Zusammenfassung IAUT

Eric Lindegger – HS25

Danke fürs Herunterladen und Verwenden meiner IAUT Zusammenfassung! Hoffe ihr findet sie nützlich. Falls ihr Fehler findet oder Inputs habt, schreibt mir doch auf eric.lindegger@stud.hslu.ch

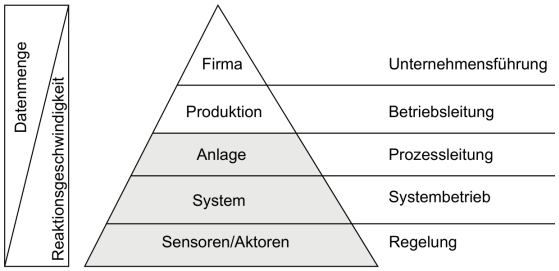
Inhaltsverzeichnis

1 Automatisierung

Definition Automatisierung

Unter Automatisierung (DIN19222) versteht man das gezielte Ausrüsten einer Einrichtung, so dass sie ganz oder teilweise ohne Mitwirkung des Menschen geschieht und arbeiten kann.

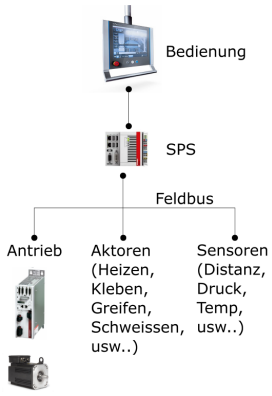
Die Automatisierungspyramide ist der Versuch die gesamte Automatisierungskette auf ein Bild abzubilden:



Automatisierungspyramid

Die Darstellung ist wie folgt zu interpretieren: Je weiter unten, desto schneller müssen die Komponenten reagieren und je weiter oben, ist die Datenmenge grösser. Die Absicht der Automatisierung ist, dass die automatisierte Anlage bis ins ERP der Firma eingepflegt wird. So muss z.B. keine spezielle Benutzeroberfläche geladen werden um ein neuer Prozess auf der Anlage zu starten.

1.1 Steuerung - Überblick



Die Bedienung ist wo das GUI (Graphical User Interface) dargestellt wird. Hier werden vom Benutzer Eingaben getätigt, wie z.B. Job oder Rezeptwahl.
Der Controller innerhalb der Steuerung/Anlage ist die CPU/SPS. Eine Automatisierung ist häufig eine (PID-)Regelungsaufgabe. Die SPS ist mit Aktoren und Sensoren verbunden:

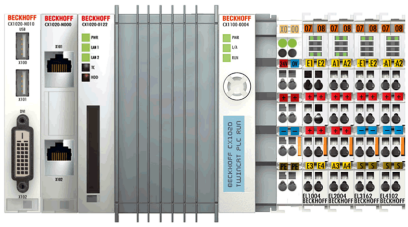
- Sensoren liefern Informationen
- Aktoren lösen Aktion aus

Die Schnittstellen der Sensoren/Aktoren ist entweder eine Spannung oder einen Strom. Die Spannung ist jedoch eher weniger gängig im Moment, da die Spannung anfälliger auf Störeinflüsse ist als ein Strom. Die gängigsten Schnittstellen sind. Ein weiterer Vorteil der Stromschnittstelle ist, dass auch ein Kabelbruch detektiert werden kann, da der Strom nie 0 wird:

- $0V - 10V$
- $4mA - 20mA$

1.1.1 Klemmen

Die Klemmen stellen die Ein- und Ausgänge einer SPS dar und können auf der Feldebene der SPS über ein Bussystem erweitert werden:



Klemmen können über RS485 abgesetzt werden und an einem anderen Ort als direkt bei der SPS eingesetzt werden.

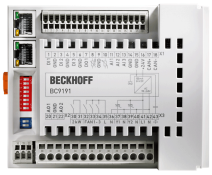
1.2 Allgemeine Aufgaben

Echtzeitanforderung	Regler Prozessabläufen usw.
keine Echtzeitanforderung	Mensch-Maschine-Interface (HMI) Produktionsplanung Archivierung usw.

