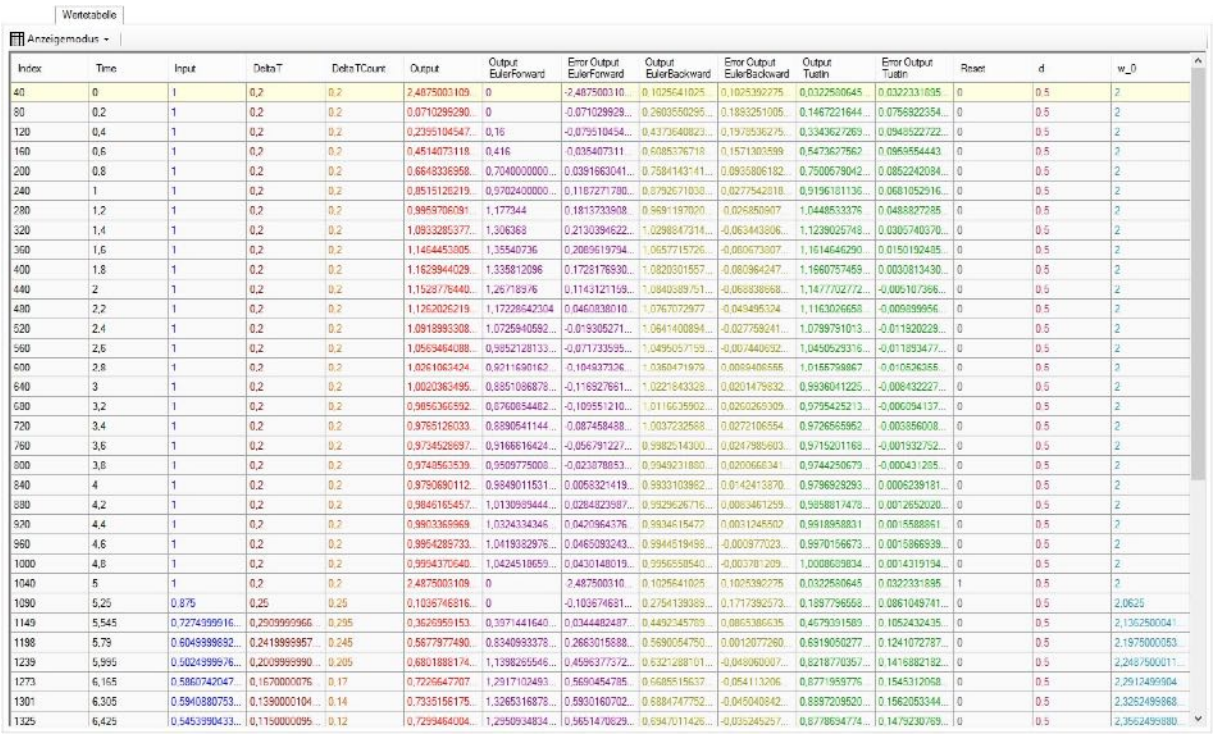


Abb. 4-31: Registerkarte „Wertetabelle“

Oberhalb der Tabelle kann mit der Schaltfläche „Anzeigemodus“ ein Filter ausgewählt werden, der entweder die Werte aller Zyklen der Simulation anzeigt oder nur die Werte von Zyklen herausfiltert, bei denen $DeltaTCount \geq DeltaT$ ist bzw. bei denen alle Funktionsblöcke aufgerufen wurden.

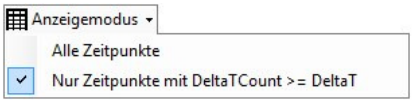


Abb. 4-32: Dropdown-Listenfeld zum Auswählen des Anzeigemodus der Wertetabelle

Für die Verarbeitung der Signalwerte in einem externen Programm können alle Zellen der Wertetabelle mit der Tastenkombination Strg + A markiert und anschließend mit Strg + C in die Zwischenablage kopiert werden. Das Sortieren der Spalten, um zum Beispiel schnell Extremwerte zu finden, ist bereits in der Testumgebung möglich.

4.6 Integrierte Entwicklungsumgebung

In die Testumgebung ist eine Entwicklungsumgebung integriert, mit der neue Funktionsblöcke programmiert werden können. Zum Erstellen eines neuen Funktionsblocks erstellt man, wie bereits erklärt, einen neuen Funktionsblock-Test und wählt dabei einen Funktionsblock aus dem Namensbereich „JustInTime“ aus. Anschließend erscheint automatisch die zusätzliche Registerkarte „Code“, über die der Programmcode des Funktionsblocks bearbeitet werden kann. Mit der Schaltfläche „Quellcode zurücksetzen“ erhält man eine Funktionsblock-Vorlage, auf die man einfach aufbauen kann.

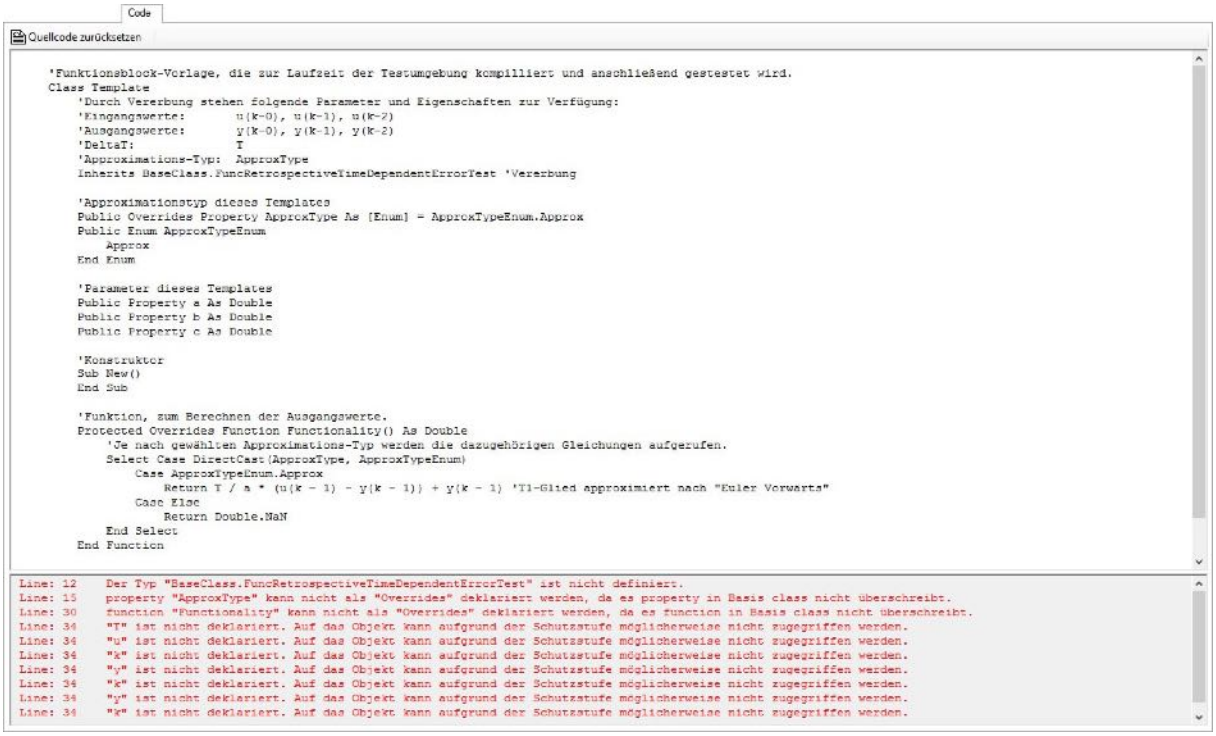


Abb. 4-33: Code-Editor der integrierten Entwicklungsumgebung

Der große Vorteil beim Entwickeln der Funktionsblöcke in der Testumgebung besteht darin, dass der Programmcode sofort nach einer Änderung im Hintergrund kompiliert und der dazu erstellte Funktionsblocktest ausgeführt wird. Im unteren Bereich des Editors werden dazu Meldungen des Just-In-Time-Compilers angezeigt und Fehler, die beim Ausführen des Funktionsblock-Tests auftreten, direkt per Popup-Fenster gemeldet. Das Zurücksetzen und Wiederholen von Codeänderungen ist wie beim Erstellen eines Funktionsblock-Tests über das Dropdown-Listenfeld „Bearbeiten“ oder über die Tastenkombinationen Strg + Z/Y möglich.

Detaillierte Beschreibungen zum Aufbau des Programmcodes sind in den Programmvorlagen als Kommentare zu finden.

4.7 Fehlerbehebung

Die Testumgebung führt die Funktionsblock-Tests in separaten Threads im Hintergrund aus. Die Statusleiste im unteren Bereich der Testumgebung zeigt dazu den Fortschritt und den Namen des aktuellen Prozessschritts an. Tritt bei irgendeinem Schritt ein Fehler auf, bleibt die Fortschrittsanzeige stehen und zeigt Fehlerdetails an.

Abb. 4-34: Statusleiste

Die Prozesse, die im Hintergrund ausgeführt werden, sind in der folgenden Tabelle kurz erklärt.

Status	**Beschreibung**
„Start Thread...“	Der Hintergrundthread läuft und überprüft, ob der zu testende Funktionsblock in der Assembly zu finden ist. Bei Fehlern wird „Reflection Error!“ in der Statusleiste angezeigt.
„Compiling...“	Der Just-In-Time-Compiler übersetzt den Code eines über die Testumgebung erstellten Funktionsblocks. Bei Fehlern wird „Compiling Error!“ in der Statusleiste angezeigt.
„Serialize...“	Der Funktionsblock serialisiert die Datenstruktur des Funktionsblock-Tests zu einer Zeichenkette im XML-Format.

„Simulate...“	Die Simulation wird auf Basis der XML-Zeichenkette ausgeführt und das Testergebnis als XML-Zeichenkette zurückgegeben.
	Zuerst wird für alle zu ermittelnden Ausgangssignale eine einzelne Instanz des zu testenden Funktionsblocks erstellt und anschließend das Testergebnis mit folgendem Zyklus berechnet:
	- Inkrementieren der Simulationszeit um 5 ms pro Zyklus
	- Wenn die $Simulationszeit \geq 0$, dann für das Signal „Output“ (Ideal) immer und für alle Signale „Output <Approximation>“, nur wenn $DeltaTCount \geq DeltaT$ ist, die folgende Sequenz ausführen:
	  - Parameter in den Funktionsblock laden
	  - Reset-Methode aufrufen (nur wenn Signalwert $Reset > 0$ und $DeltaTCount \geq DeltaT$)
	  - Ausgangsfunktion aufrufen (Signalwert $DeltaT$ mit übergeben)
	  - Quantisierungsfehler berechnen
	  - Signal $DeltaTCount$ zurücksetzen (wenn $DeltaTCount \geq DeltaT$)
	- Wenn $Simulationszeit \geq 10$, dann Simulation beenden

„Refresh Table...“	Die Wertetabelle wird aktualisiert.
„Render Chart...“	Das Schaubild wird gerendert.
„Update GUI...“	Das gerenderte Schaubild wird in die Benutzeroberfläche geladen.
„Finish...“	Der Funktionsblock-Test wurde erfolgreich abgeschlossen.
„Ready“	Der Hintergrundthread ist beendet.

5 Beispielprojekt Audio-Analyse

Im Rahmen der Masterarbeit wurde ein Beispielprojekt erstellt, das auf den Elementen der Automatisierungsbibliothek basiert und deren Möglichkeiten veranschaulichen soll. Es handelt sich hierbei um ein Softwareprojekt, das ein Audiosignal abtastet, analysiert und mit einem Lichteffektgerät visualisiert. In Anbetracht der Tatsache, dass der Mensch kleinste Verzögerungen zwischen akustischen und visuellen Reizen sehr gut erkennen kann, lässt sich mit diesem Beispielprojekt die weiche Echtzeitfähigkeit der Automatisierungsbibliothek sehr gut demonstrieren. Da bei der eingesetzten Audio-Analyse 48.000 Messwerte pro Sekunde aufwändig verarbeitet werden, lässt sich des Weiteren die Effektivität gut abschätzen, mit der die Automatisierungsbibliothek die Leistungsfähigkeit des Prozessors nutzt.

Der Testaufbau setzt sich wie folgt zusammen: Eine Wiedergabesoftware, in diesem Fall der „DJ JOE Genius“ in der folgenden Abbildung links, spielt eine Musikdatei ab und gibt das Audiosignal über den Soundkarten-Ausgang aus. Über einen Y-Adapter gelangt das Signal zum einen in eine Kompaktanlage und zum anderen in den Soundkarten-Eingang. Hier wird das Audiosignal abgetastet und von der Software „Audio Streamer“ als Stream im Netzwerk zur Verfügung gestellt. Die Software „Audio Analyzer“ empfängt den Stream, analysiert das Audiosignal und steuert über das Lichttechnikprotokoll DMX eine LED-Leiste an.

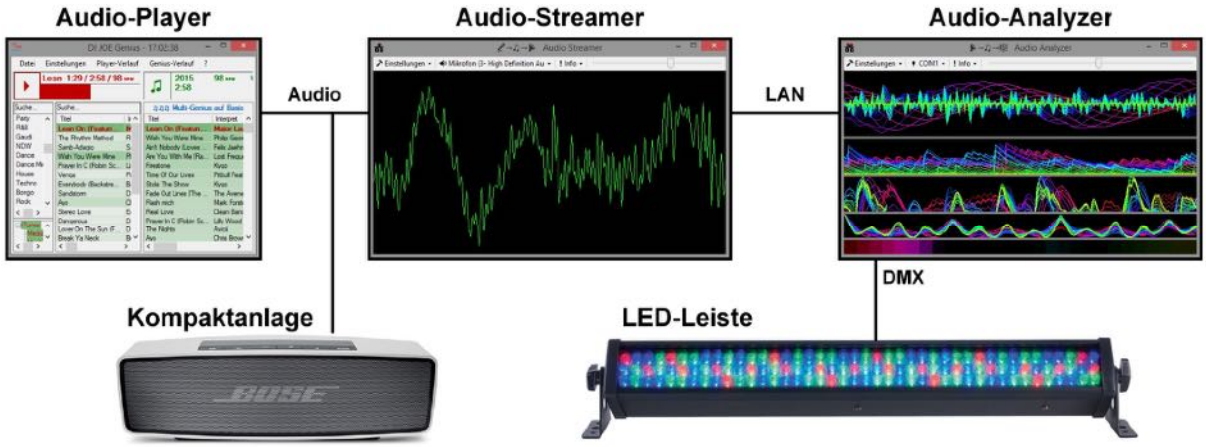


Abb. 5-1: Testaufbau des Beispielprojekts

Die LED-Leiste stellt die aktuell auftretenden Geräuschpegel der Musik in Abhängigkeit von deren Frequenzen visuell dar. Hört man auf signifikante Töne der wiedergegebenen Musik und beobachtet dabei das Verhalten der LED-Leiste, so stellt man fest, dass es zu keinen erkennbaren Verzögerungen zwischen Bild und Ton kommt. Genau genommen hat der Schall gerade einmal zwei Meter zurückgelegt, bis die LED-Leiste reagiert. Von den 6 ms Verzögerungszeit ist der Audio-Analyse selbst nur 1 ms zuzuschreiben. Die größte Latenz wird durch den internen Audio-Puffer der Soundkarte und die maximale Aktualisierungsrate der LED-Bar verursacht.

Die für dieses Projekt eingesetzten Programme „DJ JOE Genius“, „Audio Streamer“ und „Audio Analyzer“ wurden eigenständig programmiert. In den folgenden Abschnitten werden die drei Programme vorgestellt, einige Möglichkeiten, die VB.NET bietet, veranschaulicht und erklärt, wie Funktionsblöcke der Automatisierungsbibliothek eingesetzt wurden.

5.1 Programm „DJ JOE Genius“

Das Programm „DJ JOE Genius“ ist eine Verwaltungs- und Wiedergabesoftware für Disc-Jockeys. Das Besondere an dieser Software ist, dass zum aktuell gespielten Titel dazu passende Titel vorgeschlagen werden. Hierzu orientiert sich das Programm an Spielfolgen, die in eigenen Wiedergabelisten und Listen anderer DJs hinterlegt werden können.

Das Programm stellt im linken Fensterbereich alle Wiedergabelisten aus iTunes, dem Musikverwaltungsprogramm von Apple, dar. Der Player oben in der Mitte ist mit dem iTunes-Player gekoppelt und ermöglicht die Wiedergabe der Titel. Im rechten Fensterbereich ist die Genius-Funktionalität angeordnet, die nach zueinander passenden Titeln sucht.

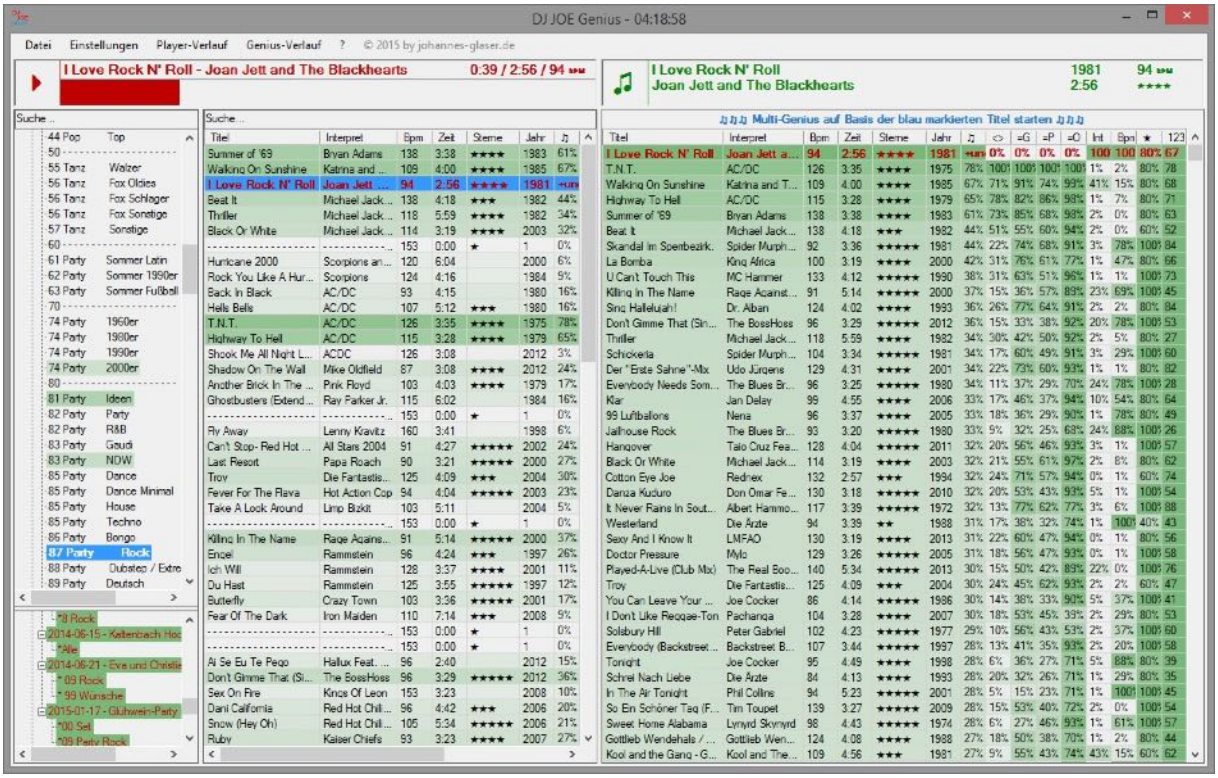


Abb. 5-2: Benutzeroberfläche des Programms „DJ JOE Genius“

Für dieses Projekt erfüllt das Programm „DJ JOE Genius“ hauptsächlich zwei Aufgeben. Zum einen wird gezeigt, dass mit VB.NET eigene Datenbankstrukturen erstellt werden können, um sehr spezielle Abfragen wie die Genius-Suche durchzuführen. Zum anderen dient das Programm als Wiedergabegerät, um die Audio-Analyse mit einem Audiosignal zu versorgen. In folgenden geplanten Projekten sollen Komponenten der später behandelten Audio-Analyse implementiert werden, um bei der Genius-Suche auch den Rhythmus der Lieder zu vergleichen.

5.1.1 Funktionsweise

Wird die Schaltfläche zum Starten der Genius-Suche gedrückt, ordnet das Programm alle Titel der Bibliothek abhängig davon, wie gut der jeweilige Titel zum aktuell gespielten Titel passt, und stellt das Ergebnis in der rechten Tabellenansicht dar. Um das Ergebnis zu präzisieren, können anschließend mehrere Titel aus der Ergebnisliste ausgewählt und die Genius-Suche auf Basis dieser Titel erneut durchgeführt werden. Bei der Genius-Suche wird berücksichtigt, wie häufig sich Titel in der gleichen Gruppe, Wiedergabeliste und im gleichen Ordner befinden. Des Weiteren werden die Ähnlichkeit des Interpreten, die Taktgeschwindigkeit sowie Bewertungen mit einbezogen.