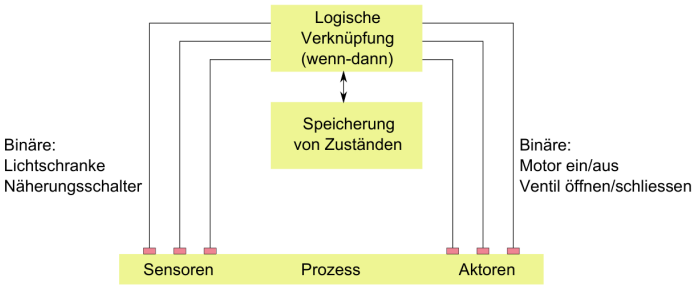
1.3 Allgemeine Anforderungen

Echtzeitfähigkeit	Betriebsysteme Speicherarchitektur
Ein- Ausgabe von Prozesssignalen	Aktoren Sensoren Kommunikationssysteme
Sicherheit Zuverlässigkeit	Hochwertige Komponente Redundanz
Resistenz gegen Umwelteinflüsse	Spezielle Gehäuse Schaltschrankmontage

2 SPS

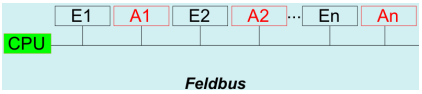


Eine SPS ist eine 'Speicherprogrammierbare Steuerung'. Auf Englisch wird sie PLC (Programmable Logic Controller) genannt. Sehr wichtig: die SPS arbeitet konventionell möglichst auf if/else Statements und verarbeitet sämtlicher Code in jedem Zyklus einmal. Das Arbeitsprinzip ist:



2.1 Feldbus

Der Feldbus ist die Kommunikationsschnittstelle der SPS zu den Klemmen und entsprechend Ein- und Ausgängen. Es hängen alle E/A seriell an einem Bus. Wo der Feldbus aktiv ist, wird Feldebene genannt. Der Aufbau des Feldbusses kann wie folgt vorgestellt werden:



Definition Feldbus

Ein Feldbus verbindet in einer Anlage Feldgeräte wie Sensoren und Aktoren zwecks Kommunikation mit einem Steuerungsgerät (SPS).

Durch den Feldbus entstehen einige wesentliche Vor- und Nachteile:

\+ geringer Verkabelungsaufwand
\+ Erweiterungen oder Änderungen sind einfach

\– Komplexität
\– Preis
\– Aufwendige Messgeräte (Analyzer)
\– Längere Reaktionszeit

2.1.1 Spezifische Feldbusse

Im Gebäude sind die gängigen Feldbusse:

- LON
- EIB
- KNX

Innerhalb der Maschine werden andere Feldbusse verwendet:

- CAN/CANOpen
- Profibus, Profinet
- EtherCAT
- Ethernet/IP

3 Programmierung

Definition SPS-Programmierung

Die EN 61131-3 ist die einzige weltweit gültige Norm für Programmiersprachen von SPS-Steuerungen. Sie definiert die folgenden fünf Sprachen:

- AWL - Anweisungsliste (Assemler)
- KOP - Kontaktplan
- FBS - Funktionsbausteinsprache
- ST - Struktureierter Text
- AS - Ablaufsprache

Die einzige (relevante) Programmiersprache hinsichtlich des IAUT Unterrichts ist ST. Die anderen werden für einfache logische Verknüpfungen gebraucht, oder für z.B. I/O-Zuweisungen. ST wird in Europa oft für die Programmierung von SPS gewählt.

3.1 EN 61499

Diese Norm stellt eine objektorientierte Erweiterung der EN61131 dar. Dieses Kapitel fasst den Aufbau und Syntax der SPS-Programmiersprache ST zusammen.

3.2 Grundaufbau

Der Aufbau eines Programmes/Funktionsblockes oder Funktion ist immer in zwei Bereiche aufgeteilt: Deklaration und Implementation. Im Deklarationsteil werden alle Inputs, Outputs deklariert und ggf. initialisiert. In der Implementation wird der eigentliche Code geschrieben. Wichtig: die Implementation wird in jedem Clock Zyklus wiederholt.

```
1  // Deklarationsteil
2  VAR_INPUT
3      // input-variablen
4      iVarIn AT %I* : INT;
5  END_VAR
6  VAR_OUTPUT
7      // output-variablen
8      iVarOut AT %Q* : INT;
9  END_VAR
10 VAR_IN_OUT
11     // in/out-variablen
12 END_VAR
13 VAR
14     // lokale variablen
15     tmp : INT;
16 END_VAR
17
18 // Implementationsteil
19 IF iVarIn > 10 THEN
20     tmp := 0;
21 ELSE
22     tmp := iVarIn;
23 END_IF
```

3.2.1 Programm

Programme können als Funktionen aufgerufen werden und besitzen keinen Rückgabewert.

```
PROGRAM ProgramName
VAR
    bTrig : BOOL;
    iVal : INT;
END_VAR

IF(bTrig = TRUE) THEN
    iVal := 15;
END_IF
```

3.2.2 Funktionsblock

Ein Funktionsblock ist Zustandsbehaftet: Interne Variablen behalten ihren Wert über die SPS-Zyklen weg. Jede Instanz besitzt einen eigenen Speicher. Jedoch müssen die Speicher instanziert werden.

```
FUNCTION_BLOCK Count
VAR_INPUT
    bTrig : BOOL;
END_VAR
VAR
    iCount : INT;
END_VAR

IF bTrig = TRUE THEN
    iCount := iCount + 1;
END_IF
```

3.2.3 Funktion

Eine Funktion kann ohne Instanzierung verwendet werden. Eine Funktion besitzt keine statische Werte, dafür einen Rückgabewert, welcher auf den Funktionsnamen zugewiesen wird.

```
FUNCTION fFunc : INT // Ruckgabewert
VAR_INPUT
    iTime : INT;
    iT : INT;
END_VAR

iTmp := iTime;
fFunc := iTime / iT;
```

3.3 Variablentypen

3.3.1 System Datentypen

Die gängigen Datentypen sind auch in ST vorhanden. Die Tabelle fasst die Datentypen zusammen:

Typ	Min	Max	Bits	Bemerkung
bool	0	1	8	Nur 1 / 0 / TRUE / FALSE sind gültig!
BYTE	0	255	8	
WORD	0	65535	16	
DWORD	0	4294967295	32	
SINT	-128	127	8	
USINT	0	255	8	
INT	-32768	32767	16	
UINT	0	65535	16	
DINT	-2147483648	2147483647	32	
UDINT	0	4294967295	32	
REAL	$-3.4 \cdot 10^{38}$	$-3.4 \cdot 10^{38}$	64	
LREAL	$-1.8\text{E}+308$	$1.8\text{E}+308$	64	
STRING	–	–	$n+1$	fügt Nullterminierung an.
TIME	T#0ms	T#71582m 47s295ms	32	
TOD	TOD#00:00	TOD#1193:02:47.295	32	
DATE	D#1970-01-01	D#2106-02-06	32	
DT	DT#1970-0-0-00:00	DT#2106-02-06-06:28:15	32	

3.3.2 Array

```
VAR
    aValues : ARRAY[0..99] OF INT;
END_VAR
```

3.3.3 DUT - Data Unit Type

Über DUT können benutzerdefinierte Datentypen angelegt werden. Die relevanten sind STRUCT und ENUM. Durch diese Typen kann ein Programm bedeutend lesbarer gestaltet werden. Für die Erstellung eines DUTs wird ein neues File in der DUT Ordnerstruktur erstellt. Die Definition geht wie folgt:

```
// Struct
TYPE stDataType:
STRUCT
    iVar1 : INT;
    iPointX : INT;
    iPointY : INT;
END_STRUCT
END_TYPE
```

```
// Enumeration
TYPE eSystemState:
(
    None, // = 0 - default
    init, // = 1
    run
) := init; // default state
END_TYPE
```

3.3.4 Globale Variablen Liste (GVL)

Die GVL deklariert Variablen, welche in jedem Programm, Funktionsblock und Funktion aufrufbar. Vielfach machts Sinn die In-/Outputs, welche mit der Hardware kommunizieren in einer GVL zu deklarieren.