Neben der Genius-Suche bietet das Programm auch eine Textsuche, mit der sowohl nach Titeln als auch nach Wiedergabelisten gesucht werden kann. Die Suche ist sehr robust ausgelegt und gibt immer die 100 Titel zurück, die am besten zur Suchanfrage passen. Gibt man zum Beispiel fälschlicherweise „Haiway to hell acdc ;-)“ ein, wird der gesuchte Titel „Highway to hell“ unter ca. 8.000 Elementen der Bibliothek trotzdem ohne Probleme gefunden und als erstes in der Ergebnisliste angezeigt. Die Helligkeit der Hintergrundfarbe gibt hierbei die Trefferpunkte an.

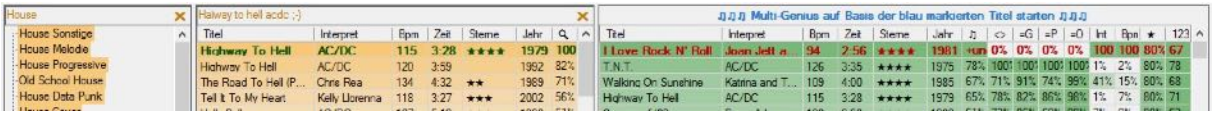


Abb. 5-3: Programm DJ-JOE-Genius

5.1.2 Programmierung

Die Anforderungen der zu implementierenden Funktionalitäten waren sehr hoch, weil eine Musikbibliothek oft aus mehreren tausend Titeln besteht und die Genius- und Such-Funktionen sehr rechenaufwändig sind. Nur die Erstellung einer speziell auf die Anwendung zugeschnittenen Datenbankstruktur ermöglichte es, solche Funktionalitäten zu implementieren und durch mehrere Threads effektiv mit Rechenleistung zu bedienen. In der aktuellen Version wird somit eine Genius-Suchanfrage bei einer Musikbibliothek, die 8.000 Titel umfasst, in weniger als einer Sekunde ausgeführt.

5.1.3 Einsatz der Automatisierungsbibliothek

Elemente der Automatisierungsbibliothek kommen bei diesem Programm hauptsächlich bei der Genius-Funktion zum Einsatz, um die Gewichtungsfunktionen, die zum Vergleichen der Titel erforderlich sind, abschnittsweise zu definieren. Hierzu ermöglicht der Funktionsbaustein „FuncOfSegments“ das Aneinanderreihen beliebiger Funktionssegmente unterschiedlichen Typs. Somit ist es zum Beispiel möglich, beim Vergleichen die Titel mit einer niedrigen Taktfrequenz-Differenz besonders positiv und Titel mit einer hohen Differenz der Taktfrequenz möglichst gleich für das Genius-Ergebnis zu werten. Hierbei können in den Gewichtungsfunktionen unscharfe Schwellwerte festgelegt werden.

5.2 Programm „Audio Streamer“

Der „Audio Streamer“ stellt das Signal einer beliebigen Aufnahmequelle, z. B. Eingang der Soundkarte oder Mikrofon, im Netzwerk als Audio-Stream zur Verfügung. Hierbei wird der Eingangspegel automatisch angepasst und die Wellenform grafisch visualisiert.

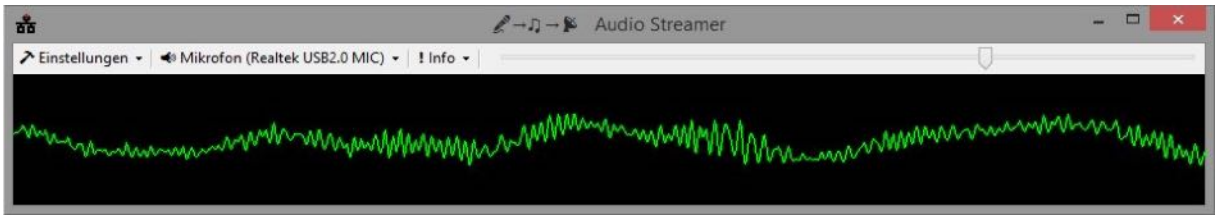


Abb. 5-4: Benutzeroberfläche des Programms „Audio Streamer“

5.2.1 Funktionsweise

Nach dem Öffnen des Programms verbindet sich die Software automatisch mit dem ausgewählten Standard-Aufnahmegerät der Windows-Einstellungen. Alle weiteren Eingänge sind über ein Dropdown-Menü anwählbar, das in der folgenden Abbildung rot markiert ist.

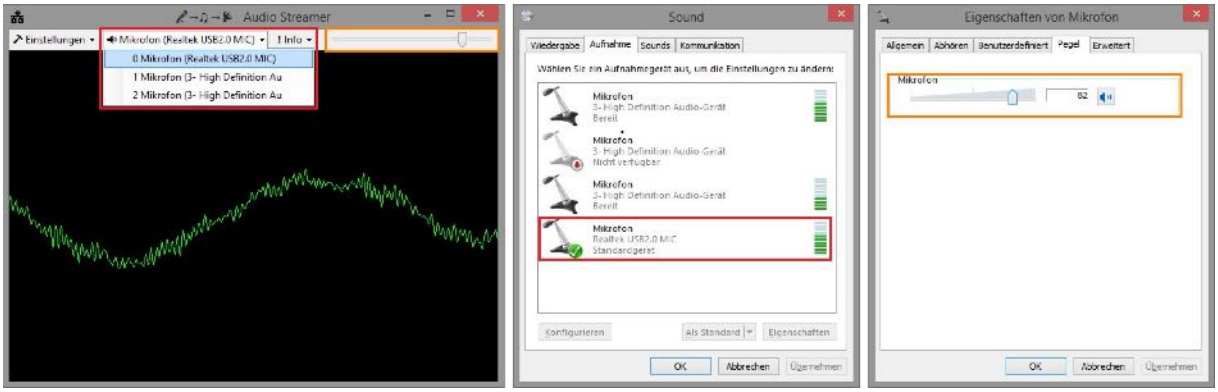


Abb. 5-5: Aufnahmegeräte (rot) und automatische Lautstärke-Regelung (orange)

Die orange umrahmten Schieberegler im Programm und in den Windows-Einstellungen stellen den Wert des Eingangsverstärkers des ausgewählten Aufnahmegeräts dar und sind softwareseitig miteinander gekoppelt. Um den Verstärkungsfaktor nicht manuell anpassen zu müssen, verfügt der Audio-Streamer über eine automatische Lautstärke-Regelung, die den Eingangspegel automatisch auf ein Optimum anpasst. Diese Funktion lässt sich in den Einstellungen des Programms aktivieren und deaktivieren, wobei das manuelle Verstellen der Schieberegler davon unabhängig jederzeit möglich ist.

Die grüne Linie visualisiert die Wellenform des Eingangssignals, wobei jeder dargestellte Pixel in horizontaler Richtung einen abgetasteten Signalwert repräsentiert. Somit ergibt sich die Aktualisierungsrate der Grafik aus der Abtastrate und der eingestellten Fensterbreite. Um den Prozessor zu entlasten, kann über die Einstellungen des Programms die Aktualisierungsrate auf 30 Bilder pro Sekunde begrenzt oder durch Minimieren des Fensters die Grafik komplett deaktiviert werden. Ein Doppelklick schaltet den Vollbildmodus des Graphen ein.

5.2.2 Programmierung

Das Abtasten, Versenden und Visualisieren des Audio-Signals wird in drei voneinander getrennten Threads ausgeführt. Hierdurch wird vermieden, dass zum Beispiel das Rendern der Visualisierung die Signalverarbeitung mit Verzögerungszeiten negativ beeinflusst.

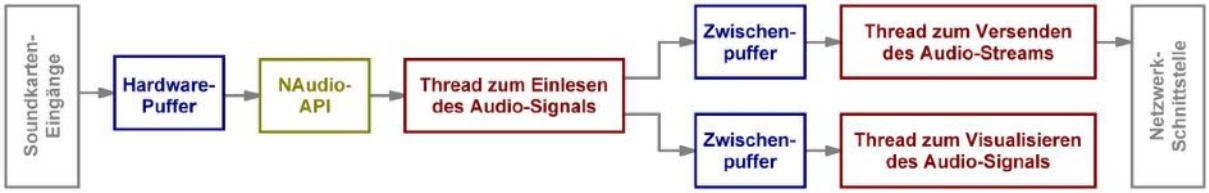


Abb. 5-6: Puffer und Threads des Audio-Streamers

Thread zum Einlesen des Audio-Signals

Der erste Thread holt die Audio-Daten von der Soundkarte ab, bereitet die Rohdaten auf und legt sie in zwei getrennte Zwischenpuffer zum Versenden und zum Visualisieren des Audio-Signals ab. Hierzu tastet die Soundkarte das Eingangssignal zweikanalig mit einer Abtastrate von 48.000 Hz bei einer Auflösung von 16 Bit ab und speichert die Audiodaten zunächst in einem Hardware-Puffer. Anschließend holt der Thread diese Rohdaten über die NAudio-API zyklisch ab und führt unter Berücksichtigung beider Kanäle (links und rechts) die automatische Lautstärkeregelung aus. Im gleichen Zyklus wird aus dem linken und rechten Audio-Signal ein Mono-Signal gebildet, das als Folge von Gleitkommazahlen (Typ Single) in die beiden Zwischenpuffer für Audio-Stream und Visualisierung abgelegt wird.

Thread zum Versenden des Audio-Streams

Der zweite Thread bedient sich an einem der beiden Zwischenpuffer und stellt das abgetastete Signal als Audio-Stream im Netzwerk zur Verfügung. Hierzu zerlegt der Thread zunächst den Datenstrom in Pakete mit maximal 800 Signalwerten, um für höchste Netzwerkkompatibilität die Datenpakete klein zu halten. Anschließend werden die Datenpakete mit einer fortlaufenden ID-Nummer sowie der Abtastrate versehen und als Audio-Stream mittels komprimierten UDP-Paketen an das Port 4242 der Broadcast-Adresse 255.255.255.255 des Netzwerks versendet. Die fortlaufende Nummerierung ermöglicht es, dem Empfänger die ankommenden Datenbündel in der chronologisch richtigen Reihenfolge zu verarbeiten.

Thread zum Visualisieren des Audio-Signals

Mit dem dritten Thread wird die Wellenform des Eingangssignals als Graph auf die Benutzeroberfläche gezeichnet. Hierzu wartet der Thread, bis sich im Zwischenpuffer genügend Daten angesammelt haben, um das Diagramm einmal über die Breite des Fensters zu zeichnen. Erst dann wird der Graph gerendert und angezeigt.

5.2.3 Einsatz der Automatisierungsbibliothek

Bei der Implementierung der einzelnen Funktionalitäten des Programms konnten einige Module aus zuvor programmierten VB.NET-Bibliotheken genutzt werden. Hierzu gehören zum Beispiel der Datenpuffer zum Zwischenspeichern des Audiosignals und eine Serialisierungs-Funktion, die Klassenobjekte in Zeichenketten umwandelt, um diese über das Netzwerk zu versenden.

Ein gutes Beispiel für den Einsatz der Automatisierungsbibliothek ist die automatische Lautstärke-Regelung des Eingangssignals. Hierbei wird die Eingangsverstärkung der Soundkarte bei zu niedrigem Eingangssignal automatisch erhöht und bei zu hohem Signal verringert. Diese Funktionalität konnte mit dem Funktionsbaustein „SlopeLimit“ aus der Automatisierungsbibliothek sehr einfach implementiert werden.

```
Input = SoundcardInput * SlopeLimit(If(Abs(Input) > 0.85, 0, 1))
```

Abb. 5-7: Programmcode zur Implementierung der automatischen Lautstärke-Regelung

Das folgende Blockschaltbild stellt den Signalfluss des obigen Programmcodes grafisch dar.

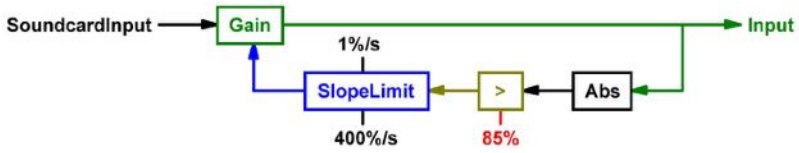


Abb. 5-8: Blockschaltbild der automatischen Lautstärkeregelung

Der Funktionsbaustein zur Steigungsbegrenzung (blau) nähert seinen Ausgangswert linear seinem Eingangswert an und begrenzt hierbei die Steigung des Ausgangssignals. In der folgenden Abbildung ist das Verhalten des Regelkreises dargestellt.

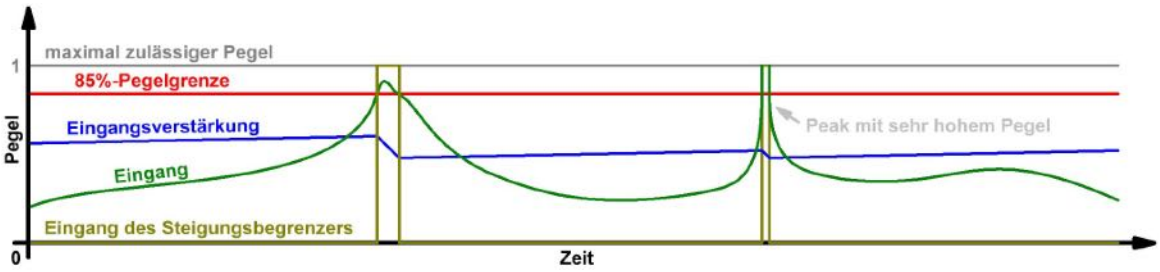


Abb. 5-9: Diagramm zur automatischen Lautstärke-Regelung

Überschreitet das Eingangssignal (grün) die Pegelgrenze von 85% (rot) des maximal zulässigen Pegels (grau), so liegt am Eingang des Steigungsbegrenzers (dunkelgelb) der Wert 1 an und der Verstärkungsfaktor wird mit 400% pro Sekunde sehr schnell verringert. Unterschreitet das Eingangssignal die Pegelgrenze, so liegt am Eingang des Steigungsbegrenzers der Wert 0 an und der Verstärkungsfaktor erhöht sich langsam mit 1% pro Sekunde. Die Regelung der Eingangsverstärkung gewährleistet, dass der Messbereich und somit die Auflösung des Analog- zu Digital-Wandlers der Soundkarte angemessen genutzt wird. Im Vergleich zu einem einfachen Proportionalregler ist diese Regelung gegenüber Peaks mit sehr hohem Pegel unempfindlich.

5.3 Programm „Audio Analyzer“

Das Programm „Audio Analyzer“ empfängt das vom „Audio Streamer“ gesendete Audiosignal, analysiert dieses und steuert davon abhängig eine LED-Leiste über das DMX-Protokoll an.

5.3.1 Funktionsweise

Die Audio-Analyse erfolgt in vier Schritten, die in der Benutzeroberfläche visualisiert sind. Im nächsten Unterkapitel werden diese genauer erläutert.

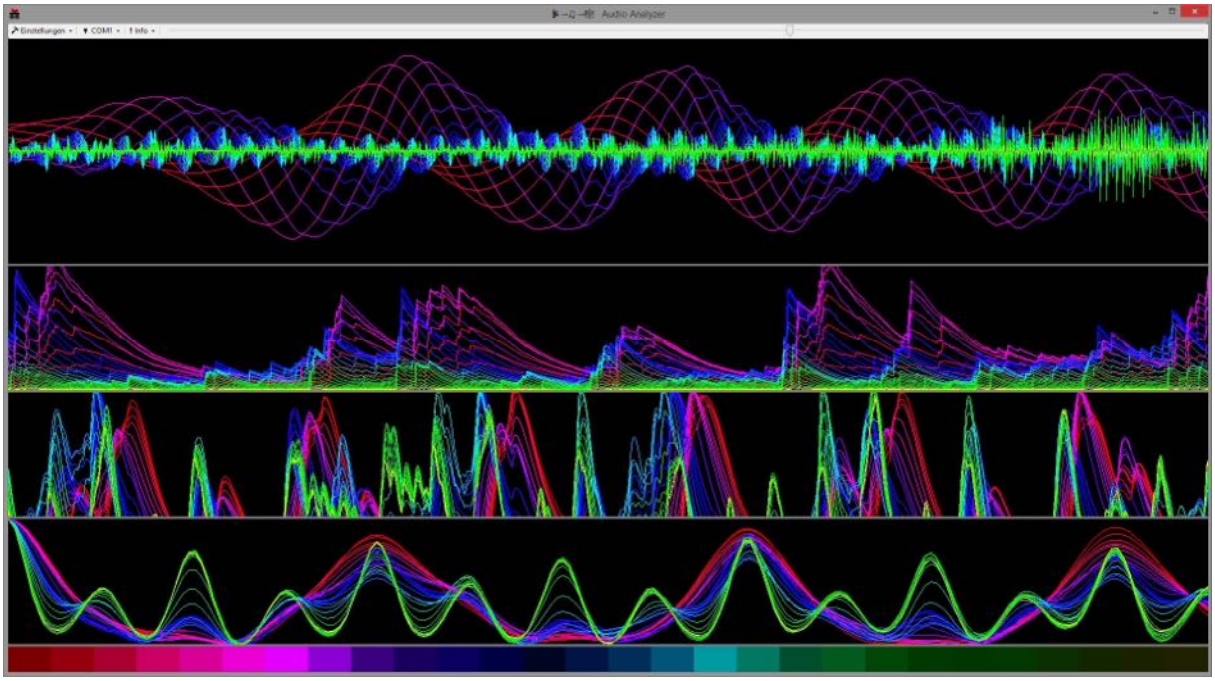


Abb. 5-10: Benutzeroberfläche des Programms „Audio Analyzer“

Über die Menüleiste der Benutzeroberfläche kann die DMX-Schnittstelle zur Ansteuerung der LED-Leiste ausgewählt, eine automatische Lautstärke-Regelung aktiviert und Einstellungen zu den Visualisierungen vorgenommen werden. So ist es zum Beispiel möglich, die Aktualisierungsrate der Grafiken auf 30 oder 1 Bild pro Sekunde zu reduzieren oder durch Minimieren des Fensters komplett zu deaktivieren, um die CPU zu entlasten. Durch einen Doppelklick werden die Visualisierungen hingegen im Vollbildmodus angezeigt.

Die folgende Abbildung zeigt die Zuordnung von Linienfarben und Grenzfrequenzen. Diese Legende kann ebenfalls über die Menüleiste aufgerufen werden.

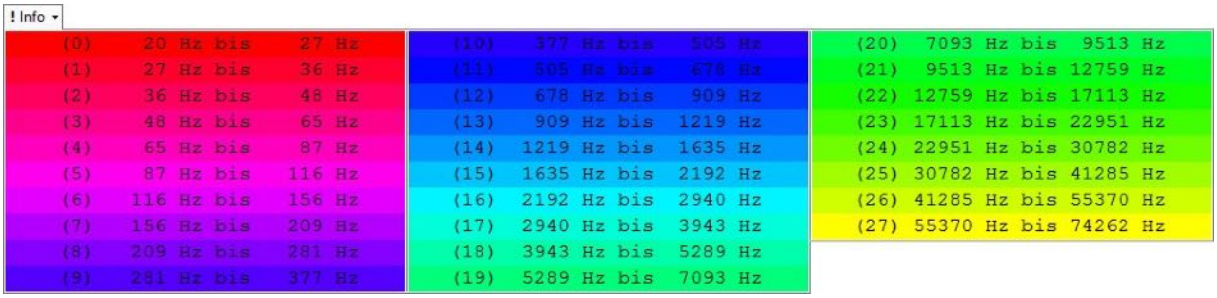


Abb. 5-11: Zuordnung der Linienfarben und Grenzfrequenzen

5.3.2 Programmierung

Das Audiosignal wird in sieben Abschnitten analysiert. In den ersten Abschnitten nehmen Funktionsaufrufe der Signalverarbeitung sehr kurze CPU-Zeiten in Anspruch, werden dafür aber wegen der hohen Datenrate sehr oft aufgerufen. In den letzten Abschnitten der Audio-Analyse kann mit geringeren Datenraten gearbeitet werden, wobei die hier eingesetzten Algorithmen zum Verarbeiten der Spektren pro Abtastung sehr viel Rechenleistung fordern. Aufgrund dieser speziellen Anforderungen der Audio-Analyse wird für jeden Abschnitt der Signalverarbeitung ein eigener Thread eingesetzt und die Datenrate des Signals in den Puffern zwischen den Threads umgerechnet.

Für diese Anwendung wurde ein spezieller Puffer programmiert, der threadsicherer ist und eine dynamische Abtastratenkonvertierung mit Unterdrückung von Aliasing-Effekten unterstützt. Dieser Puffertyp wird neben der Signalverarbeitungskette auch für alle Visualisierungen eingesetzt, um deren Threads mit Spektren in der geforderten Datenrate gepuffert zu bedienen.