3.3.5 Type Casting

Variablentypen können mit Hilfe von Casting Funktionen umgewandelt werden. Fast alle Typen können gewandelt werden:

```
rRealVal := INT_TO_REAL(iVal);
```

Casting ist die zielführende Methode, um keine Fehler im Implementieren des Programms zu generieren. TwinCat hat implizite Casts nicht so gerne.

3.4 Conditional Statements

3.4.1 If/Else

```
IF <cond1> THEN
    ...
ELSIF <cond2> THEN
    ...
ELSE
    ...
END_IF
```

3.4.2 Switch-Case

```
CASE iVar OF
    1:
        // do something for 1
    2..5:
        // do something for range 2 - 5
ELSE
    // do this as default-assignment
END_CASE
```

Die Switch-Bedingung kann auch für Enum-Typen gebraucht werden:

```
CASE eState OF
    Init:
        // handle init state
    Running:
        // handle running state
    Error:
        // handle error state
ELSE
    // it is beneficial to always have a default
    statement
END_CASE
```

3.5 Schleifen

Wie in C gibt es in TwinCat Schleifen. Dabei ist es von zentraler Bedeutung, dass diese Schleife nicht Endlos ist und eine Abbruchbedingung enthält. Ansonsten kann es vorkommen, dass auf die SPS nicht mehr zugegriffen werden kann. Die verschiedenen Schleifen werden hier aufgeführt.

3.5.1 WHILE

WHILE-Schleife

Sofern die Bedingung beim ersten Erreichen nicht zutrifft, wird der Inhalt der Schleife nicht ausgeführt.

```
WHILE iCounter <> 0 DO
    iVar1 := iVar2;
    iCounter := iCounter - 1;
END_WHILE
```

3.5.2 REPEAT

REPEAT-Schleife

Diese Schleife wird mindestens einmal durchgeführt.

```
REPEAT
    iVar1 := iVar2;
    iCounter := iCounter - 1;
UNTIL
    iCounter = 0;
END_REPEAT
```

3.5.3 FOR

```
FOR iCounter = 1 TO 5 BY 1 DO
    iVar1 := iVar2;
END_FOR
```

3.5.4 Schleifen Modifikatoren

Mit diesen Modifikatoren kann wie in C beeinflusst werden, wie das Verhalten in der Schleife ist.

```
EXIT        // Ausbruch aus aktuellem Loop
CONTINUE    // Direkter Sprung zu Loop Anfang
```

4 TwinCat

TwinCat ist die Entwicklungsumgebung von Beckoff für deren SPS Systeme. TwinCat liefert diverse Funktionen vorgefertigt für den Benutzer. In diesem Kapitel werden die relevantesten aufegelistet.

Semicolons

TwinCat ist sehr tolerant was Semicolons nach `END_*` Anweisungen betrifft. Gemäss Standard wird jede Anweisung mit ; abgeschlossen. TwinCat toleriert das nicht Setzen dieser Semicolons.

4.1 Bitshifting

Mit Bitshifting Operationen können sämtliche Bits innerhalb einer Variable nach links oder rechts geshiftet werden. Die Bits am Rand werden verworfen und neu entstandene Lücken mit 0 aufgefüllt.

```
res := SHL(bByte, 2);    // Shift Left
res := SHR(bByte, 2);    // Shift Right
```

Mit Bitrotate Befehlen können herausgeschobene Bits rotiert werden (herausgeschobene Bits werden am anderen Ende wieder hineingeschoben).

```
res := ROL(bByte,2);     // Circular Shift Left
res := ROL(bByte,3);     // Circular Shift Right
```

Die Shift-Operationen können mit allen gängigen Datentypen verwendet werden.