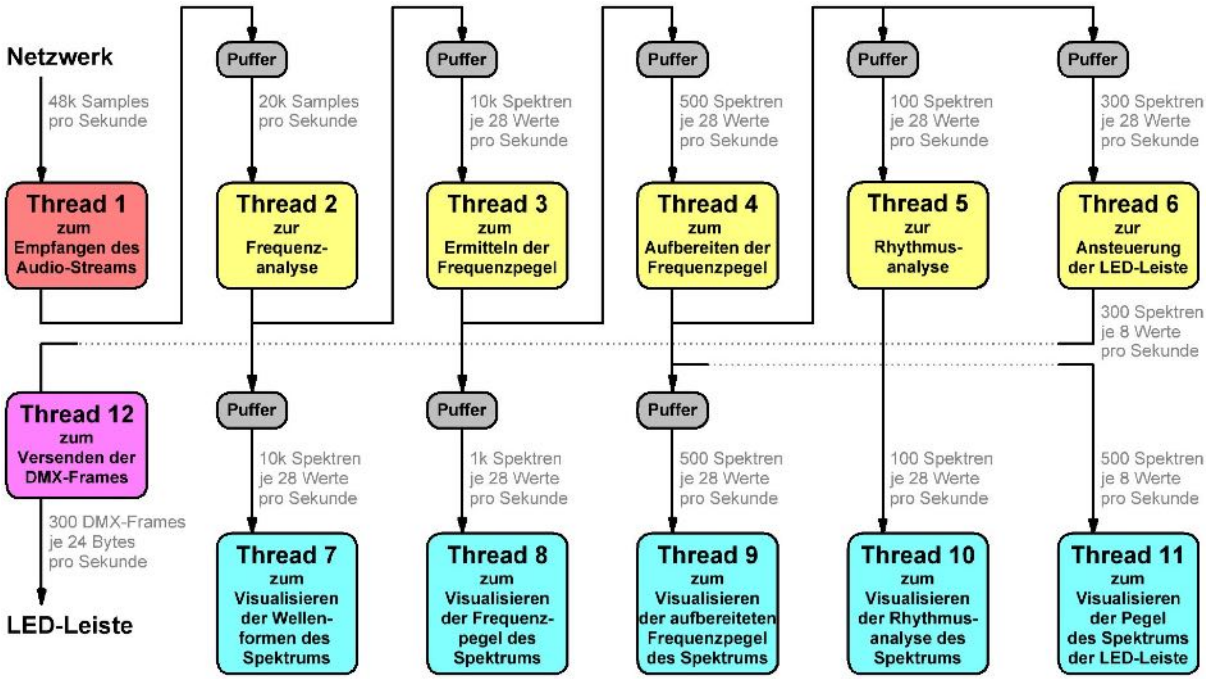


Abb. 5-12: Datenfluss der Signalverarbeitung durch Threads und Puffer

Aufgrund der parallel laufenden Threads wird die zeitliche Verzögerung des Signals minimal gehalten und nicht von den Threads der Visualisierungen beeinflusst. Zudem konvertieren die Puffer die Datenraten der Signale auf ein erforderliches Minimum, um Rechenleistung zu sparen. Die Anwendung lastet so, trotz aufwändiger Berechnungen und Visualisierungen, eine Intel® Core™ i7-4910MQ CPU nur zu ca. 20% aus. Deaktiviert man alle Visualisierungen durch Minimieren des Fensters, halbiert sich dieser Wert. Im Arbeitsspeicher werden nur ca. 30 MB belegt, da nicht mehr benötigte Objekte sofort für die Speicherbereinigung freigegeben werden.

Auf die einzelnen Threads wird in den folgenden Abschnitten eingegangen.

Thread 1 - Empfangen des Audio-Streams

Der erste Thread empfängt die vom „Audio Streamer“ über Netzwerk gesendeten UDP-Pakete und gewinnt daraus das abgetastete Audio-Signal. Hierzu werden die Datenpakete zunächst entpackt, deserialisiert und anschließend die Richtigkeit der chronologischen Reihenfolge anhand der beigefügten Paket-ID überprüft. Dies ist wichtig, um eine Fehlermeldung ausgeben zu können, wenn mehrere „Audio Streamer“ gleichzeitig Datenpakete an denselben Port senden.

Die Datenrate des Audio-Signals beträgt 48.000 Werte pro Sekunde und wird im nachgeschalteten Puffer für die weitere Verarbeitung auf 20.000 Werte pro Sekunde reduziert.

Thread 2 - Frequenzanalyse

Die Frequenzanalyse zerlegt das Audio-Signal in seine Frequenzanteile. Hierzu werden 28 Bandpässe der 2. Ordnung eingesetzt, deren Grenzfrequenzen sich aneinanderliegend und logarithmisch verteilt über einen Spektralbereich von 20 - 55370 Hz erstrecken. Gegenüber einer zu schnellen Fourier-Transformation (FFT) hat diese Variante der Frequenzanalyse den großen Vorteil, dass keine breiten Zeitfenster betrachtet werden müssen und somit Verzögerungen des Signals vermieden werden. Allerdings ist für die Berechnung von 28 Bandpässen gegenüber einer FFT wesentlich mehr Rechenleistung erforderlich, die aber ein moderner Prozessor ohne weiteres zur Verfügung stellt.

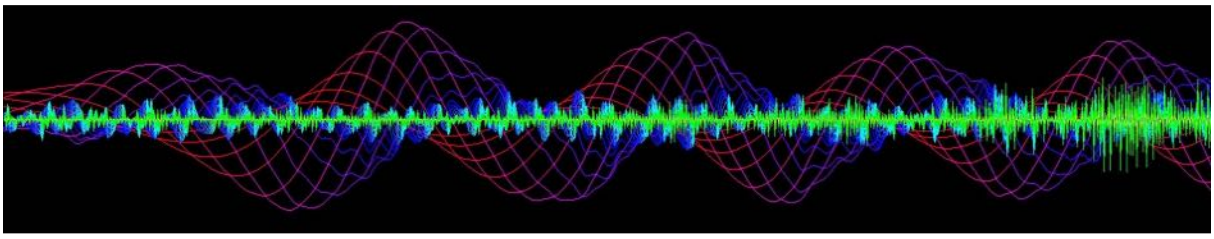


Abb. 5-13: Wellenformen des Spektrums in Abhängigkeit der Zeit

Nachdem das Audiosignal in seine Frequenzanteile zerlegt wurde, liegen die einzelnen Frequenzpegel weit unter dem Gesamtpegel des ursprünglichen Audio-Signals. Des Weiteren sind die hohen Frequenzpegel im Vergleich zu den niedrigen für die weitere Verarbeitung zu klein. Deshalb werden zunächst die Frequenzanteile logarithmisch abhängig von ihrer Frequenz mit einem Faktor von $3\text{^}(f/55370)$ verstärkt. Um anschließend die Frequenzpegel zu normalisieren, wird zudem, ähnlich wie beim „Audio Streamer“, eine automatische Lautstärke-Regelung (Seite 190) durchgeführt, die sich hier aber immer am aktuell größten Frequenzpegel des Spektrums orientiert. Die Pegelgrenze der automatischen Lautstärke-Regelung liegt hier bei 85%, die Steigungsbegrenzungen bei 400% pro Sekunde für das Verringern des Verstärkungsfaktors und für das Erhöhen bei 0,25% pro Sekunde.

Die Datenrate des Audio-Signals beträgt nach der Frequenzanalyse 20.000 Spektren mit je 28 Werten pro Sekunde und wird im nachgeschalteten Puffer auf 10.000 verringert.

Thread 3 - Ermitteln der Frequenzpegel

Nach der Zerlegung des Audio-Signals in seine Frequenzanteile werden nun die Pegel der einzelnen Wellenformen durch Gleichrichten und Glätten der Signale ermittelt. Hierzu wird der Betrag der Signalwerte berechnet und mit einem Verzögerungsglied der 1. Ordnung geglättet. Der hierfür eingesetzte Funktionsbaustein arbeitet, solange der Eingangswert kleiner als der Ausgangswert ist, wie ein normales T1-Glied. Ist der Wert am Ausgang allerdings größer als der des Eingangs, so wird der Eingangswert unverändert übernommen. Die Zeitkonstanten der Verzögerungsglieder wurden abhängig zur Frequenz mit $0{,}1s \cdot (f/55370)$ gewählt.

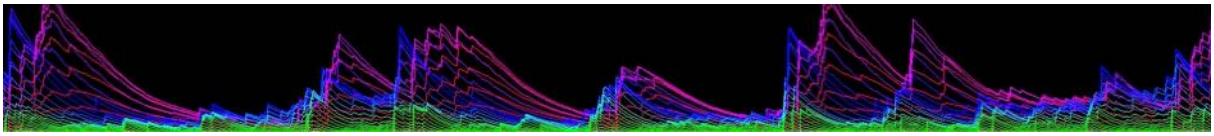


Abb. 5-14: Frequenzpegel des Spektrums in Abhängigkeit der Zeit

Für die folgenden Schritte der Signalverarbeitung beschreibt das nun vorliegende Signal mit einer Datenrate von 10.000 Spektren, mit je 28 Werten pro Sekunde, wie stark die einzelnen Frequenzbereiche des Spektrums im Audio-Signal vertreten sind. Da in diesem Signal die Information der Phasenlage nicht mehr enthalten ist, kann der nachgeschaltete Puffer die Datenrate ohne große Informationsverluste auf 500 Spektren pro Sekunde verringern.

Thread 4 - Aufbereiten der Frequenzpegel

Für die weitere Signalverarbeitung wirkt es sich positiv aus, wenn der Verlauf der Frequenzpegel möglichst glatt und die Dynamik dabei möglichst hoch ist. Deshalb werden für die Aufbereitung der Signale Bandpässe eingesetzt, die mit ihrer oberen Grenzfrequenz von 10 Hz durch das Gleichrichten entstandene Rippel entfernen und infolge ihrer unteren Grenzfrequenz langanhaltende Offsets unterbinden und so die Dynamik des Signals erhöhen.

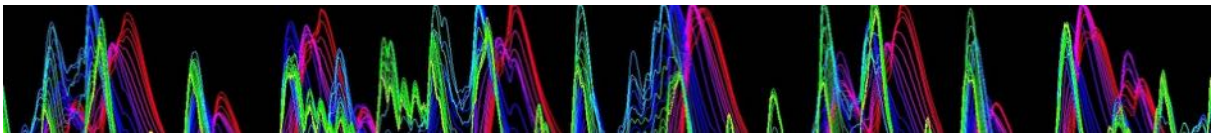


Abb. 5-15: Aufbereitete Frequenzpegel des Spektrums in Abhängigkeit der Zeit

Bevor der Thread die aufbereiteten Frequenzpegel an die Rhythmus-Analyse weitergibt, wird ein weiteres Mal eine Normalisierung der Signale durchgeführt, um die durch Bandpässe entstandenen Dämpfungen auszugleichen. Die einzelnen Frequenzpegel werden hierbei getrennt voneinander angeglichen, wobei die Zeitkonstante einheitlich 20 Sekunden beträgt und der Verstärkungsfaktor auf den Faktor $1/0.02$ begrenzt ist, um das Rauschen einer Nulllinie nicht in den für die LED-Leiste relevanten Pegelbereich zu skalieren.

Für die folgende Rhythmusanalyse kann die Datenrate des aufbereiteten Signals aufgrund der Glättung im nachgeschalteten Puffer auf 100 Spektren pro Sekunde reduziert werden.

Thread 5 - Rhythmusanalyse

Am Eingang der Rhythmusanalyse liegt das aufbereitete Spektrum des Audio-Signals an. Die Aufgabe der Rhythmusanalyse ist es nun, aus diesem Signal wiederkehrende Tonhöhen zu filtern und dabei nicht wiederkehrende Töne zu ignorieren. Das Ausgangssignal der Analyse ist in der folgenden Abbildung dargestellt und beschreibt, wie stark rhythmische Töne in den einzelnen Frequenzbereichen gefunden wurden. Die horizontale Achse des Diagramms gibt dabei den Taktabstand und die vertikale Achse die Intensität des wiederkehrenden Tons an. Die Farbe beschreibt dazu das Frequenzband, in dem der rhythmische Ton gefunden wurde.

Aus dem Diagramm lässt sich zum Beispiel ablesen, dass der Bass des Testsignals im Taktabstand gespielt wird und sich dazu im oberen Frequenzbereich Töne mit dem halben und viertel Taktabstand überlagern.

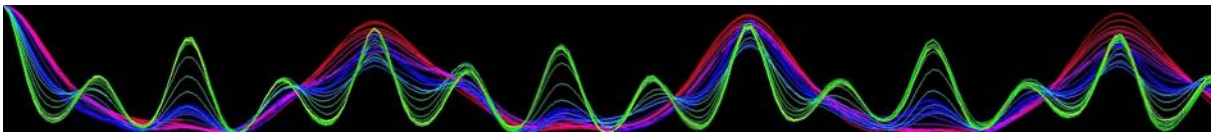


Abb. 5-16: Intensität der rhythmischen Tonhöhen in Abhängigkeit des Taktabstands

Der Algorithmus zur Rhythmusanalyse arbeitet wie folgt: Zuerst wird das Spektrum am Eingang der Analyse in 200 Puffern gespeichert und mit unterschiedlichen Pufferzeiten verzögert. Der erste Puffer verzögert das Spektrum um 10 Millisekunden, der zweite Puffer um 20 Millisekunden und so weiter. Anschließend werden die Frequenzpegel der 200 zeitlich versetzten Spektren mit den Frequenzpegeln des aktuellen Eingangsspektrums multipliziert. Der Betrag des Produkts sagt nun aus, wie gut die Frequenzpegel der verzögerten Spektren zu den Frequenzpegeln des aktuellen Spektrums passen. Bevor die Visualisierung das Ergebnis darstellt, wird es von Verzögerungsgliedern der 2. Ordnung mit einer Zeitkonstante von 0.2 Sekunden geglättet und zudem normalisiert. Die Graphen beschreiben nicht den zeitlichen Verlauf, sondern stellen, abhängig von der Frequenz und dem zeitlichen Versatz, die Übereinstimmung der aktuellen Frequenzpegel zu den zeitlich versetzten Frequenzpegeln dar.

Wie bereits in der Beschreibung des Programms „DJ JOE Genius“ erwähnt, soll im Rahmen eines folgenden Projekts die Rhythmusanalyse mit in die Genius-Suche implementiert werden, um diese durch die Berücksichtigung der Rhythmen beim Vergleich der Titel zu verbessern.

Thread 6 - Ansteuerung der LED-Leiste

Der letzte Thread bereitet das Spektrum für die Ausgabe mit der LED-Leiste vor. Hierzu wird das Spektrum, das 28 Frequenzpegel umfasst, auf eine Bandbreite von 8 Werten umgerechnet und eine Skalierung (180%) mit Offset (40%) und Gammawertkorrektur (γ=0,8) durchgeführt.

Abb. 5-17: Spektrum mit 28 Frequenzpegeln

Thread 7 bis 11 - Visualisierungen

Alle Visualisierungs-Threads haben die gleiche Aufgabe, Spektren aus dem vorgeschalteten Puffer abzuholen, deren Frequenzpegel durch Rendern einer Grafik visuell darzustellen und dabei die Framerate zu begrenzen. Jedoch unterscheiden sich die Threads aufgrund der unterschiedlichen Darstellungsarten in der Art und Weise, wie sie die Grafiken rendern.

Der Thread zum Visualisieren der Wellenformen des Spektrums triggert zum Beispiel das Signal auf eine positive Flanke der kleinsten Frequenz des Spektrums und gibt die Grafik erst dann aus, wenn sich genügend Spektren im vorgeschalteten Puffer angesammelt haben, um die Wellenformen über die komplette Fensterbreite zu rendern. Eine Pixelreihe in vertikaler Richtung repräsentiert hierbei die 28 Werte eines Spektrums.

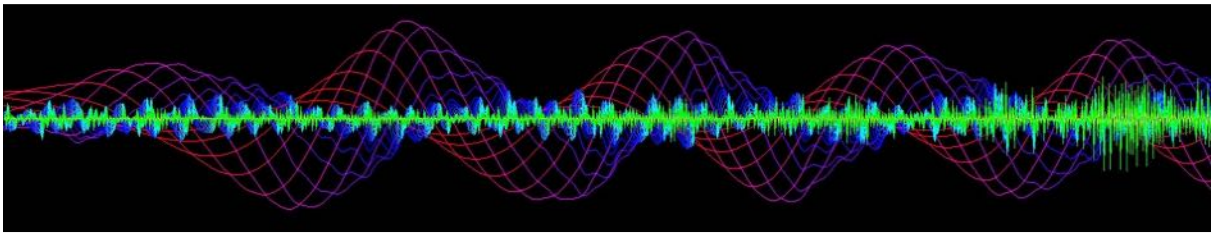


Abb. 5-18: Visualisierung der Wellenformen des Spektrums in Abhängigkeit der Zeit

Die Threads zur Visualisierung der Frequenzpegel beginnen hingegen mit dem Rendern, sobald sich neue Spektren im Puffer befinden und überschreiben damit die ältesten.

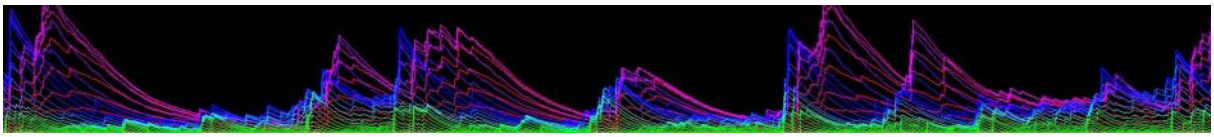


Abb. 5-19: Visualisierung der Frequenzpegel des Spektrums in Abhängigkeit der Zeit

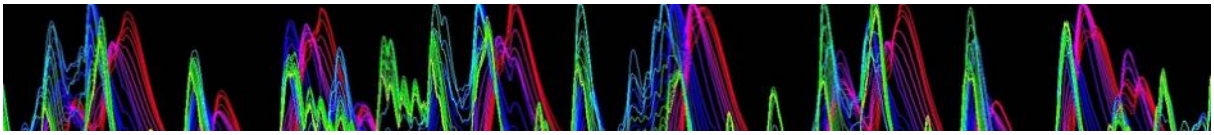


Abb. 5-20: Visualisierung der aufbereiteten Frequenzpegel in Abhängigkeit der Zeit

Des Weiteren wird bei der Visualisierung der Rhythmusanalyse ohne vorgeschalteten Puffer mit jedem Frame der aktuelle Zustand der Analyse gerendert und dargestellt.

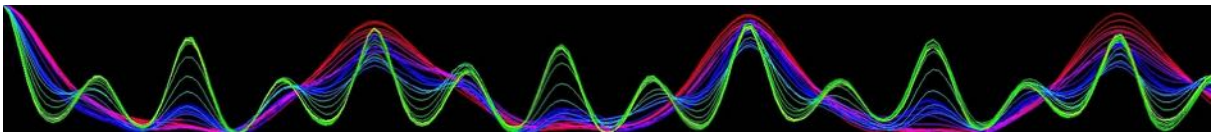


Abb. 5-21: Visualisierung der Rhythmusintensität in Abhängigkeit des Taktabstands

Ebenso werden auch die aktuellen Frequenzpegel ohne Zwischenpuffer visualisiert.

Abb. 5-22: Visualisierung der 28 Frequenzpegel des Spektrums

Thread 12 - Versenden der DMX-Frames

Der letzte Thread kommuniziert mit dem selbst gebauten USB-zu-DMX-Konverter, um die im Thread 6 aufbereiteten Frequenzpegel des aktuellen Spektrums an die LED-Leiste auszugeben. Hierzu werden die DMX-Adressen 1 bis 24 genutzt, um 8 LED-Segmente mit je 3 Farbkanälen (RGB) anzusteuern. Der Frequenzpegel gibt hierbei die Helligkeit und dessen Frequenz den Farbton bei einer Sättigung von 100% vor.

5.3.3 Einsatz der Automatisierungsbibliothek

Bei der Umsetzung des letzten Softwaretools dieses Projekts konnten besonders viele Aufgaben mit Elementen der Automatisierungsbibliothek gelöst werden. Der Einsatz von zuvor programmierten Funktionsblöcken hat den Programmieraufwand erheblich gesenkt und den Programmcode vereinfacht.

Im Bereich der Signalverarbeitung konnte zum Beispiel auf einige Übertragungsglieder der Bibliothek zurückgegriffen werden. Mit Bandpässen wurden hier Signale gefiltert, um eine besonders flinke Frequenzanalyse durchzuführen oder Frequenzpegelverläufe für die weitere Verarbeitung von Rippeln und Offsets zu befreien. Verzögerungsglieder fanden hingegen ihren Einsatz beim Glätten von Signalverläufen oder in Kombination mit wertebegrenzenden Funktionsblöcken bei der Normalisierung von Signalen und Spektren. Auch sehr grundlegende Funktionsblöcke zum Ermitteln des Betrages oder Vorzeichens wurden aus der Bibliothek für das Gleichrichten des Audio-Signals eingesetzt, da diese im Gegensatz zu den mathematischen .NET-Funktionen auch besondere Double-Werte (z. B. double.NaN) verarbeiten können.

Für den Bereich Visualisierung konnte die Automatisierungsbibliothek mit Zeitgliedern und Flankenerkennungen dienen. Beispielsweise wurde für die Begrenzung der Aktualisierungsrate eine Abfallverzögerung eingesetzt und die Wellenformen des Audio-Signals mit Hilfe einer positiven Flankenerkennung getriggert.

Die Automatisierungsbibliothek stellte bei der Programmierung der Software darüber hinaus auch Funktionsblöcke bereit, die sehr gut für Debugging-Zwecke genutzt werden konnten. Zum Beispiel wurde mit der Stoppuhr der Bibliothek und einem nachgeschalteten Verzögerungsglied die durchschnittliche Prozessorzeit verschiedener Code-Abschnitte ermittelt, um diese zu optimieren. Des Weiteren war es mit den Signalgeneratoren und Filtern der Bibliothek möglich, in wenigen Code-Zeilen Testsignale, wie zum Beispiel weißes oder rosa Rauschen, zu erzeugen, um die Funktionalität des Programms zu testen.

6 Ergebnis

Das Ziel dieser Masterarbeit war es, ein umfassendes Werkzeug zu schaffen, mit dem sehr spezielle und komplexe Problemstellungen der Automatisierungstechnik gelöst werden können. Ganz konkret sollte eine Umgebung geschaffen werden, mit der die Steuerungssoftware von computerbasierten Anlagensteuerungen für besonders hohe Anforderungen erstellt werden kann. Diese Zielsetzung konnte durch die Entwicklung einer universellen Automatisierungsbibliothek erfüllt werden, die das umfangreiche .NET-Framework von Microsoft um grundlegende Funktionen der Steuerungs- und Regelungstechnik erweitert und so den effektiven Einsatz der Programmiersprache VB.NET für automatisierungstechnische Anwendungen ermöglicht. Die erstellte Bibliothek setzt sich aus insgesamt 50 Funktionsblöcken zusammen, die in dieser Arbeit ausführlich dokumentiert wurden.

Zum Erstellen der Funktionsblöcke wurde im Rahmen der Masterarbeit eine umfangreiche Entwicklungs- und Testumgebung programmiert, die speziell für das Entwickeln und Testen von Funktionsblöcken ausgelegt ist. Hiermit konnten bereits in der Entwicklungsphase alle Funktionsbausteine der Bibliothek umfassend getestet und aus den Testergebnissen die Diagramme für die Dokumentation generiert werden. Auch bei folgenden Projekten stellt die Entwicklungs- und Testumgebung ein sehr nützliches Werkzeug dar, mit dem Programmcode komfortabel getestet werden und die Automatisierungsbibliothek um eigene Funktionsblöcke erweitert werden kann.

Um die Vorteile und Möglichkeiten der durch die Bibliothek geschaffenen Umgebung zu veranschaulichen, wurde ein abschließendes Beispielprojekt mit dem Visual Studio und VB.NET umgesetzt. Durch den Einsatz vieler Elemente der Automatisierungsbibliothek wurden hierbei drei sehr unterschiedliche Programme erstellt, um das breite Spektrum an Möglichkeiten des geschaffenen Automatisierungswerkzeugs zu präsentieren und aufzuzeigen, wie universell und vielseitig die erstellte Bibliothek eingesetzt werden kann. Als konkretes Anwendungsbeispiel wurde unter Benutzung der Bibliothekfunktionalitäten ein zeitkritisches Audiosignal eingelesen, per Audio-Stream über das Netzwerk übertragen und mit Multithreading sehr aufwändig verarbeitet, um letztendlich Lichteffektgeräte über das DMX-Protokoll anzusteuern.

Abschließend lässt sich festhalten, dass das Masterarbeitsprojekt nicht nur hinsichtlich der erfüllten Zielsetzung ein voller Erfolg war, sondern auch im Zuge dessen die Idee zur Gründung eines Ingenieurbüros, das individuelle Steuerungssoftware für Anlagen entwickelt, geboren wurde. Hierzu soll die in der Masterarbeit programmierte Bibliothek genutzt werden, um das Visual Studio und die Programmiersprache VB.NET zum Lösen sehr spezieller Problemstellungen der Steuerungs- und Automatisierungstechnik einzusetzen.

Literaturverzeichnis

Diplomarbeit - Automatisiertes Bewickeln von Gummikompensatoren / Verf. Glaser Johannes. - Nordheim : [s.n.], 2014. - S. 240.

Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software [Buch] / Verf. Gamma Erich [et al.]. - Hallbergmoos : Pearson Deutschland GmbH, 2011. - 6.

Fachkunde Elektrotechnik [Buch] / Verf. Tkotz Klaus [et al.]. - Haan-Gruiten : Europa-Lehrmittel, 2014. - 29.

Halbleiter-Schaltungstechnik [Buch] / Verf. Tietze Ulrich, Schenk Christoph und Gamm Eberhard. - Erlangen und München : Springer, August 2012. - 14.

Mathematische Formelsammlung: für Ingenieure und Naturwissenschaftler [Buch] / Verf. Papula Lothar. - Wiesbaden : Vieweg+Teubner Verlag | GWV Fachverlage, 2009. - 10.

Regelungstechnik I: Klassische Verfahren zur Analyse und Synthese linearer kontinuierlicher Regelsysteme, Fuzzy-Regelsysteme [Buch] / Verf. Unbehauen Heinz. - Wiesbaden : Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH, 2007. - 14.

Regelungstechnik II: Zustandsregelungen, digitale und nichtlineare Regelsysteme [Buch] / Verf. Unbehauen Heinz. - Wiesbaden : Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH, 2007. - 9.

Tabellenbuch Elektrotechnik XXL [Buch] / Verf. Tkotz Klaus [et al.]. - Haan-Gruiten : Europa-Lehrmittel, 2014. - 3.

Visual Basic .Net Praxisorientiert: Datenbanken und Web-Anwendungen [Buch] / Verf. Wirtz Klaus Werner. - Norderstedt : Books on Demand, 2010. - Bd. 2.

Visual Basic .Net Praxisorientiert: Programmieren für Windows [Buch] / Verf. Wirtz Klaus Werner. - Norderstedt : Books on Demand, 2010. - Bd. 1.

Abbildungsverzeichnis

Abb. 1-1: Verschiedene Anlagentypen 7
Abb. 1-2: Selbsthalteschaltung, umgesetzt in FUP, KOP, AWL und ST 8
Abb. 1-3: Blinkende LED, umgesetzt in C, C++ und Java 9
Abb. 1-4: Kaffeeautomat, umgesetzt in LabVIEW 10
Abb. 1-5: Einfache Autosimulation, umgesetzt in VB.NET und C# 11
Abb. 1-6: Aufbauschema einer klassischen Automatisierungssteuerung 12
Abb. 1-7: Aufbauschema einer modernen Automatisierungssteuerung 13
Abb. 1-8: Visuelle Darstellung der Idee 14
Abb. 1-9: 3D-Modell der Wickelmaschine 15
Abb. 1-10: Berechnetes Wickelbild 16
Abb. 1-11: Schematischer Aufbau der Anlagensteuerung 16
Abb. 1-12: Editieren von Programmcode während der Laufzeit 17
Abb. 1-13: Auslastung der CPU-Kerne in Abhängigkeit der Zeit 17
Abb. 1-14: Benutzeroberfläche der Steuerungssoftware 18
Abb. 1-15: Aufbau der Kugelwaage 19
Abb. 1-16: Funktionsblockeditor zum Regeln der Kugelwaage 20
Abb. 1-17: Symbol des „T2S“-Funktionsblocks 22
Abb. 1-18: LTspice-Simulation eines Reihenschwingkreises 22
Abb. 1-19: Bildschirmaufnahme der Funktionsblock-Testumgebung 24
Abb. 1-20: Elektronischer Schwingkreis 28
Abb. 1-21: Schwingende Masse 28
Abb. 2-1: Aufbau der Automatisierungsbibliothek 29
Abb. 2-2: Klassenstruktur aller Basisklassen 31
Abb. 2-3: Signalflussdiagramm der Beispiel-Anwendung 36
Abb. 3-1: Funktionsblöcke des Namensbereichs „Controller“ 42
Abb. 3-2: Funktionsblöcke des Namensbereichs „Generator“ 42
Abb. 3-3: Funktionsblöcke des Namensbereichs „Timer“ 42
Abb. 3-4: Funktionsblöcke des Namensbereichs „Flank“ 42
Abb. 3-5: Funktionsblöcke des Namensbereichs „Analog“ 43
Abb. 3-6: Funktionsblöcke des Namensbereichs „Generic“ 43
Abb. 3-7: Funktionsblöcke des Namensbereichs „FuncSegment“ 43
Abb. 3-8: Funktionsblöcke des Namensbereichs „JustInTime“ 43
Abb. 3-9: Diskrete Zykluszeiten und Werte 46
Abb. 3-10: Äquidistante Zykluszeiten und Werte 46
Abb. 3-11: Transformationen zum Berechnen einer Differenzengleichung 47

Abb. 3-12: Funktionsblock-Testergebnis: Proportionalglied 49
Abb. 3-13: Funktionsblock-Testergebnis: Integrationsglied 51
Abb. 3-14: Funktionsblock-Testergebnis: Differenzierer 53
Abb. 3-15: Funktionsblock-Testergebnis: „PIDT1“-Übertragungsglied 55
Abb. 3-16: Parallelstruktur und Sprungantwort des „PIDT1“-Übertragungsglieds 56
Abb. 3-17: Funktionsblock-Testergebnis: Verzögerungsglied 1. Ordnung 59
Abb. 3-18: Funktionsblock-Testergebnis: Verzögerungsglied 2. Ordnung 61
Abb. 3-19: Funktionsblock-Testergebnis: Verzögernder Differenzierer 63
Abb. 3-20: Funktionsblock-Testergebnis: Fallschirm 65
Abb. 3-21: Funktionsblock-Testergebnis: Bandpass 67
Abb. 3-22: Funktionsblock-Testergebnis: Bandpass höherer Ordnung 69
Abb. 3-23: Funktionsblock-Testergebnis: Totzeitglied 71
Abb. 3-24: Funktionsblock-Testergebnis: Steigungsbegrenzer 73
Abb. 3-25: Funktionsblock-Testergebnis: Signalgenerator (Approximationstypen) 76
Abb. 3-26: Funktionsblock-Testergebnis: Sägezahn-Generator 77
Abb. 3-27: Funktionsblock-Testergebnis: Dreieck-Generator 79
Abb. 3-28: Funktionsblock-Testergebnis: PWM-Generator 81
Abb. 3-29: Funktionsblock-Testergebnis: Sinus-Generator 83
Abb. 3-30: Funktionsblock-Testergebnis: Anzugsverzögerung 87
Abb. 3-31: Funktionsblock-Testergebnis: Abfallverzögerung 89
Abb. 3-32: Funktionsblock-Testergebnis: Abfallverzögerung 91
Abb. 3-33: Funktionsblock-Testergebnis: Stoppuhr 93
Abb. 3-34: Funktionsblock-Testergebnis: Positive Flankenerkennung 97
Abb. 3-35: Funktionsblock-Testergebnis: Negative Flankenerkennung 99
Abb. 3-36: Funktionsblock-Testergebnis: Flankenerkennung 101
Abb. 3-37: Funktionsblock-Testergebnis: Toggle-Flipflop 103
Abb. 3-38: Funktionsblock-Testergebnis: Wertänderung 107
Abb. 3-39: Funktionsblock-Testergebnis: Hysterese 109
Abb. 3-40: Funktionsblock-Testergebnis: Zufallsgenerator 111
Abb. 3-41: Auswirkung des Parameters „RndCount“ auf die Wahrscheinlichkeitsverteilung 112
Abb. 3-42: Funktionsblock-Testergebnis: Zykluszeit 113
Abb. 3-43: Funktionsblock-Testergebnis: Betrag 117
Abb. 3-44: Funktionsblock-Testergebnis: Vorzeichen 119
Abb. 3-45: Funktionsblock-Testergebnis: Modulo1 121
Abb. 3-46: Funktionsblock-Testergebnis: Modulo2 123
Abb. 3-47: Funktionsblock-Testergebnis: Begrenzer 125

Abb. 3-48: Funktionsblock-Testergebnis: Gültigkeitsbereich 127
Abb. 3-49: Funktionsblock-Testergebnis: AntiNaN 129
Abb. 3-50: Beispiel für eine abschnittsweise definierte Funktion 131
Abb. 3-51: Funktionsblock-Testergebnis: Abschnittsweise definierte Funktion 131
Abb. 3-52: Funktionsblock-Testergebnis: Konstante Funktion 135
Abb. 3-53: Funktionsblock-Testergebnis: Lineare Funktion 137
Abb. 3-54: Funktionsblock-Testergebnis: Stufenfunktion 139
Abb. 3-55: Funktionsblock-Testergebnis: Zufallsfunktion 141
Abb. 3-56: Funktionsblock-Testergebnis: Potenzfunktion 143
Abb. 3-57: Funktionsblock-Testergebnis: Logarithmische Funktion 145
Abb. 3-58: Funktionsblock-Testergebnis: Rechteck-Funktion 147
Abb. 3-59: Funktionsblock-Testergebnis: PWM-Funktion 149
Abb. 3-60: Funktionsblock-Testergebnis: Sägezahn-Funktion 151
Abb. 3-61: Funktionsblock-Testergebnis: Dreieck-Funktion 153
Abb. 3-62: Funktionsblock-Testergebnis: Sinus-Funktion 155
Abb. 3-63: Funktionsblock-Testergebnis: Cosinus-Funktion 157
Abb. 3-64: Funktionsblock-Testergebnis: Funktionsblock-Vorlage 161
Abb. 3-65: Funktionsblock-Testergebnis: Erweiterte Funktionsblock-Vorlage 163
Abb. 4-1: Benutzeroberfläche der Testumgebung 165
Abb. 4-2: Signale zur zeitlichen Taktung der Funktionsaufrufe 168
Abb. 4-3: Ein- und Ausgangssignale des „T2S“-Funktionsbausteins 169
Abb. 4-4: Quantisierungsfehler der Ausgangssignale 170
Abb. 4-5: Parametersignale 170
Abb. 4-6: Tabellenansichten zum Bearbeiten von Funktionsblock-Tests 171
Abb. 4-7: Dropdown-Listenfeld „Datei“ 171
Abb. 4-8: Angezeigter Name der Projektdatei „BeispielProjekt.xml“ 171
Abb. 4-9: Darstellung der XML-Datei als Tabelle, Baumstruktur oder Text 172
Abb. 4-10: Grober Aufbau der XML-Struktur 172
Abb. 4-11: Dropdown-Listenfeld „Bearbeiten“ 172
Abb. 4-12: Tabelle der Funktionsblock-Tests 173
Abb. 4-13: Explorer-Ansicht des Unterverzeichnisses „Images“ 173
Abb. 4-14: Tabelle der Testsignale 174
Abb. 4-15: Signal-Farbpalette 174
Abb. 4-16: Tabelle der Funktions-Segmente 175
Abb. 4-17: Beispiel eines Eingangssignals „Input“ und dazugehörige Ausgangssignale 175
Abb. 4-18: Funktionssegmente zum Erstellen von Testsignalen 175

Abb. 4-19: Tabelle der Parameter ... 176
Abb. 4-20: Registerkarte „Schaubild“ ... 177
Abb. 4-21: Regel für Wertesprünge der Signale ... 177
Abb. 4-22: Symbolleiste des Schaubilds ... 178
Abb. 4-23: Kontextmenü des Schaubilds ... 178
Abb. 4-24: Signalverläufe mit Skalier-Modus „Automatisch mit Begrenzung“ ... 179
Abb. 4-25: Signalverläufe mit Skalier-Modus „Automatisch“ ... 179
Abb. 4-26: Darstellungs-Modus „2D-Ansicht“ ... 180
Abb. 4-27: Darstellungs-Modus „3D-Ansicht“ ... 180
Abb. 4-28: Dropdown-Listenfeld „Datei“ ... 181
Abb. 4-29: Dialogfeld „Drucken“ ... 181
Abb. 4-30: Fenster „Seitenansicht“ ... 181
Abb. 4-31: Registerkarte „Wertetabelle“ ... 182
Abb. 4-32: Dropdown-Listenfeld zum Auswählen des Anzeigemodus der Wertetabelle ... 182
Abb. 4-33: Code-Editor der integrierten Entwicklungsumgebung ... 183
Abb. 4-34: Statusleiste ... 184
Abb. 5-1: Testaufbau des Beispielprojekts ... 185
Abb. 5-2: Benutzeroberfläche des Programms „DJ JOE Genius“ ... 186
Abb. 5-3: Programm DJ-JOE-Genius ... 187
Abb. 5-4: Benutzeroberfläche des Programms „Audio Streamer“ ... 188
Abb. 5-5: Aufnahmegeräte (rot) und automatische Lautstärke-Regelung (orange) ... 188
Abb. 5-6: Puffer und Threads des Audio-Streamers ... 189
Abb. 5-7: Programmcode zur Implementierung der automatischen Lautstärke-Regelung ... 190
Abb. 5-8: Blockschaltbild der automatischen Lautstärkeregelung ... 190
Abb. 5-9: Diagramm zur automatischen Lautstärke-Regelung ... 190
Abb. 5-10: Benutzeroberfläche des Programms „Audio Analyzer“ ... 191
Abb. 5-11: Zuordnung der Linienfarben und Grenzfrequenzen ... 191
Abb. 5-12: Datenfluss der Signalverarbeitung durch Threads und Puffer ... 192
Abb. 5-13: Wellenformen des Spektrums in Abhängigkeit der Zeit ... 193
Abb. 5-14: Frequenzpegel des Spektrums in Abhängigkeit der Zeit ... 194
Abb. 5-15: Aufbereitete Frequenzpegel des Spektrums in Abhängigkeit der Zeit ... 194
Abb. 5-16: Intensität der rhythmischen Tonhöhen in Abhängigkeit des Taktabstands ... 195
Abb. 5-17: Spektrum mit 28 Frequenzpegeln ... 195
Abb. 5-18: Visualisierung der Wellenformen des Spektrums in Abhängigkeit der Zeit ... 196
Abb. 5-19: Visualisierung der Frequenzpegel des Spektrums in Abhängigkeit der Zeit ... 196
Abb. 5-20: Visualisierung der aufbereiteten Frequenzpegel in Abhängigkeit der Zeit ... 196

Abb. 5-21: Visualisierung der Rhythmusintensität in Abhängigkeit des Taktabstands..... 196
Abb. 5-22: Visualisierung der 28 Frequenzpegel des Spektrums 196

Selbstständigkeitserklärung

Die vorliegende Abschlussarbeit wurde von mir selbstständig verfasst und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt. Wörtliche und sinngemäße Zitate im Text sind als solche gekennzeichnet.