4.2 Limit

Die Limit-Funktion limitiert die Eingangsvariable innerhalb von einer Minimal- und Maximalwert. Die Funktion kann wie folgt verwendet werden:

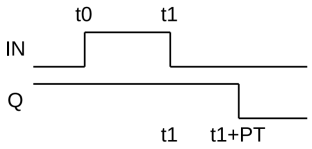
```
rUsat := LIMIT(umin, rU, umax);
// rU wird von umin und umax limitiert
```

4.3 Timer

Verschiedene Timer Module können direkt aus TwinCat instanziiert werden. Diese haben jeweils verschiedene Verhaltensmuster und sind für andere Anwendungen nützlich.

4.3.1 Timer Off-Delay (TOF)

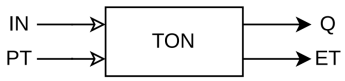
In: Eingangssignal (1 zu 0 startet Timer)
PT: Verzögerungszeit
Q: Ausgang des Timers (1 wenn PT erreicht)
ET: Abgelaufene Zeit



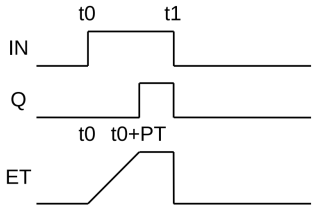

```
VAR
    fbTOF : TOF;
END_VAR

fbTOF(IN := bTrig, PT := T#500ms);
bOut := fbTOF.Q;
```

4.3.2 Timer On-Delay (TON)



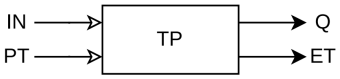
In: Eingangssignal (0 zu 1 startet Timer)
PT: Verzögerungszeit
Q: Ausgang des Timers (1 wenn PT erreicht)
ET: Abgelaufene Zeit



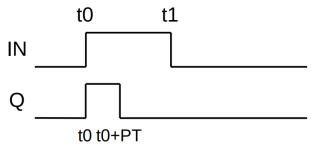

```
VAR
    fbTON : TON;
END_VAR

fbTON(IN := bTrig, PT := T#500ms);
bOut := fbTON.Q;
```

4.3.3 Timer Pulse Generator (TP)



In: Eingangssignal (0 zu 1 startet Timer)
PT: Pulslänge
Q: Ausgang des Timers (1 wenn t < PT)
ET: Abgelaufene Zeit




```
VAR
    fbTP : TP;
END_VAR

fbTP(IN := bTrig, PT := T#500ms);
bOut := fbTP.Q;
```

5 Digitale Regelung

Für Automationsanwendungen werden sehr häufig PID-Regler verwendet. Für eine digitale Regelung muss der Prozess und der Regler diskretisiert werden. Verschiedene Diskretisierungsmethoden existieren.

5.1 Laplace Diskretisierung

Um ein System in Laplace Bereich zu diskretisieren (in den z-Bereich) zu transformieren existieren verschiedene Methoden. Diese sind hier aufgeführt.

5.1.1 Rückwärts-Rechteckregel

$$s = \frac{z-1}{T_S z} = \frac{1-z^{-1}}{T_S}$$

Wobei T_S die Zykluszeit der SPS ist. Diese ist standardmässig 10 ms. Gängigste Art zu diskretisieren

5.1.2 Vorwärtsrechteck-Regel

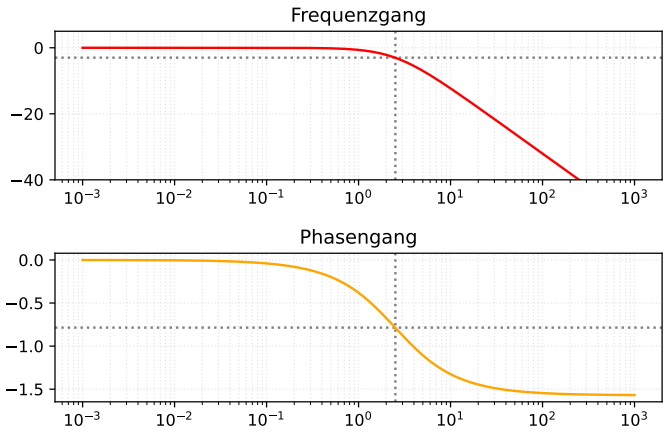
$$s = \frac{z-1}{T_S}$$

Für Regelung eine ungeeignete Diskretisierungsmethode.