Nordheim, den 30.09.2015

Dipl.-Ing. (FH) Johannes Glaser
Masterand

Anhang

Programmcode-Verzeichnis

(Namensbereiche, Funktionsblock-Klassen, Dienstfunktionen, Module und Oberklassen)

Namespace AutomationLibrary 208
Namespace BaseClass 209
Namespace Controller 211
Namespace Flank 222
Namespace FuncSegment 226
Namespace Generator 217
Namespace Generic 224
Namespace JustInTime 228
Namespace Timer 219
Public Class Abs 224
Public Class AntiNaN 225
Public Class Any 222
Public Class Bandpass 214
Public Class BandpassX 215
Public Class Change 222
Public Class Constant 226
Public Class Cos 228
Public Class D 211
Public Class DeadTime 216
Public Class DeltaT 223
Public Class DT1 213
Public Class FuncOfSegments 225
Public Class Hi 222
Public Class Hysteresis 222
Public Class I 211
Public Class Limit 224
Public Class Linear 226
Public Class Lo 222
Public Class Log 227
Public Class Mod1 224
Public Class Mod2 224
Public Class P 211
Public Class Parachute 214
Public Class PIDT1 212

Public Class Pow 227
Public Class Puls 220
Public Class PWM 218, 227
Public Class Random 223, 226
Public Class Rectangle 227
Public Class Relay 222
Public Class Sawtooth 217, 228
Public Class Sign 224
Public Class Sin 219, 228
Public Class SlopeLimit 217
Public Class Steps 226
Public Class StopWatch 221
Public Class T1 212
Public Class T2S 213
Public Class Template 229
Public Class TemplateX 229
Public Class Triangle 218, 228
Public Class TurnOffDelay 220
Public Class TurnOnDelay 219
Public Class ValidRange 225
Public Function BooleanToDouble 209
Public Function CallInst 208
Public Function DetectAnyFlank 209
Public Function DetectHiFlank 209
Public Function DetectLoFlank 209
Public Function DoubleToBoolean 209
Public Module Utility 208
Public MustInherit Class Func 209
Public MustInherit Class FuncRetrospective 209
Public MustInherit Class FuncRetrospectiveTimeDependent 210
Public MustInherit Class FuncSegment 210
Public MustInherit Class FuncSegmentPeriodic 210