5.1.3 Trapezoidal-Regel (Tustin Approximation)

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

5.2 PT1-Prozess

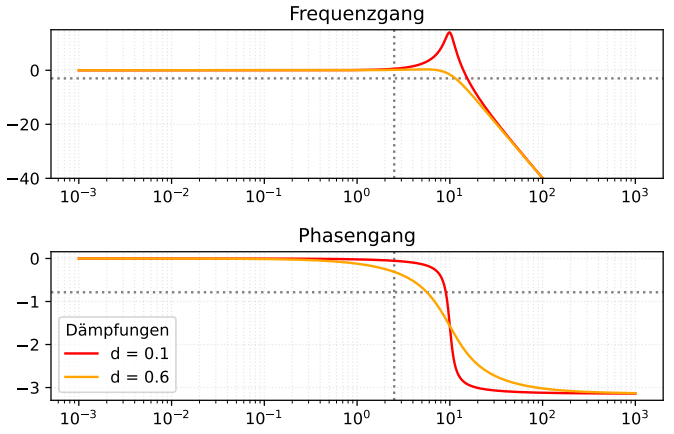


PT1 Glieder besitzen eine Phasenverschiebung von 90° bzw. $\frac{\pi}{2}$. Die Grenzfrequenz liegt bei $\omega_0 = 2\pi\tau$.

$$G(s) = \frac{1}{1+\tau s} \rightarrow y[k] = a_1 x[k] - a_2 y[k-1]$$

$$\text{Mit: } a_1 = \frac{T_S}{T_S+\tau} \quad a_2 = \frac{\tau}{T_S+\tau}$$

5.3 PT2-Prozess



Mit zunehmender Dämpfung d sinkt die Resonanz bei der Grenzfrequenz ω_0. Die Phasenverschiebung ist 180° respektive π.

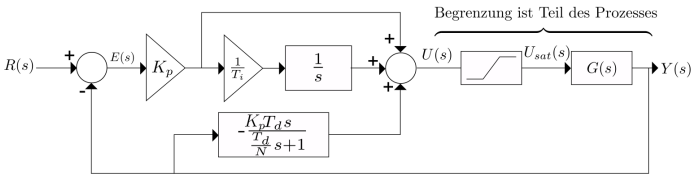
$$G(s) = \frac{\omega_0^2}{s^2 + 2d\omega_0 s + \omega_0^2}$$

Diskretisiert:

$$y[k] = \frac{a_3}{a_1}x[k] - \frac{a_2}{a_1}y[k-1] - \frac{1}{a_1}y[k-2]$$

Mit: $a_1 = 1 + 2d\omega_0 T_s + \omega_0^2 \quad a_2 = -2 - 2d\omega_0 T_s \quad a_3 = T_s^2\omega_0^2$

5.4 Diskreter PID-Regler



Der PID-Regler muss auch diskretisiert werden. Der zeitkontinuierliche PID-Regler hat die Form:

$$C(s) = \frac{U(s)}{E(s)} = k_P\left(1 + \frac{1}{T_i s} + \frac{T_d s}{\frac{T_d}{N}s + 1}\right)$$

Legende:

k_P:	Reglerverstärkung
T_i:	Nachstellzeit [s]
T_d:	Vorhaltzeit [s]
N:	$\frac{T_d}{N}$ ist die Zeitkonstante des Filters

5.4.1 P-Anteil

Der diskretisierte P-Anteil ist somit:

$$u_P[k] = k_P \cdot e[k]$$

5.4.2 I-Anteil

Der diskretisierte I-Anteil des Reglers ist:

$$u_I = u_I[k-1] + \frac{k_P T_s}{T_i}e[k]$$

5.4.3 D-Anteil

Der diskretisierte D-Anteil des Reglers ist:

$$u_D = \frac{k_P \cdot T_d}{T_s}(e[k] - e[k-1])$$

Kompletter diskreter PID-Regler

Der diskretisierte PID-Regler ist damit zusammengesetzt:

$$u = u_P + u_I + u_D$$

Die vorhergehenden Fehler und Stellgrössen müssen immer mitgeführt werden.