Programmcode der Automatisierungsbibliothek

```
'Autor: Dipl.-Ing.(FH) Johannes Glaser
'Datum: 2015-09-26 (Letzte Änderung)

Option Explicit On 'Explizite Deklaration erzwingen
Option Strict On 'Datenkonvertierungen erzwingen

'-----------------------------------------------------------------------------------------------------'
'Automatisierungsbibliothek
Namespace AutomationLibrary

'-----------------------------------------------------------------------------------------------------'
'Dienstfunktionen
Public Module Utility

'Ruft die Instanz eines Funktionsblocks anhand seines Typs und einer Id auf.
Public Function CallInst(ByVal TypeKey As Type, ByVal Id As Object, ByVal Input As Double,
                         ByVal Paramter As String, Optional DeltaT As Double = Double.NaN) As Double
    'Beispielaufruf:
    'TrackBar2.Value = CallInst(GetType(Controller.T2S), "123", TrackBar1.Value, "w_0=1,d=0.2", 0.01)
    'TypeKey:   Funktionsblock-Typ               z. B. GetType(Controller.T2S)
    'Id:        Id zum Zuordnen der Instanzen    z. B. "123"
    'Input:     Eingangswert des Funktionsblocks z. B. 5.4
    'Parameter: Parameter als String             z. B. "w_0=1,d=0.2"
    'DeltaT:    Nur bei zeitabh. Funktionsblöcken z. B. 0.01
    'return:    Ausgangswert des Funktionsblocks z. B. 1.8

    'Die Instanzen der Funktionsblöcke werden in zwei ineinander verschachtelten Verzeichnissen
    '(Dictionarys) gespeichert. Das Äußere ordnet den Funktionsblock-Typ und das Innere die Id zu.

    'Instanz des äußeren Verzeichnisses erstellen
    Static TypeDictionarys As Dictionary(Of Type, Dictionary(Of Object, BaseClass.Func))
    If TypeDictionarys Is Nothing Then
        TypeDictionarys = New Dictionary(Of Type, Dictionary(Of Object, BaseClass.Func))
    End If
    'Für jeden Funktionsblock-Typ ein eigenes Unterverzeichnis anlegen
    If Not TypeDictionarys.ContainsKey(TypeKey) Then 'Unterverzeichnis noch nicht vorhanden?
        TypeDictionarys.Add(TypeKey, New Dictionary(Of Object, BaseClass.Func))
    End If
    'Funktionsblock-Verzeichnis mit passendem Funktionsblock-Typ suchen
    Dim IdDictionary As Dictionary(Of Object, BaseClass.Func) = TypeDictionarys(TypeKey)
    'Für jede Id eine eigene Funktionsblock-Instanz erstellen.
    If Not IdDictionary.ContainsKey(Id) Then 'Instanz zu dieser Id noch nicht vorhanden?
        IdDictionary.Add(Id, DirectCast(Activator.CreateInstance(TypeKey), BaseClass.Func))
    End If
    'Funktionsblock-Instanz mit passender Id suchen
    Dim Instance As BaseClass.Func = IdDictionary(Id)
    'Parameter des Funktionsblocks aktualisieren
    Dim Parameter As String() = Paramter.Replace(" ", "").Split(","c) 'Parameter-Zeichenkette teilen
    For Each ParameterStr As String In Parameter 'Alle Parameter durchlaufen
        Dim ParameterElements As String() = ParameterStr.Split("="c) 'Name und Wert trennen
        Dim ParameterName As String = ParameterElements(0) 'Name
        Dim ParameterValue As Double = Double.Parse(ParameterElements(1).Replace(".", ",")) 'Wert
        'Parameter (Property) setzen (per Reflection)
        CallByName(Instance, ParameterName, CallType.Set, New Object() {ParameterValue})
    Next
    'Aufruf der Funktion Output
    Dim OutputValue As Double = Double.NaN 'Funktionsblock-Ausgangswert
    Dim FuncOutputParameter As Object() = {Input} 'Parameter der Funktion "Output"
    If TypeOf Instance Is BaseClass.FuncRetrospectiveTimeDependent Then
        FuncOutputParameter = {Input, DeltaT} 'Bei zeitabhängigen Funktionsblöcken DeltaT mit übergeben
    End If
    'Aufruf der Funktion "Output" (per Reflection)
    OutputValue = Convert.ToDouble(CallByName(Instance, "Output", CallType.Method,
                                   FuncOutputParameter))
    Return OutputValue 'Ausgangswert des Funktionsblocks zurückgeben

End Function
```

```
'Abbildung von Double auf Boolean
Public Function DoubleToBoolean(ByVal Value As Double) As Boolean
    '0 oder NaN -> False, Sonst -> True
    Return If(Value > 0 Or Value < 0, True, False)
End Function

'Abbildung von Boolean auf Double
Public Function BooleanToDouble(ByVal Value As Boolean) As Double
    'True -> 1, Sonst -> 0
    Return If(Value = True, 1, 0)
End Function

'Positive Flanke erkennen
Public Function DetectHiFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung in positiver Richtung -> True, Sonst -> False
    Return If(NewValue > LastValue, True, False)
End Function

'Negative Flanke erkennen
Public Function DetectLoFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung in negativer Richtung -> True, Sonst -> False
    Return If(NewValue < LastValue, True, False)
End Function

'Flanke erkennen
Public Function DetectAnyFlank(ByVal NewValue As Double, ByVal LastValue As Double) As Boolean
    'Wertänderung -> True, Sonst -> False
    Return If(NewValue > LastValue Or NewValue < LastValue, True, False)
End Function

End Module

'-----------------------------------------------------------------------------------------------------'
'Basisklassen der Funktionsblöcke
Namespace BaseClass

'-----------------------------------------------------------------------------------------------------'
'Funktionsblock, der über eine Ausgangsfunktion verfügt.
Public MustInherit Class Func

    'Ausgangsfunktion
    MustOverride Function Output(Input As Double) As Double

End Class

'-----------------------------------------------------------------------------------------------------'
'Funktionsblock, der sich auf vorherige Ein- und Ausgangswerte bezieht.
Public MustInherit Class FuncRetrospective
    Inherits Func 'Vererbung

    'Zugriff auf Ein- und Ausgangswerte vorheriger Funktionsblockaufrufe ermöglichen
    Protected Const k As Integer = 0 'Für Schreibweise z. B. u(k-2)
    Protected Const Length As Integer = 2 'Schieberegister-Größe
    Private _u(0 To Length) As Double 'Eingangswerte
    Private _y(0 To Length) As Double 'Ausgangswerte
    Delegate Function FuncDelegate(ByVal i As Integer) As Double 'Schreibweise z. B. u(k-2)
    Protected u As FuncDelegate = Function(i As Integer) _u(-i) 'Schieberegister für Eingangswerte
    Protected y As FuncDelegate = Function(i As Integer) _y(-i) 'Schieberegister für Ausgangswerte

    'Ausgangsfunktion
    Public Overrides Function Output(ByVal Input As Double) As Double
        _u(k) = Input 'Eingangswert in Schieberegister speichern
        _y(k) = Double.NaN 'Ausgangswert ist noch nicht definiert
        _y(k) = Functionality() 'Ausgangswert berechnen
        For i As Integer = Length To 1 Step -1 'Werte des Schieberegisters verschieben
            _u(k + i) = _u(k + i - 1) 'Schieberegister Eingang
            _y(k + i) = _y(k + i - 1) 'Schieberegister Ausgang
        Next
        Return _y(k) 'Ausgangswert zurückgeben
    End Function

    'Definition der Funktionalität des Funktionsblocks (z. B. mit einer Differenzengleichung)
    Protected MustOverride Function Functionality() As Double
```

```
    'Schieberegister zurücksetzen
    Public Overridable Sub Reset()
        Array.Clear(_u, 0, _u.Length) : Array.Clear(_y, 0, _y.Length)
    End Sub

    'Anfangsbedingungen setzen
    Public Sub SetInitialConditions(ByVal u As Double(), ByVal y As Double())
        Reset() 'Schieberegister zurücksetzen
        u.CopyTo(_u, 0) : y.CopyTo(_y, 0) 'Werte der Anfangsbedingungen übernehmen
    End Sub

End Class

'-----------------------------------------------------------------------------------------------'
'Funktionsblock, der sich auf vorherige Ein- und Ausgangswerte bezieht und zeitabhängig ist.
Public MustInherit Class FuncRetrospectiveTimeDependent
    Inherits FuncRetrospective 'Vererbung

    Protected Property T As Double 'DeltaT
    Public MustOverride Property ApproxType As [Enum] 'Näherungsverfahren

    'Ausgangsfunktion
    Public Overloads Function Output(ByVal Input As Double, ByVal DeltaT As Double) As Double
        T = DeltaT 'DeltaT speichern
        Return MyBase.Output(Input)
    End Function

End Class

'-----------------------------------------------------------------------------------------------'
'Funktionsblock, der eine mathematische Funktion in einem Definitionsbereich beschreibt.
Public MustInherit Class FuncSegment
    Inherits Func 'Vererbung

    Property Length As Double = 10 'Definitionslänge auf der X-Achse ([0; Length])
    Property Factor As Double = 1 'Streckung der Funktion in Y-Richtung
    Property Offset As Double = 0 'Verschiebung der Funktion in Y-Richtung

    'Ausgangsfunktion
    Public Overrides Function Output(Input As Double) As Double
        'Nur Definitionsbereich [0; Length] zulassen
        If Not (Input >= 0 And Input <= Length) Then Return Double.NaN
        'Streckung und Verschiebung der mathematischen Funktion
        Return RawFunc(Input) * Factor + Offset
    End Function

    'Mathematische Funktion
    Protected MustOverride Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])

End Class

'-----------------------------------------------------------------------------------------------'
'Funktionsblock, der eine periodische mathematische Funktion beschreibt.
Public MustInherit Class FuncSegmentPeriodic
    Inherits FuncSegment 'Vererbung

    Property Frequency As Double = 1 'Frequenz in Hz (]-∞; ∞[)
    Property Phase As Double = 0 'Phase in 1 = 360° (]-∞; ∞[)

    'Mathematische Funktion (wird von der Basisklasse in Y-Richtung skaliert und verschoben)
    Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0 to ∞])
        If Double.IsInfinity(Input) Then Return Double.NaN 'Unendlich -> NaN
        Input = Input * Frequency + Phase 'RawFunc in X-Richtung strecken und verschieben
        Input = Input Mod 1 'Eingangswert in Definitionsbereich ]-1 to 1[ bringen
        If Input < 0 Then Input += 1 'Eingangswert in den Definitionsbereich [0 to 1[ bringen (Mod2)
        Return RawFuncOnePeriod(Input)
    End Function

    'Eine Periode der mathematischen Funktion (in X- und Y-Richtung weder skaliert noch verschoben)
    Protected MustOverride Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)

End Class

End Namespace
```

```
'-----------------------------------------------------------------------------------------------------'
'Regelglieder
Namespace Controller

'-----------------------------------------------------------------------------------------------------'
'#### Proportionalglied ####
Public Class P
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.None
    Public Enum ApproxTypeEnum
        None 'Keine Approximation
    End Enum

    Public Property K_p As Double = 1 'Proportionalverstärkung

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.None 'Differenzengleichung
                Return K_p * u(0)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Integrationsglied ####
Public Class I
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_i As Double = 1 'Integrationszeitkonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T / T_i * u(k - 1) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return T / T_i * u(k) + y(k - 1)
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return T / (2 * T_i) * (u(k - 1) + u(k)) + y(k - 1)
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return T / T_i * u(k) + y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Differenzierer ####
Public Class D
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.EulerBackward
    Public Enum ApproxTypeEnum
        EulerBackward 'Approximation nach Euler-Rückwärts
    End Enum

    Public Property T_d As Double = 1 'Differentiationskonstante
```

```
    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return T_d / T * (u(k) - u(k - 1))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### PIDT1-Übertragungsglied ####
Public Class PIDT1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property K_r As Double = 0.4 'Regler-Verstärkung
    Public Property T_i As Double = 1 'Integrationszeitkonstante bzw. Nachstellzeit
    Public Property T_d As Double = 2 'Differentiationskonstante bzw. Vorhaltzeit
    Public Property T_a As Double = 0.5 'Zeitkonstante des Differenzierers

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / (T_i * T_a) *
                            (K_r * T_i * (T_d + T_a) * u(k) _
                            + K_r * (T_i * (T - 2 * (T_d + T_a)) + T_a * T) * u(k - 1) _
                            + K_r * (T_i * (T_d + T_a - T) - T * (T_a - T)) * u(k - 2) _
                            + T_i * (2 * T_a - T) * y(k - 1) _
                            - T_i * (T_a - T) * y(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_i * (T_a + T)) *
                            (K_r * T_i * (T_d + T_a) * u(k - 2) _
                            - K_r * (T_i * (2 * (T_d + T_a) + T) + T_a * T) * u(k - 1) _
                            + K_r * (T_i * (T_d + T_a + T) + T * (T_a + T)) * u(k) _
                            - T_i * T_a * y(k - 2) _
                            + T_i * (2 * T_a + T) * y(k - 1))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (2 * T_i * (2 * T_a + T)) *
                            (K_r * (2 * (T_i * (2 * (T_d + T_a) - T) - T_a * T) + T ^ 2) * u(k - 2) _
                            + 2 * K_r * (-4 * T_i * (T_d + T_a) + T ^ 2) * u(k - 1) _
                            + K_r * (2 * T_i * (2 * (T_d + T_a) + T) + T * (2 * T_a + T)) * u(k) _
                            - 2 * T_i * (2 * T_a - T) * y(k - 2) _
                            + 8 * T_i * T_a * y(k - 1))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Verzögerungsglied 1. Ordnung ####
Public Class T1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_a As Double = 1 'Zeitkonstante
```

```
    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T / T_a * (u(k - 1) - y(k - 1)) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_a + T) * (T_a * y(k - 1) + T * u(k))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (2 * T_a + T) * ((2 * T_a - T) * y(k - 1) + T * u(k - 1) + T * u(k))
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return (1 - Math.Exp(-1 / T_a * T)) * u(k - 1) + Math.Exp(-1 / T_a * T) * y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Verzögerungsglied 2. Ordnung ####
Public Class T2S
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum 'Unterstützte Approximationstypen
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property w_0 As Double = 2 'Kennkreisfrequenz
    Public Property d As Double = 0.5 'Dämpfungskonstante

    Protected Overrides Function Functionality() As Double 'Funktionalität des Funktionsblocks
        Dim a As Double = 1 / (w_0 ^ 2 * T ^ 2) 'Substitution
        Dim b As Double = (2 * d) / (w_0 * T) 'Substitution
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / a * ((b - a - 1) * y(k - 2) + (2 * a - b) * y(k - 1) + u(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (a + b + 1) * ((2 * a + b) * y(k - 1) - a * y(k - 2) + u(k))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (4 * a + 2 * b + 1) * ((2 * b - 4 * a - 1) * y(k - 2) _
                                                + (8 * a - 2) * y(k - 1) + u(k - 2) _
                                                + 2 * u(k - 1) + u(k))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Verzögernder Differenzierer ####
Public Class DT1
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_d As Double = 1 'Differentiationskonstante
    Public Property T_a As Double = 1 'Zeitkonstante

    Protected Overrides Function Functionality() As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return T_d / T_a * (u(k) - u(k - 1)) + (1 - T / T_a) * y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T + T_a) * (T_d * (u(k) - u(k - 1)) + T_a * y(k - 1))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (T + 2 * T_a) * (T_d * (2 * u(k) - 2 * u(k - 1)) - (T - 2 * T_a) * y(k - 1))
```

```
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                Return T_d / T_a * (u(k) - u(k - 1)) + Math.Exp(-1 / T_a * T) * y(k - 1)
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Fallschirm ####
Public Class Parachute
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
        MatchedPoleZero 'Approximation nach Matched-Z-Transformation
    End Enum

    Public Property T_a As Double = 1 'Zeitkonstante
    Public Property Offset As Double = 0 'Versatz des Bezugspunkts

    Protected Overrides Function Functionality() As Double
        Dim OutputTmp As Double = Double.NaN
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                OutputTmp = T / T_a * ((u(k - 1) + Offset) - y(k - 1)) + y(k - 1)
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                OutputTmp = 1 / (T_a + T) * (T_a * y(k - 1) + T * (u(k) + Offset))
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                OutputTmp = 1 / (2 * T_a + T) * ((2 * T_a - T) * y(k - 1) +
                                                 T * (u(k - 1) + Offset) + T * (u(k) + Offset))
            Case ApproxTypeEnum.MatchedPoleZero 'Differenzengleichung nach Matched-Z-Transformation
                OutputTmp = (1 - Math.Exp(-1 / T_a * T)) * (u(k - 1) + Offset) +
                            Math.Exp(-1 / T_a * T) * y(k - 1)
        End Select
        If u(k - 0) > OutputTmp Then OutputTmp = u(k - 0) 'Fallschirm anheben
        Return OutputTmp
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Bandpass ####
Public Class Bandpass
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property f_l As Double = 0.1 'Untere Grenzfrequenz
    Public Property f_h As Double = 1 'Obere Grenzfrequenz

    Protected Overrides Function Functionality() As Double
        Dim T_l As Double = 1 / (2 * Math.PI * f_h)
        Dim T_h As Double = 1 / (2 * Math.PI * f_l)
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.EulerForward 'Differenzengleichung approximiert nach Euler-Vorwärts
                Return 1 / (T_l * T_h) * (T * T_h * (u(k - 1) - u(k - 2)) _
                                + (2 * T_l * T_h - T * (T_l + T_h)) * y(k - 1) _
                                - (T_l * T_h - T * (T_l + T_h) + T ^ 2) * y(k - 2))
            Case ApproxTypeEnum.EulerBackward 'Differenzengleichung approximiert nach Euler-Rückwärts
                Return 1 / (T_l * T_h + T * (T_l + T_h) + T ^ 2) * (
                                T * T_h * (u(k) - u(k - 1)) _
                                + (2 * T_l * T_h + T * (T_l + T_h)) * y(k - 1) _
                                - (T_l * T_h) * y(k - 2))
```

```
            Case ApproxTypeEnum.Tustin 'Differenzengleichung approximiert nach Tustin-Methode
                Return 1 / (4 * T_l * T_h + T * (2 * (T_l + T_h) + T)) * (
                               2 * T * T_h * (u(k) - u(k - 2)) _
                               + (8 * T_l * T_h - 2 * T ^ 2) * y(k - 1) _
                               - (4 * T_l * T_h - T * (2 * (T_l + T_h) - T)) * y(k - 2))
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Bandpass höherer Ordnung ####
Public Class BandpassX
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Tustin
    Public Enum ApproxTypeEnum
        EulerForward 'Approximation nach Euler-Vorwärts
        EulerBackward 'Approximation nach Euler-Rückwärts
        Tustin 'Approximation nach Tustin-Methode
    End Enum

    Public Property f_l As Double = (New Bandpass).f_l 'Untere Grenzfrequenz
    Public Property f_h As Double = (New Bandpass).f_h 'Obere Grenzfrequenz
    Public Property Order As Double = 2 'Ordnung (1 => 2. Ordnung, 2 => 4. Ordnung, 3 => 6. Ordnung...)

    Private Bandpasss() As Bandpass 'Alle Bandpässe (pro 2 Ordnungen 1 Bandpass)

    Protected Overrides Function Functionality() As Double
        If Not Order > 0 Then Return Double.NaN 'Bandpass muss mindestens 2. Ordnung (Order=1) sein

        'Bandpässe neu instanziieren, falls sich die Ordnung geändert hat.
        If Bandpasss Is Nothing OrElse Not Bandpasss.Length = Order Then
            ReDim Bandpasss(CInt(Order) - 1)
            For i As Integer = 0 To Bandpasss.Length - 1 'Alle Bandpässe durchlaufen
                Bandpasss(i) = New Bandpass 'Neue Instanz eines Bandpasses erstellen
                Bandpasss(i).f_l = f_l : Bandpasss(i).f_h = f_h 'Parameter übernehmen
            Next
        End If
        'Parameter aktualisieren, falls sich einer geändert hat.
        Static Lastf_l As Double : Static Lastf_h As Double
        If Not (Lastf_l = f_l And Lastf_h = f_h) Then
            For Each Bandpass In Bandpasss 'Alle Bandpässe durchlaufen
                Bandpass.f_l = f_l : Bandpass.f_h = f_h 'Parameter aktualisieren
                Lastf_l = f_l : Lastf_h = f_h 'Neue Parameter speichern
            Next
        End If
        'Ausgangsfunktion der Bandpässe aufrufen
        Dim Value As Double = u(0) 'Eingangswert des ersten Bandpasses
        For Each Bandpass In Bandpasss 'Alle Bandpässe durchlaufen
            Bandpass.ApproxType = ApproxType 'Approximations-Typ aktualisieren
            Value = Bandpass.Output(Value, T) 'Ausgangsfunktion aufrufen (Alle Bandpässe in Serie)
        Next
        'Ausgangswert des letzten Bandpasses zurückgeben.
        Return Value
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        'Jeden Bandpass einzeln zurücksetzen
        For Each Bandpass In Bandpasss 'Alle Bandpässe durchlaufen
            If Bandpass Is Nothing Then Exit For
            Bandpass.Reset() 'Bandpass zurücksetzen
        Next
    End Sub

End Class
```

```
'-----------------------------------------------------------------------------------------------------'
'#### Totzeitglied ####
Public Class DeadTime
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.ShorterTimeDelay
    Public Enum ApproxTypeEnum
        ShorterTimeDelay 'Kürzere Verzögerungszeit
        LongerTimeDelay 'Längere Verzögerungszeit
        Interpolation 'Genaueste Verzögerungszeit
    End Enum

    Property T_t As Double = 1 'Verzögerungszeit

    Private PQueue As New System.Collections.Queue() 'Alle gespeicherten Punkte
    Private Structure DeltaTAndValue 'Punkt bestehend aus Zykluszeit und Wert
        Property DeltaT As Double : Property Value As Double
        Sub New(ByVal DeltaT As Double, ByVal Value As Double)
            Me.DeltaT = DeltaT : Me.Value = Value
        End Sub
    End Structure

    Protected Overrides Function Functionality() As Double
        'Verzögern eines nicht zeitäquidistanten abgetasteten Signals.
        ' P5         P4     P?              P3            P2         P1         P0 (Punkte)
        ' |          |      !               |            |          |          !
        ' |-DeltaTP4-|----------DeltaTP3----|--DeltaTP2---|-DeltaTP1-|-DeltaTP0-!  (DeltaT einzeln)
        '                     !              |<------TimeAfter-------------------!  (DeltaT danach)
        '            |<-----------------------------TimeBefore------------------!  (DeltaT davor)
        '                     !<--Delay=-(T_t-SmallestDeltaT/2)------------------!  (Verzögerungszeit)

        'Die Werte und dazugehörigen Zykluszeiten aller Funktionsblockaufrufe werden gespeichert.
        'Zum Verzögern des Signals wird im Verlauf nach einem T_t-SmallestDeltaT/2 alten Wert gesucht
        'und je nach Approximations-Typ der Wert davor oder danach oder eine Interpolation der beiden
        'Werte zurückgegeben. Dabei wird T_t um SmallestDeltaT/2 korrigiert, weil der ausgegebene Wert
        'voraussichtlich über eine Zykluszeit lang aktuell sein wird. Die Korrektur erfolgt um die im
        'Verlauf kleinste vorkommende Zykluszeit, da bei schwankenden Zykluszeiten sonst evtl. Werte
        'zurückgegeben werden, die älter sind als bereits zurückgegebene Werte. Nicht mehr benötigte
        'Werte werden aus dem Verlauf gelöscht.

        'Falls noch kein Punkt im Speicher ist, einen Punkt mit dem Wert NaN hinzufügen
        If PQueue.Count = 0 Then PQueue.Enqueue(New DeltaTAndValue(1 / 0, Double.NaN))
        'Aktuellen Punkt (DeltaT mit Wert) hinzufügen
        PQueue.Enqueue(New DeltaTAndValue(T, u(0))) 'T=Zykluszeit, u(0)=Eingangswert
        'Kleinstes DeltaT aller Punkte ermitteln
        Dim SmallestDeltaT As Double = Double.PositiveInfinity 'Kleinste Zykluszeit
        For Each DeltaTAndValue As DeltaTAndValue In PQueue 'Alle Punkte durchlaufen
            If DeltaTAndValue.DeltaT < SmallestDeltaT Then SmallestDeltaT = DeltaTAndValue.DeltaT
        Next
        'Queue für bessere Laufzeit in ein Array kopieren
        Dim PArray(PQueue.Count - 1) As DeltaTAndValue 'Array dass den Werteverlauf enthält
        Dim PIndex As Integer = PQueue.Count - 1 'Array-Index
        For Each DeltaTAndValue As DeltaTAndValue In PQueue 'Queue durchlaufen
            PArray(PIndex) = DeltaTAndValue 'Array rückwärts beschreiben (PArray(0) = Aktuellster)
            PIndex -= 1 'Array-Index dekrementieren
        Next
        'Ausgangswert berechnen
        Dim TimeBefore, TimeAfter As Double 'Zeitpunkt vor und nach dem gesuchten Punkt Px
        'Bis zur geforderten Verzögerungszeit zurückzählen, um einen verzögerten Wert auszugeben.
        For i As Integer = 1 To PArray.Count - 1 'Alle Index der Punkte durchlaufen
            'Zeitpunkt davor durch Aufsubtrahieren der P(x).DeltaT ermitteln
            TimeBefore -= PArray(i - 1).DeltaT 'Zeitpunkt negativ, weil er in der Vergangenheit liegt
            'Gesuchten Punkt ermitteln (T_t wird um DeltaTAverage/2 korrigiert (Beschreibung oben))
            Dim Delay As Double = -(T_t - SmallestDeltaT / 2)
            'Liegt der gesuchte Punkt in der Zukunft, suche den aktuellsten Punkt.
            If Delay > 0 Then Delay = 0
            'Liegt der gesuchte Punkt zwischen den beiden aktuellen Punkten (DateBefore und DateAfter)?
            If TimeBefore <= Delay And TimeAfter >= Delay Then '>= und <= weil Zykluszeit 0 sein kann
                'Überflüssige Punkte aus der Queue löschen
                'i+1, weil evtl. darauf zurückgegriffen wird
                For k As Integer = i + 1 To PArray.Count - 1
                    PQueue.Dequeue() 'Überflüssigen Punkt der Vergangenheit löschen
                Next
```

```
                Select Case DirectCast(ApproxType, ApproxTypeEnum)
                    Case ApproxTypeEnum.ShorterTimeDelay 'Kürzere Verzögerungszeit
                        Return PArray(i - 1).Value
                    Case ApproxTypeEnum.LongerTimeDelay 'Längere Verzögerungszeit
                        Return PArray(i).Value
                    Case ApproxTypeEnum.Interpolation 'Genaueste Verzögerungszeit
                        'Verhältnis für Interpolation
                        Dim Ratio As Double = (Delay - TimeBefore) / (TimeAfter - TimeBefore)
                        Return Ratio * PArray(i - 1).Value + (1 - Ratio) * PArray(i).Value
                End Select
            End If
            TimeAfter = TimeBefore 'Der Punkt davor wird im nächsten Durchlauf der Punkt danach sein.
        Next
        Return Double.NaN 'Der Punkt zur gesuchten Verzögerungszeit liegt außerhalb des Verlaufs.
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Steigungsbegrenzer ####
Public Class SlopeLimit
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.None
    Public Enum ApproxTypeEnum
        None 'Keine Approximation
    End Enum

    Public Property Slope As Double = 1 'Maximale Steigung

    Protected Overrides Function Functionality() As Double
        Dim Output As Double = y(k - 1) 'Ausgang speichern
        Dim Sign As Double = Double.NaN 'Richtung, in die der Ausgangswert laufen soll
        If y(k - 1) < u(k) Then Sign = 1 'Positive Richtung
        If y(k - 1) > u(k) Then Sign = -1 'Negative Richtung
        If y(k - 1) = u(k) Then Sign = 0 'Stagnieren
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.None
                Output += Sign * T * Slope 'Ausgangswert verändern
                'Springe auf den Soll-Wert, anstatt ihn zu überspringen
                If Sign * Output > Sign * u(k) Then Output = u(k)
                Return Output
            Case Else
                Return Double.NaN
        End Select
    End Function

End Class

End Namespace

'-----------------------------------------------------------------------------------------------------'
'Signalgeneratoren
Namespace Generator

'-----------------------------------------------------------------------------------------------------'
'#### Sägezahn-Generator ####
Public Class Sawtooth
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously
    Public Enum ApproxTypeEnum
        ReturnToZero 'X-Position wird nach jeder Periode zurückgesetzt
        Continuously 'X-Position wird nach jeder Periode um eine Periodenlänge zurückgesetzt
    End Enum

    Property Offset As Double = 0 'Offset (Verschiebung in Y-Richtung)
    Property Factor As Double = 1 'Verstärkung (Skalierung in Y-Richtung)
    Property Phase As Double = 0 'Phase (Verschiebung in X-Richtung)
    Property Frequency As Double = 1 'Frequenz (Skalierung in X-Richtung)

    Private PosInFunc As Double 'X-Position in "RawFunc"-Funktion [0; 1[
```

```
    Protected Delegate Function RawFunc_Delagate(ByVal Input As Double) As Double
    Protected RawFunc As RawFunc_Delagate = Function(ByVal Input As Double) As Double 'Periode [0; 1[
                                                Return Input 'Eine Periode der Sägezahn-Funktion
                                            End Function

    Protected Overrides Function Functionality() As Double
        If Double.IsNaN(u(k - 0)) Then Return y(k - 1) 'Bei NaN am Eingang Generator einfrieren
        If Not (u(k - 0) > 0 Or u(k - 0) < 0) Then 'Bei False am Eingang
            PosInFunc = 0 'X-Position zurücksetzen
            Return Offset 'Offset zurückgeben
        End If

        'Bei Frequenzänderung bleibt die Funktion stetig, bei Phasenänderung nicht.
        PosInFunc += T * Frequency 'X-Position abhängig von der Frequenz erhöhen
        Dim PosInFuncTmp As Double
        Select Case DirectCast(ApproxType, ApproxTypeEnum)
            Case ApproxTypeEnum.ReturnToZero 'Approximations-Typ (siehe oben)
                If PosInFunc >= 1 Then PosInFunc = 0
                PosInFuncTmp = PosInFunc
            Case ApproxTypeEnum.Continuously 'Approximations-Typ (siehe oben)
                PosInFunc = PosInFunc Mod 1
                PosInFuncTmp = PosInFunc
                PosInFuncTmp -= T * Frequency / 2 'Korrektur um halbe Zykluszeit bei "Continuously"
            Case Else
                Return Double.NaN
        End Select
        PosInFuncTmp += Phase 'Phasenverschiebung
        PosInFuncTmp = PosInFuncTmp Mod 1 'Wert in den Bereich ]-1; 1[ bringen
        If PosInFuncTmp < 0 Then PosInFuncTmp += 1 'Wert in den Definitionsbereich [0; 1[ bringen
        PosInFuncTmp = Offset + Factor * RawFunc(PosInFuncTmp) 'Offset und Skalierung
        Return PosInFuncTmp
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        PosInFunc = 0 'X-Position zurücksetzen
    End Sub

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Dreieck-Generator ####
Public Class Triangle
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Private FuncSegmentTriangle As New FuncSegment.Triangle 'Zugrundeliegendes Funktionssegment

    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentTriangle.Offset = 1 / 2
        FuncSegmentTriangle.Factor = 1 / 2
        FuncSegmentTriangle.Frequency = 1
        FuncSegmentTriangle.Phase = -1 / 4
        FuncSegmentTriangle.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentTriangle.Output(Input)
                  End Function
    End Sub

End Class

'-----------------------------------------------------------------------------------------------------'
'#### PWM-Generator ####
Public Class PWM
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Property DutyCycle As Double = 0.2 'Tastgrad (Verhältnis Impulsdauer zu Periodendauer)

    Private FuncSegmentPWM As New FuncSegment.PWM 'Zugrundeliegendes Funktionssegment
```

```
    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentPWM.Offset = 0
        FuncSegmentPWM.Factor = 1
        FuncSegmentPWM.Frequency = 1
        FuncSegmentPWM.Phase = 0
        FuncSegmentPWM.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentPWM.Output(Input)
                  End Function
    End Sub

    Protected Overrides Function Functionality() As Double
        FuncSegmentPWM.DutyCycle = DutyCycle 'Tastgrad bei Aufruf der Ausgangsfunktion aktualisieren
        Return MyBase.Functionality
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Sinus-Generator ####
Public Class Sin
    Inherits Sawtooth

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Continuously

    Private FuncSegmentSin As New FuncSegment.Sin 'Zugrundeliegendes Funktionssegment

    Sub New()
        'Eine Periode mit der Funktion "RawFunc" beschreiben
        'Skalierung und Verschiebung in X- und Y-Richtung führt die Basisklasse durch
        FuncSegmentSin.Offset = 0
        FuncSegmentSin.Factor = 1
        FuncSegmentSin.Frequency = 1
        FuncSegmentSin.Phase = 0
        FuncSegmentSin.Length = 1
        RawFunc = Function(ByVal Input As Double) As Double '(Input [0; 1[)
                      Return FuncSegmentSin.Output(Input)
                  End Function
    End Sub

End Class

End Namespace

'Zeitglieder
Namespace Timer

'-----------------------------------------------------------------------------------------------------'
'#### Anzugsverzögerung ####
Public Class TurnOnDelay
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Verzögerungszeit

    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch >= Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T >= Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 >= Delay, 1, 0)
```

```
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else 'Bei Input = False
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 0
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class

'-------------------------------------------------------------------------------------------------------'
'#### Abfallverzögerung ####
Public Class TurnOffDelay
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Verzögerungszeit

    Private StopWatch As Double = Double.PositiveInfinity 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If Not (u(k - 0) > 0 Or u(k - 0) < 0) Then 'Bei Input = False
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch < Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T < Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 < Delay, 1, 0)
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else 'Bei Input = True
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 1
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = Double.PositiveInfinity 'Zeitzähler zurücksetzen
    End Sub

End Class

'-------------------------------------------------------------------------------------------------------'
'#### Impuls ####
Public Class Puls
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.TooLate
    Public Enum ApproxTypeEnum
        TooLate 'Eher zu spät reagieren
        TooEarly 'Eher zu früh reagieren
        Punctually 'Möglichst genau reagieren
    End Enum

    Property Delay As Double 'Impulslänge
```

```
    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Or y(k - 1) = 1 Then
            'Bei Input = True oder wenn der Impuls noch nicht vollständig ausgegeben wurde
            Select Case DirectCast(ApproxType, ApproxTypeEnum)
                Case ApproxTypeEnum.TooLate 'Eher zu spät reagieren
                    Functionality = If(StopWatch < Delay, 1, 0)
                Case ApproxTypeEnum.TooEarly 'Eher zu früh reagieren
                    Functionality = If(StopWatch + T < Delay, 1, 0)
                Case ApproxTypeEnum.Punctually 'Möglichst genau reagieren
                    Functionality = If(StopWatch + T / 2 < Delay, 1, 0)
                Case Else
                    Functionality = Double.NaN
            End Select
            StopWatch += T 'Zeitzähler um Zykluszeit erhöhen
        Else
            'Bei Input = False, aber nur wenn der Impuls schon vollständig ausgegeben wurde
            StopWatch = 0 'Zeitzähler zurücksetzen
            Functionality = 0
        End If
        Return Functionality
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Stoppuhr ####
Public Class StopWatch
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.ResetToZero
    Public Enum ApproxTypeEnum
        ResetToZero 'Auf 0 zurücksetzen
        ResetToDeltaT 'Auf Zykluszeit vorladen
        ResetToHalfDeltaT 'Auf halbe Zykluszeit vorladen
    End Enum

    Private StopWatch As Double 'Zeitzähler

    Protected Overrides Function Functionality() As Double
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            StopWatch += T 'Zeit hochzählen
            If u(k - 1) = 0 Then 'Bei Flanke auf True
                StopWatch = 0 'Zeitzähler zurücksetzen
                Select Case DirectCast(ApproxType, ApproxTypeEnum)
                    Case ApproxTypeEnum.ResetToZero
                        StopWatch += 0 'Zeitzähler zurücksetzen
                    Case ApproxTypeEnum.ResetToDeltaT
                        StopWatch += T 'Zeitzähler auf Zykluszeit vorladen
                    Case ApproxTypeEnum.ResetToHalfDeltaT
                        StopWatch += T / 2 'Zeitzähler auf halbe Zykluszeit vorladen
                    Case Else
                        StopWatch += Double.NaN
                End Select
            End If
        End If
        Return StopWatch
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch = 0 'Zeitzähler zurücksetzen
    End Sub

End Class

End Namespace
```

```
'-----------------------------------------------------------------------------------------------------'
'Flankenerkennungen
Namespace Flank

'-----------------------------------------------------------------------------------------------------'
'#### Positive Flankenerkennung ####
Public Class Hi
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1), 1, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Negative Flankenerkennung ####
Public Class Lo
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) < u(k - 1), 1, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Flankenerkennung ####
Public Class Any
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1) Or u(k - 0) < u(k - 1), 1, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Toggle-Flipflop ####
Public Class Relay
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) > u(k - 1), If(y(k - 1) = 0, 1, 0), y(k - 1))
    End Function

End Class

End Namespace

'Analoge Übertragungsglieder
Namespace Analog

'-----------------------------------------------------------------------------------------------------'
'#### Wertänderung ####
Public Class Change
    Inherits BaseClass.FuncRetrospective

    Protected Overrides Function Functionality() As Double
        Return If(Not u(k - 0).Equals(u(k - 1)), 1, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Hysterese ####
Public Class Hysteresis
    Inherits BaseClass.FuncRetrospective

    Property Hi As Double = 2 / 3 'Schwellwert zum Einschalten
    Property Lo As Double = 1 / 3 'Schwellwert zum Ausschalten
```

```
    Protected Overrides Function Functionality() As Double
        Return If(u(k - 0) <= Lo, 0, If(u(k - 0) >= Hi, 1, y(k - 1)))
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Zufallsgenerator ####
Public Class Random
    Inherits BaseClass.FuncRetrospective

    Property RndCount As Double = 1 'Anzahl der Würfel (Beeinflusst die Wahrscheinlichkeitsverteilung)
    Shared Rnd As New System.Random(Convert.ToInt32(Now.Ticks Mod Integer.MaxValue)) 'Zufallsgenerator

    Protected Overrides Function Functionality() As Double
        If Not (RndCount >= 1 And RndCount <= Int32.MaxValue) Then
            Throw New Exception("RndCount is outside the allowed value range!")
        End If
        If u(k - 0) > 0 Or u(k - 0) < 0 Then 'Bei Input = True
            Dim Value As Double
            'Alle Würfel werfen
            For i As Integer = 0 To Convert.ToInt32(RndCount) - 1
                Value += Rnd.NextDouble Mod 1 'Würfeln
            Next
            Value /= RndCount 'Mittelwert aller Würfel
            Return Value
        Else 'Bei Input = False
            Return y(k - 1) 'Letzten Zufallswert zurückgeben
        End If
    End Function

    Public Overrides Sub Reset()
        'Semaphore setzen weil, die Funktion "SetInitialConditions" selbst "Reset" aufruft.
        Static Semaphor As Integer = 0
        If Threading.Interlocked.Exchange(Semaphor, 1) = 1 Then Exit Sub
        Me.SetInitialConditions({}, {0, 0.5}) 'Letzten Zufallswert auf 0,5 setzen
        Semaphor = 0
    End Sub

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Zykluszeit ####
Public Class DeltaT
    Inherits BaseClass.FuncRetrospective

    Private StopWatch As New Stopwatch 'Hochauflösender Zeitgeber
    Private LastElapsedTicks As Long 'Verstrichene Zeit seit dem ersten Funktionsaufruf in Ticks

    Protected Overrides Function Functionality() As Double
        'Verstrichene Zeit seit erstem Funktionsaufruf ermitteln in Ticks
        Dim ElapsedTicks As Long = StopWatch.ElapsedTicks
        'Verstrichene Zeit seit letztem Funktionsaufruf in Sekunden ermitteln
        Dim DeltaT As Double = (ElapsedTicks - LastElapsedTicks) / Stopwatch.Frequency
        'Verstrichene Zeit seit erstem Funktionsaufruf merken
        LastElapsedTicks = ElapsedTicks
        'Zeitgeber beim ersten Durchlauf starten (danach nicht mehr zurücksetzen)
        If Not StopWatch.IsRunning Then StopWatch.Start()
        'Verstrichene Zeit seit dem letzten Funktionsaufruf zurückgeben
        Return DeltaT
    End Function

    Public Overrides Sub Reset()
        MyBase.Reset()
        StopWatch.Reset() 'Zeitgeber zurücksetzen
        LastElapsedTicks = 0 'Seit erstem Funktionsaufruf verstrichene Zeit zurücksetzen
    End Sub

End Class

End Namespace
```

```
'-----------------------------------------------------------------------------------------------------'
'Grundlegende Funktionen
Namespace Generic

'-----------------------------------------------------------------------------------------------------'
'#### Betrag ####
Public Class Abs
    Inherits BaseClass.Func

    Public Overrides Function Output(ByVal Input As Double) As Double
        Return Math.Abs(Input)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Vorzeichen ####
Public Class Sign
    Inherits BaseClass.Func

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return Double.NaN
        Return Math.Sign(Input)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Modulo1 ####
Public Class Mod1
    Inherits BaseClass.Func

    Property Divisor As Double = 1 'Divisor

    Public Overrides Function Output(ByVal Input As Double) As Double
        Return Input Mod Divisor
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Modulo2 ####
Public Class Mod2
    Inherits BaseClass.Func

    Property Divisor As Double = 1 'Divisor

    Public Overrides Function Output(ByVal Input As Double) As Double
        Input = Input Mod Divisor 'Eingangswert in den Bereich ]-1 to 1[ bringen
        Input += If(Input < 0, Math.Abs(Divisor), 0) 'Eingangswert in den Def.bereich [0 to 1[ bringen
        Return Input
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Begrenzer ####
Public Class Limit
    Inherits BaseClass.Func

    Property Max As Double = +1 'Wertebereich Obergrenze
    Property Min As Double = -1 'Wertebereich Untergrenze

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Input > Max Then Return Max 'Auf Obergrenze begrenzen
        If Input < Min Then Return Min 'Auf Untergrenze begrenzen
        Return Input
    End Function

End Class
```

```
'-----------------------------------------------------------------------------------------------------'
'#### Gültigkeitsbereich ####
Public Class ValidRange
    Inherits BaseClass.Func

    Property Min As Double = -1
    Property Max As Double = +1

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return Double.NaN
        Return If(Input >= Min And Input <= Max, 1, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### AntiNaN ####
Public Class AntiNaN
    Inherits BaseClass.Func

    Property InfinityValue As Double = Single.MaxValue 'Ersatzwert für den Wert "unendlich"

    Public Overrides Function Output(ByVal Input As Double) As Double
        If Double.IsNaN(Input) Then Return 0
        If Input > InfinityValue Then Return InfinityValue
        If Input < -InfinityValue Then Return -InfinityValue
        Return Input
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Abschnittsweise definierte Funktion ####
Public Class FuncOfSegments
    Inherits BaseClass.Func

    'Die Funktions-Segmente werden wie im folgenden Beispiel aneinandergereiht:
    'Segment a(x) mit der Länge 1 beschreibt den Bereich [0;1[
    'Segment b(x) mit der Länge 2 beschreibt den Bereich [1;3[
    'Segment c(x) mit der Länge 2 beschreibt den Bereich [3;5]
    'Die Aufrufe der Ausgangsfunktion geben dann folgende Werte zurück:
    'f(-0.1) => NaN
    'f(0.0) => a(0.0)
    'f(1.0) => b(0.0)
    'f(3.0) => c(0.0)
    'f(4.0) => c(1.0)
    'f(5.0) => c(2.0)
    'f(6.0) => NaN

    'Liste der Funktions-Segmente
    Property Items As New List(Of BaseClass.FuncSegment)

    Public Overrides Function Output(ByVal Input As Double) As Double
        'Numerische Fehler werden auf Kosten der Gleitkommazahl-Genauigkeit unterdrückt
        Dim Length As Double = GetLength() 'Länge zwischenspeichern für bessere Laufzeit
        If Items.Count = 0 Then Return Double.NaN 'Keine Funktions-Segmente
        If Input < 0 Then Return Double.NaN 'Außerhalb der unteren Definitionsgrenze
        If Convert.ToDouble(Convert.ToSingle(Input)) = Convert.ToDouble(Convert.ToSingle(Length)) _
        AndAlso Items.Count > 0 Then 'Auf der oberen Definitionsgrenze (mit Toleranz Single.Epsilon)
            Return Items(Items.Count - 1).Output(Items(Items.Count - 1).Length())
        End If
        If Input > Length Then Return Double.NaN 'Außerhalb der oberen Definitionsgrenze
        'Funktions-Segmente bis zur gesuchten X-Position durchlaufen und den Funktions-Wert zurückgeben
        Dim PosX As Double = -0.0001 'X-Position
        For Each FunSegment As BaseClass.FuncSegment In Items 'Alle Funktionssegmente durchlaufen
            'Numerische Fehler unterdrücken, indem auf Single-Genauigkeit gerundet wird
            Dim PosInSegRounded As Double = Convert.ToDouble(Convert.ToSingle(Input - PosX))
            Dim FunSegLengthRounded As Double = Convert.ToDouble(Convert.ToSingle(FunSegment.Length))
            'Ist die Position im Definitionsbereich des Funktionssegments?
            If PosInSegRounded < FunSegLengthRounded Then
                'Durch die Konvertierung in Single kann PosInSegment minimal kleiner als 0 oder minimal
                'größer als FunSegment.Length() sein. Deshalb Wert wieder in den Def.-Bereich bringen.
                If PosInSegRounded < 0 Then PosInSegRounded = 0
                If PosInSegRounded > FunSegment.Length() Then PosInSegRounded = FunSegment.Length()
                Return FunSegment.Output(PosInSegRounded) 'Gebe den Funktionswert zur X-Position zurück
            End If
```

```
            PosX += FunSegment.Length() 'Verschiebe die aktuelle X-Position um die Segment-Länge weiter
        Next
        Return Double.NaN
    End Function

    Public Function GetLength() As Double 'Gesamtlänge aller Funktionssegmente
        Dim LengthDouble As Double 'Summe aller Funktionssegment-Längen
        For Each Segment As BaseClass.FuncSegment In Items 'Alle Funktionssegmente durchlaufen
            LengthDouble += Segment.Length() 'Längen aufaddieren
        Next
        Return LengthDouble 'Gesamtlänge zurückgeben
    End Function

End Class

End Namespace

'Funktions-Segmente
Namespace FuncSegment

'-----------------------------------------------------------------------------------------------------'
'#### Konstante Funktion ####
Public Class Constant
    Inherits BaseClass.FuncSegment

    Sub New()
        Factor = 0 'Standardwert überschreiben
    End Sub

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return 0
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Lineare Funktion ####
Public Class Linear
    Inherits BaseClass.FuncSegment

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return Input
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Stufenfunktion ####
Public Class Steps
    Inherits BaseClass.FuncSegment

    Property StepLength As Double = 1 'Stufenlänge

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        Return Math.Truncate(Input / StepLength)
    End Function

End Class

'-----------------------------------------------------------------------------------------------------'
'#### Zufallsfunktion ####
Public Class Random
    Inherits BaseClass.FuncSegment

    Property Seed As Double = 0 'Startwert (ordnet den Eingangswerten andere Ausgangswerte zu)
    Property RndCount As Double = 1 'Anzahl der Würfel (beeinflusst die Wahrscheinlichkeitsverteilung)

    Protected Overrides Function RawFunc(ByVal Input As Double) As Double '(Input [0; ∞])
        If Not (RndCount >= 1 And RndCount <= Int32.MaxValue) Then
            Throw New Exception("RndCount is outside the allowed value range!")
        End If
        'Startwert des Zufallsgenerators aus Input und Seed generieren.
        Dim V As Byte() = BitConverter.GetBytes(Input) 'Bits des Double-Werts als Byte-Array darstellen
        Dim S As Byte() = BitConverter.GetBytes(Seed) 'Bits des Double-Werts als Byte-Array darstellen
        Dim IntegerSeed As Byte() = { 'Bits beider Double-Werte zusammenmischen
```

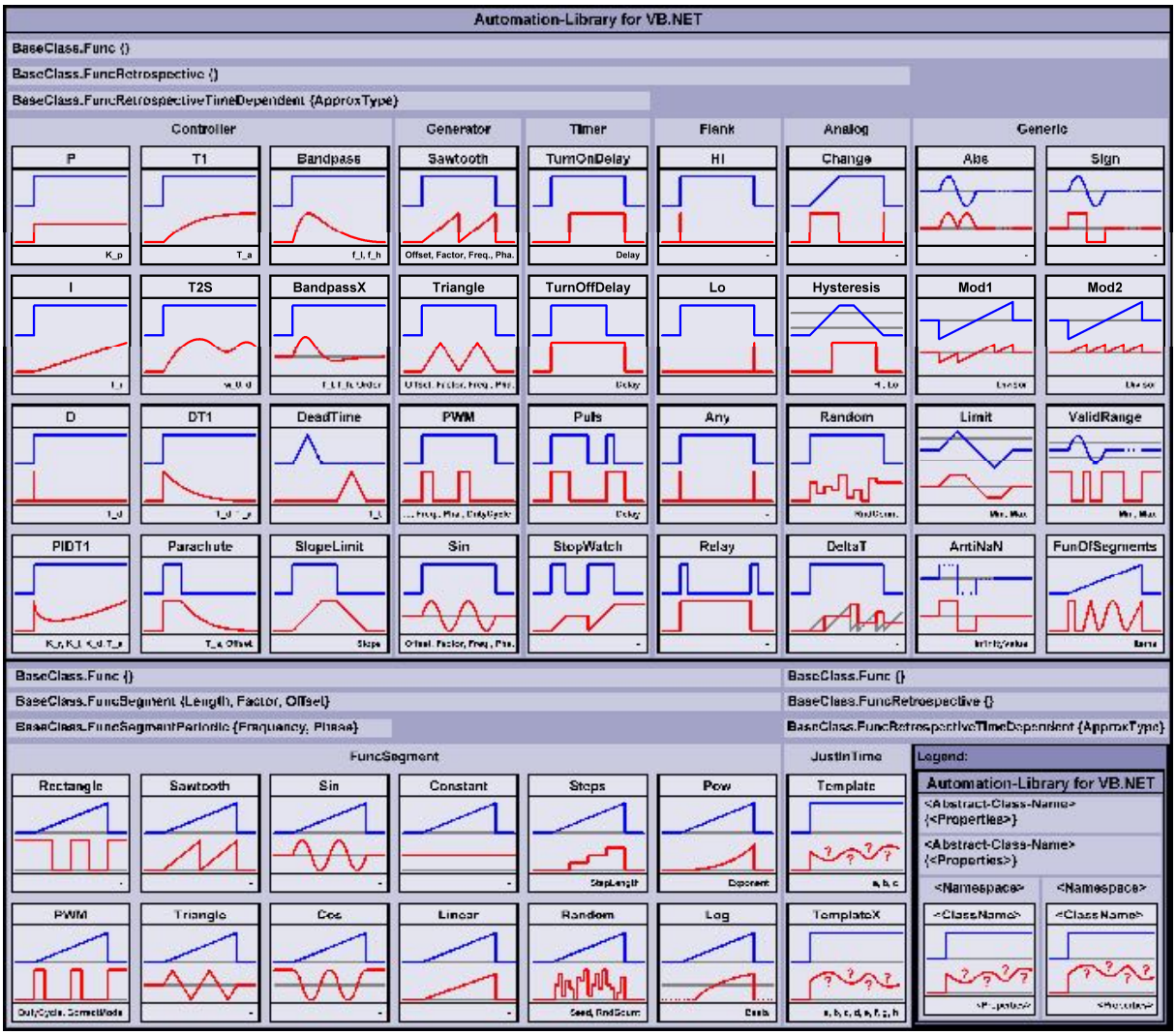
```
            CByte((CInt(V(1)) + V(4) + S(1) + S(4)) Mod 255),
            CByte((CInt(V(2)) + V(5) + S(2) + S(5)) Mod 255),
            CByte((CInt(V(3)) + V(6) + S(3) + S(6)) Mod 255),
            CByte((CInt(V(4)) + V(7) + S(4) + S(7)) Mod 255)}
            'Byte-Array in einen Integer-Wert zurück konvertieren
            Dim GeneratorSeed As Integer = BitConverter.ToInt32(IntegerSeed, 0) 'Startwert
            'Zufallsgenerator mit Startwert (abhängig von Input und Seed) erstellen
            Dim RandomGenerator As New System.Random(GeneratorSeed)
            'Den Durchschnitt mehrerer Zufallswerte berechnen, um den Verteilungsgrad zu erreichen
            Dim ValueSum As Double 'Zufallswert
            For i As UInt32 = 0 To Convert.ToUInt32(RndCount - 1) 'Alle Würfel durchlaufen
                'Zufallswert für einen Würfel errechnen [-∞; ∞]
                Dim InputByteArray As Byte() = BitConverter.GetBytes(Input)
                '8 zufällige Bytes errechnen
                For j As Integer = 0 To 7
                    InputByteArray(j) = CByte(RandomGenerator.Next() Mod 255)
                Next
                '8 Bytes zu einem Integer-Wert zusammenführen
                Dim IntegerValue As UInt64 = BitConverter.ToUInt64(InputByteArray, 0)
                'Integer-Wert zu einem Double-Wert [0; 1]
                Dim Value As Double = IntegerValue / UInt64.MaxValue
                'Würfelergebnis aufaddieren
                ValueSum += Value
            Next
            Return ValueSum / RndCount 'Mittelwert aller Würfel bestimmen
        End Function

    End Class

    '-------------------------------------------------------------------------------------------------------'
    '#### Potenzfunktion ####
    Public Class Pow
        Inherits BaseClass.FuncSegment

        Property Exponent As Double = 2 'Exponent der Potenz-Funktion

        Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0; ∞])
            Return Math.Pow(Input, Exponent)
        End Function

    End Class

    '-------------------------------------------------------------------------------------------------------'
    '#### Logarithmische Funktion ####
    Public Class Log
        Inherits BaseClass.FuncSegment

        Property Basis As Double = 10 'Basis der logarithmischen Funktion

        Protected Overrides Function RawFunc(Input As Double) As Double '(Input [0; ∞])
            Return Math.Log(Input) / Math.Log(Basis)
        End Function

    End Class

    '-------------------------------------------------------------------------------------------------------'
    '#### Rechteck-Funktion ####
    Public Class Rectangle

        Inherits BaseClass.FuncSegmentPeriodic

        Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
            Return If(Input < 0.5, 1, 0)
        End Function

    End Class

    '-------------------------------------------------------------------------------------------------------'
    '#### PWM-Funktion ####
    Public Class PWM
        Inherits BaseClass.FuncSegmentPeriodic 'Vererbung

        Property DutyCycle As Double = 0.2 'Tastgrad (Verhältnis Impulsdauer zu Periodendauer)
        Property CorrectMode As Boolean = False 'frequenz- und phasenrichtig
```

```
    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        If CorrectMode Then 'Frequenz- und phasenrichtige Ausgabe?
            Return If((Convert.ToSingle(Input) < Convert.ToSingle(DutyCycle / 2)) Or
                      (Convert.ToSingle(Input) >= Convert.ToSingle(1 - DutyCycle / 2)), 1, 0)
        Else
            Return If(Input < DutyCycle, 1, 0)
        End If
    End Function

End Class

'-----------------------------------------------------------------------------------------------'
'#### Sägezahn-Funktion ####
Public Class Sawtooth

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return Input
    End Function

End Class

'-----------------------------------------------------------------------------------------------'
'#### Dreieck-Funktion ####
Public Class Triangle

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return 4 * If(Input >= 1 / 4 And Input < 3 / 4, -Input + 2 / 4, Input) _
                 + If(Input >= 3 / 4, -4, 0)
    End Function

End Class

'-----------------------------------------------------------------------------------------------'
'#### Sinus-Funktion ####
Public Class Sin

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return System.Math.Sin(Input * 2 * System.Math.PI)
    End Function

End Class

'-----------------------------------------------------------------------------------------------'
'#### Cosinus-Funktion ####
Public Class Cos

    Inherits BaseClass.FuncSegmentPeriodic

    Protected Overrides Function RawFuncOnePeriod(ByVal Input As Double) As Double '(Input [0; 1[)
        Return System.Math.Cos(Input * 2 * System.Math.PI)
    End Function

End Class

End Namespace

'-----------------------------------------------------------------------------------------------'
'Kompilierung zur Laufzeit
Namespace JustInTime
```

```
'-----------------------------------------------------------------------------------------------------------'
'#### Funktionsblock-Vorlage ####
Public Class Template
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx
    Public Enum ApproxTypeEnum
        Approx
    End Enum

    Public Property a As Double = 1
    Public Property b As Double = 1
    Public Property c As Double = 1

    Protected Overrides Function Functionality() As Double
        Return Double.NaN
    End Function

    Public Shared Function GetDefaultVbNetCode() As String
        Return <![CDATA[
'Funktionsblock-Vorlage, die zur Laufzeit der Testumgebung kompiliert wird.
Class Template
'Durch Vererbung stehen folgende Parameter und Eigenschaften zur Verfügung:
'Eingangswerte:         u(k-0), u(k-1), u(k-2)
'Ausgangswerte:         y(k-0), y(k-1), y(k-2)
'DeltaT:                T
'Approximations-Typ:    ApproxType
Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

'Approximationstyp dieses Templates
Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx
Public Enum ApproxTypeEnum
Approx
End Enum

'Parameter dieses Templates
Public Property a As Double
Public Property b As Double
Public Property c As Double

'Konstruktor
Sub New()
End Sub

'Funktion, zum Berechnen der Ausgangswerte.
Protected Overrides Function Functionality() As Double
'Je nach gewähltem Approximations-Typ werden die dazugehörigen Gleichungen aufgerufen.
Select Case DirectCast(ApproxType, ApproxTypeEnum)
    Case ApproxTypeEnum.Approx 'T1-Glied approximiert nach "Euler Vorwärts"
        Return T / a * (u(k - 1) - y(k - 1)) + y(k - 1)
    Case Else
        Return Double.NaN
End Select
End Function

End Class
]]>.Value()
    End Function

End Class

'-----------------------------------------------------------------------------------------------------------'
'#### Erweiterte Funktionsblock-Vorlage ####
Public Class TemplateX
    Inherits BaseClass.FuncRetrospectiveTimeDependent

    Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx0
    Public Enum ApproxTypeEnum
        Approx0
        Approx1
        Approx2
        Approx3
        Approx4
    End Enum
```

```
    Public Property a As Double = 1
    Public Property b As Double = 1
    Public Property c As Double = 1
    Public Property d As Double = 1
    Public Property e As Double = 1
    Public Property f As Double = 1
    Public Property g As Double = 1
    Public Property h As Double = 1

    Protected Overrides Function Functionality() As Double
        Return Double.NaN
    End Function

    Public Shared Function GetDefaultVbNetCode() As String
        Return <![CDATA[
'Funktionsblock-Vorlage, die zur Laufzeit der Testumgebung kompiliert wird.
Class TemplateX
'Durch Vererbung stehen folgende Parameter und Eigenschaften zur Verfügung:
'Eingangswerte:         u(k-0), u(k-1), u(k-2)
'Ausgangswerte:         y(k-0), y(k-1), y(k-2)
'DeltaT:                T
'Approximations-Typ:    ApproxType
Inherits BaseClass.FuncRetrospectiveTimeDependent 'Vererbung

'Approximationstyp dieses Templates
Public Overrides Property ApproxType As [Enum] = ApproxTypeEnum.Approx0
Public Enum ApproxTypeEnum
    Approx0
    Approx1
    Approx2
    Approx3
    Approx4
End Enum

'Parameter dieses Templates
Public Property a As Double = 1
Public Property b As Double = 1
Public Property c As Double = 1
Public Property d As Double = 1
Public Property e As Double = 1
Public Property f As Double = 1
Public Property g As Double = 1
Public Property h As Double = 1

'Funktion, zum Berechnen der Ausgangswerte.
Protected Overrides Function Functionality() As Double
    'Je nach gewähltem Approximations-Typ werden die dazugehörigen Gleichungen aufgerufen.
    Select Case DirectCast(ApproxType, ApproxTypeEnum)
        Case ApproxTypeEnum.Approx0 'T1-Glied approximiert nach "Euler Vorwärts"
            Return T / a * (u(k - 1) - y(k - 1)) + y(k - 1)
        Case ApproxTypeEnum.Approx1 'T1-Glied approximiert nach "Euler Rückwärts"
            Return 1 / (a + T) * (a * y(k - 1) + T * u(k))
        Case ApproxTypeEnum.Approx2 'T1-Glied approximiert nach "Tustin"
            Return 1 / (2 * a + T) * ((2 * a - T) * y(k - 1) + T * u(k - 1) + T * u(k))
        Case ApproxTypeEnum.Approx3 'T1-Glied approximiert nach Pol-Nullstellen-Abbildung
            Return (1 - Math.Exp(-1 / a * T)) * u(k - 1) + Math.Exp(-1 / a * T) * y(k - 1)
        Case ApproxTypeEnum.Approx4
            Return Double.NaN
        Case Else
            Return Double.NaN
    End Select
End Function

End Class
]]>.Value()
    End Function

End Class

End Namespace

End Namespace
```

Namensbereich-Verzeichnis



Basisklasse „Func“ → Seite 31

Basisklasse „FuncRetrospective“ → Seite 32

Basisklasse „FuncRetrospectiveTimeDep.“ → Seite 33

Basisklasse „FuncSegment“ → Seite 34

Basisklasse „FuncSegmentPeriodic“ → Seite 35

Namensbereich „Controller“ → Seite 45

Namensbereich „Generator“ → Seite 75

Namensbereich „Timer“ → Seite 85

Namensbereich „Flank“ → Seite 95

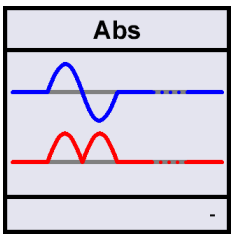
Namensbereich „Analog“ → Seite 105

Namensbereich „Generic“ → Seite 115

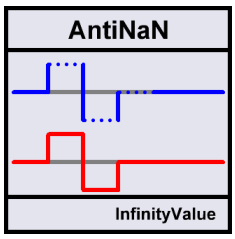
Namensbereich „FuncSegment“ → Seite 133

Namensbereich „JustInTime“ → Seite 159

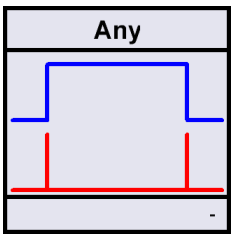
Funktionsblock-Verzeichnis (alphabetisch sortiert)



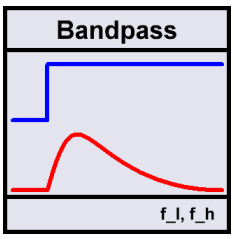
→ Seite 117



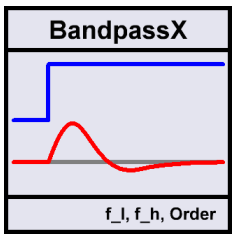
→ Seite 129



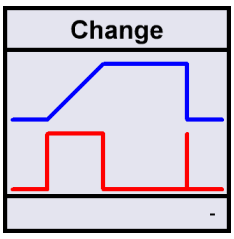
→ Seite 101



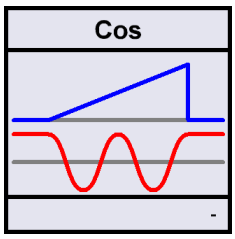
→ Seite 67



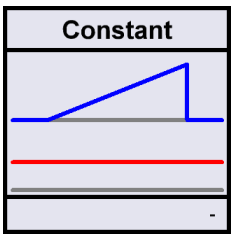
→ Seite 69



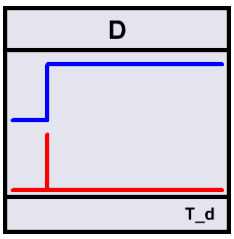
→ Seite 107



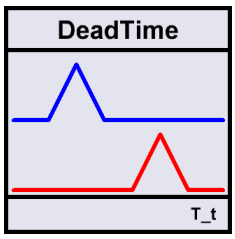
→ Seite 157



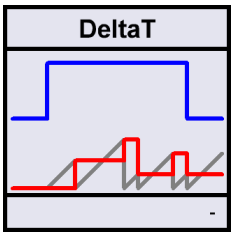
→ Seite 135



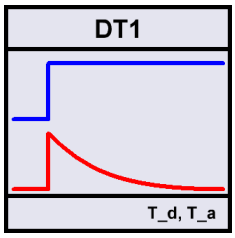
→ Seite 53



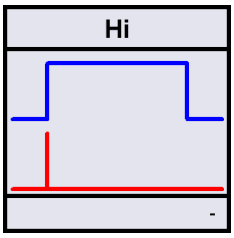
→ Seite 71



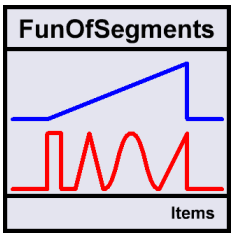
→ Seite 113



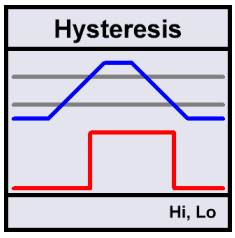
→ Seite 63



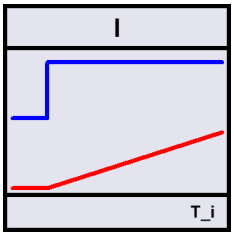
→ Seite 97



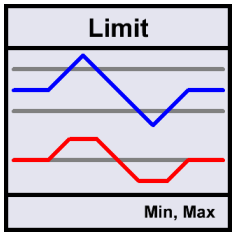
→ Seite 131



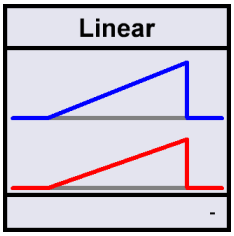
→ Seite 109



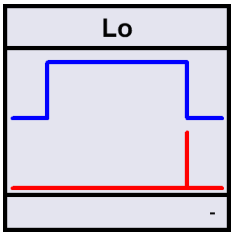
→ Seite 51



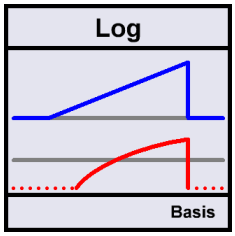
→ Seite 125



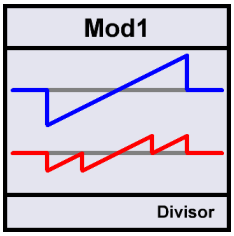
→ Seite 137



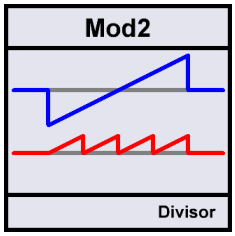
→ Seite 99



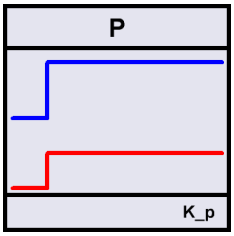
→ Seite 145



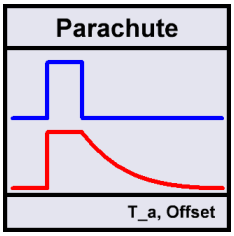
→ Seite 121



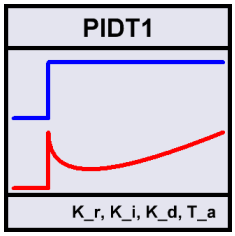
→ Seite 123



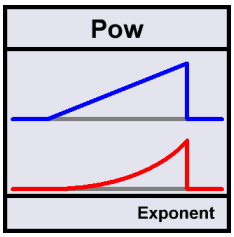
→ Seite 49



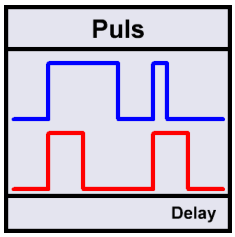
→ Seite 65



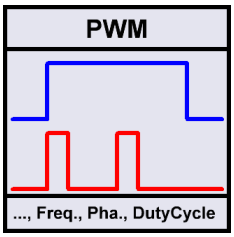
→ Seite 55



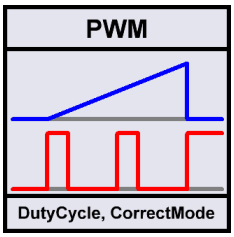
→ Seite 143



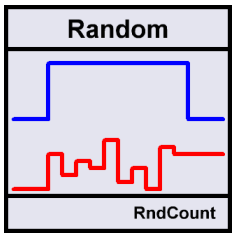
→ Seite 91



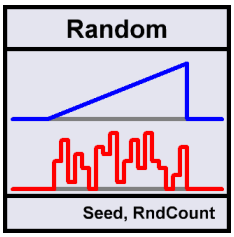
→ Seite 81



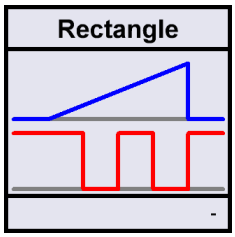
→ Seite 149



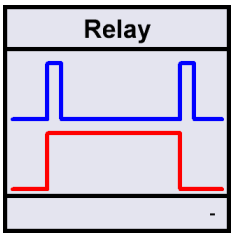
→ Seite 111



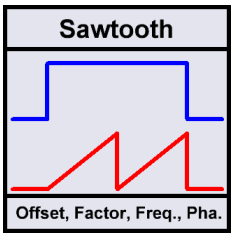
→ Seite 141



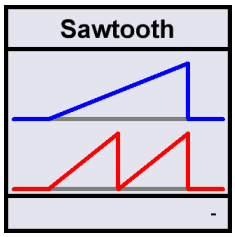
→ Seite 147



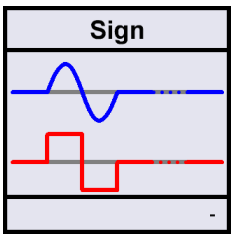
→ Seite 103



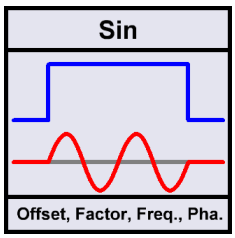
→ Seite 77



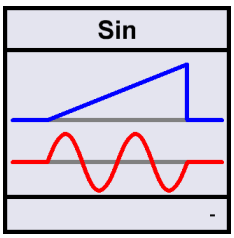
→ Seite 151



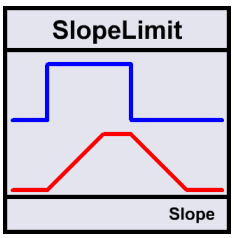
→ Seite 119



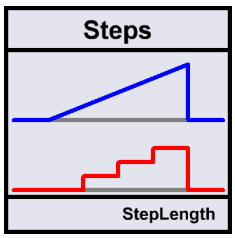
→ Seite 83



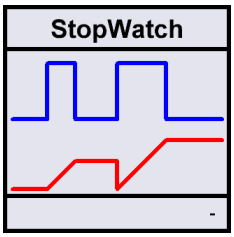
→ Seite 155



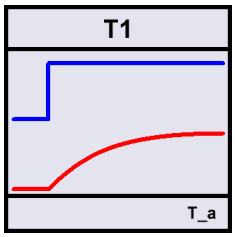
→ Seite 73



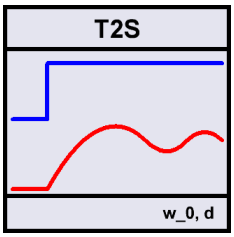
→ Seite 139



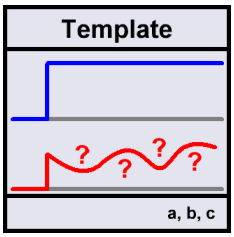
→ Seite 93



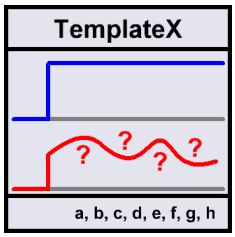
→ Seite 59



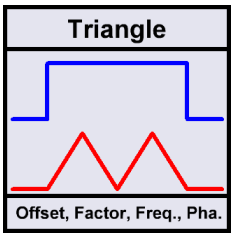
→ Seite 61



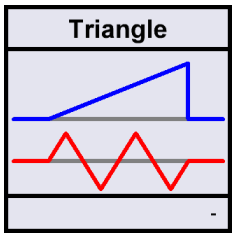
→ Seite 161



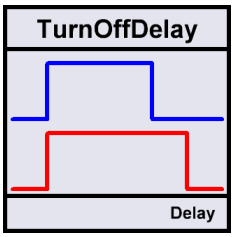
→ Seite 163



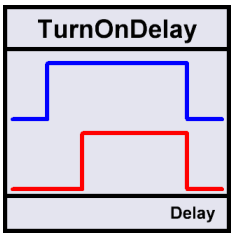
→ Seite 79



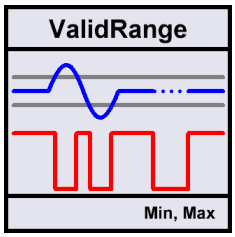
→ Seite 153



→ Seite 89

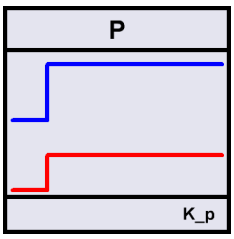


→ Seite 87

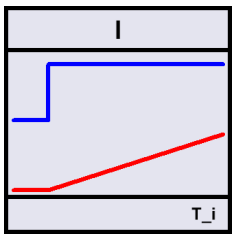


→ Seite 127

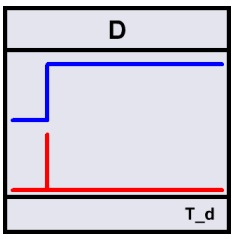
Funktionsblock-Verzeichnis (nach Namensbereich sortiert)



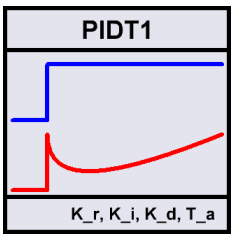
→ Seite 49



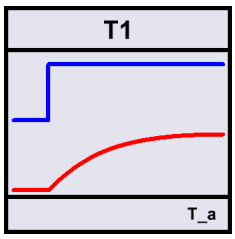
→ Seite 51



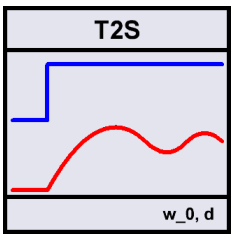
→ Seite 53



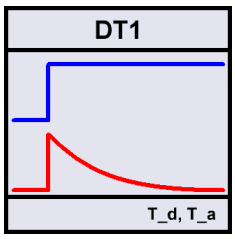
→ Seite 55



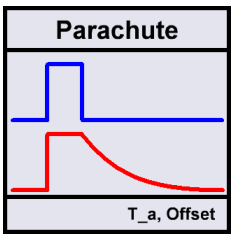
→ Seite 59



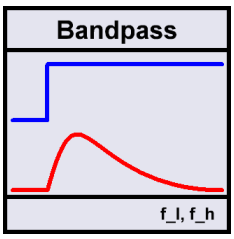
→ Seite 61



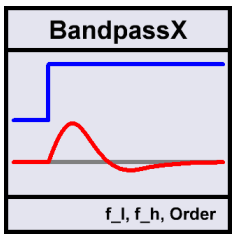
→ Seite 63



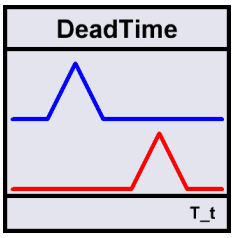
→ Seite 65



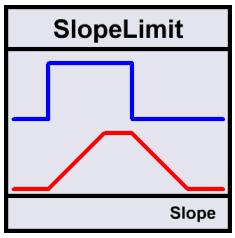
→ Seite 67



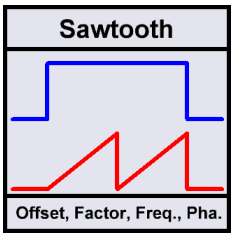
→ Seite 69



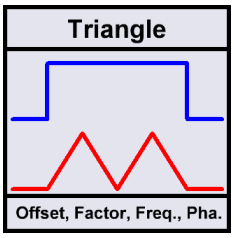
→ Seite 71



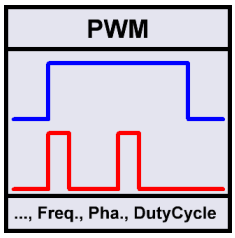
→ Seite 73



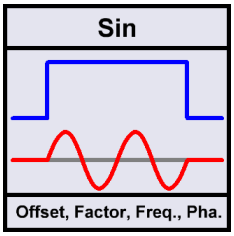
→ Seite 77



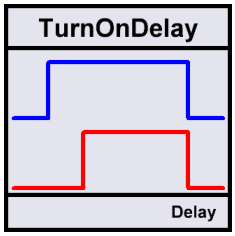
→ Seite 79



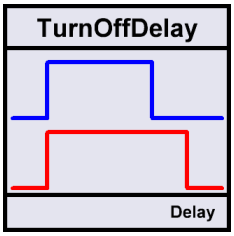
→ Seite 81



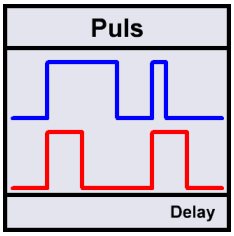
→ Seite 83



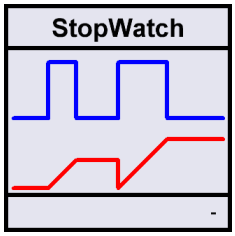
→ Seite 87



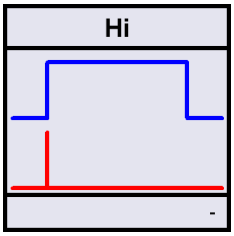
→ Seite 89



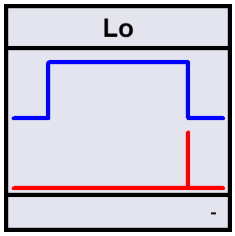
→ Seite 91



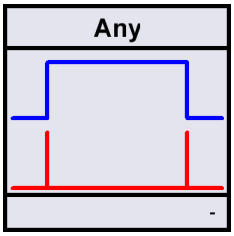
→ Seite 93



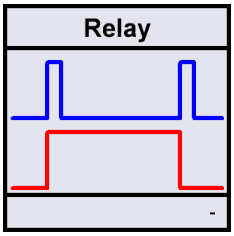
→ Seite 97



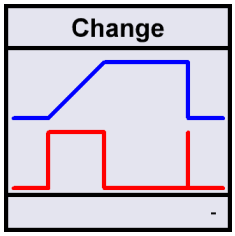
→ Seite 99



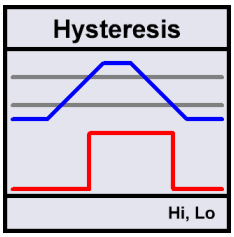
→ Seite 101



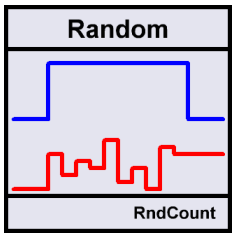
→ Seite 103



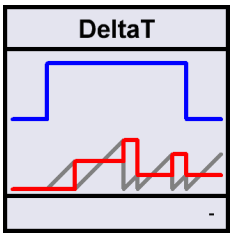
→ Seite 107



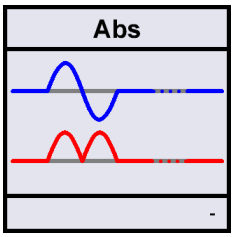
→ Seite 109



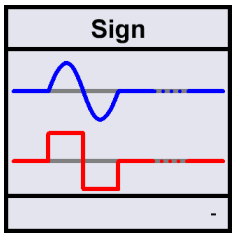
→ Seite 111



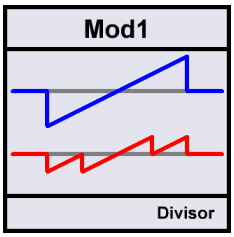
→ Seite 113



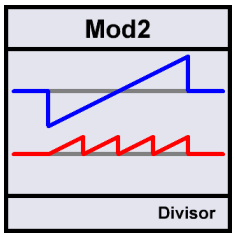
→ Seite 117



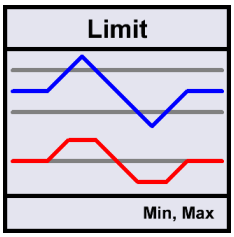
→ Seite 119



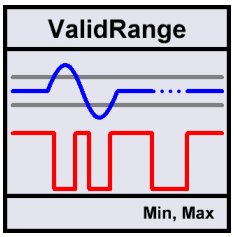
→ Seite 121



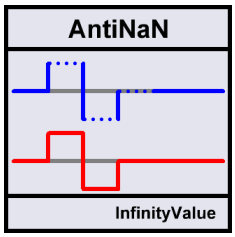
→ Seite 123



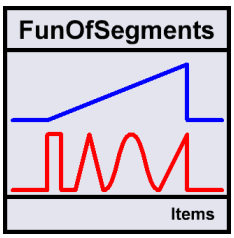
→ Seite 125



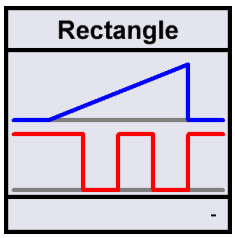
→ Seite 127



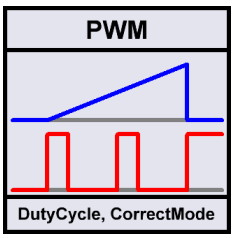
→ Seite 129



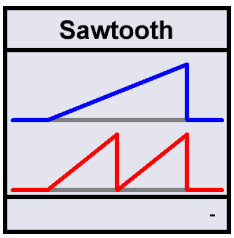
→ Seite 131



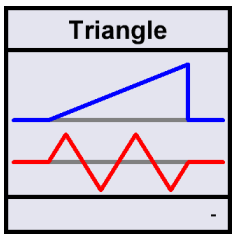
→ Seite 147



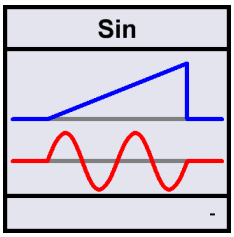
→ Seite 149



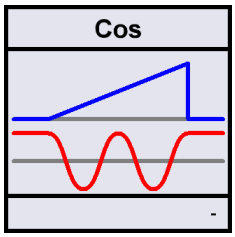
→ Seite 151



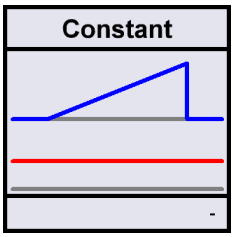
→ Seite 153



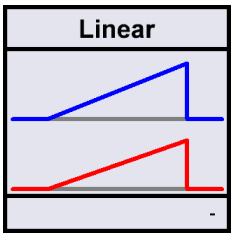
→ Seite 155



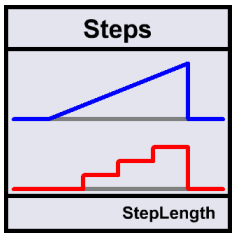
→ Seite 157



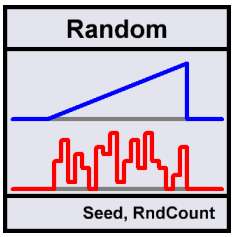
→ Seite 135



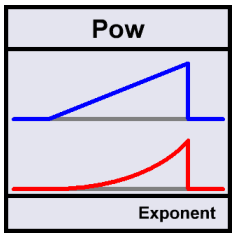
→ Seite 137



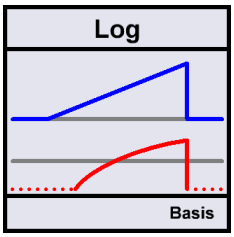
→ Seite 139



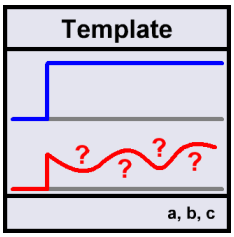
→ Seite 141



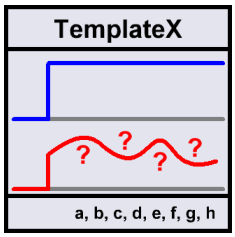
→ Seite 143



→ Seite 145



→ Seite 161



→ Seite 163