5.5 Stellgrössensättigung

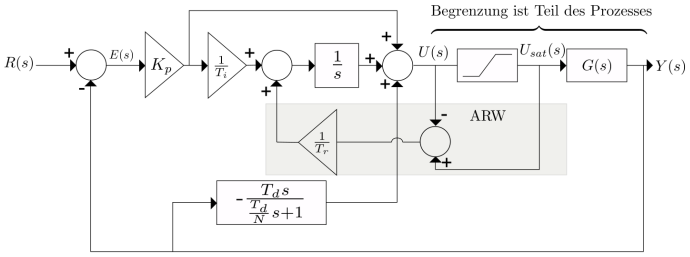
Fast alle Stellgrössen sind in ihrem Output limitiert. Damit die Hardware keinen Schaden nehmen kann, muss die Stellgrösse in der Software gesättigt werden. Die Sättigungsfunktion funktioniert anhand eines einfachen mathematischen Prinzips:

$$u_{sat}[k] = \begin{cases} u_{max}, & \text{wenn } u[k] > u_{max} \\ u_{min}, & \text{wenn } u[k] < u_{min} \\ u[k], & \text{sonst} \end{cases}$$

Die Stellgrössensättigung generiert ein neues Problem: der I-Anteil kann sich in die Unendlichkeit aufintegrieren. Um das Problem einzugrenzen, wird ein Antireset-Windup (ARW) implementiert. Das Sättigungsverfahren wird in ST wie folgt gelöst:

```
1 usat := LIMIT(umin, u, umax);
```

5.5.1 Antireset-Windup (ARW)



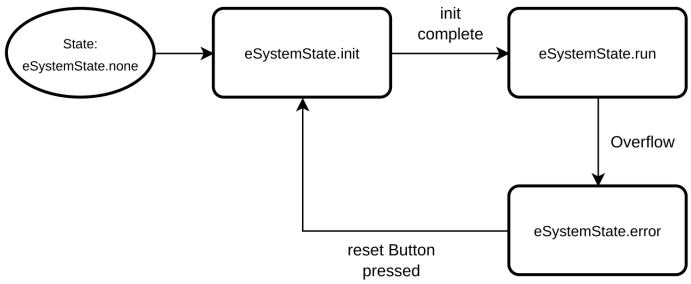
Wird ein ARW implementiert, integriert sich der I-Anteil nicht mehr über das Limit heraus. Der Regler ist damit viel reaktiver. Der I-Anteil mit ARW ist somit:

$$u_I[k] = u_I[k-1] + \frac{k_P T_s}{T_i}e[k] + \frac{T_s}{T_r}(u_{sat}[k] - u[k])$$

Wobei $u[k]$ die gesamte PID-Stellgrösse ohne Sättigung ist.

6 FSM - Finite State Machine

Finite State Machines oder FSM sind Zustandsmaschinen, welche je nach Zustand Ein- und Ausgänge anders behandeln. Wichtig dabei sind die Übergangsbedinungen, zu welchen der Zustand wechselt. Dargestellt werden sie in Zustandsdiagrammen. Ein einfaches Diagramm:



In ST ist es üblich, dass jeweils ein Enumerationstyp generiert wird, welcher die Zustände im code lesbarer macht:

```
TYPE eSystemState:
(
    none,
    init,
    run,
    error
)
END_TYPE
```

Im Programm geht das Ganze dann wie folgt von statten:

```
PROGRAM
VAR
    state : eSystemState;
    iTemp : REAL;
END_VAR

CASE state OF
    init:
        // handle init
        ....
        // next state
        IF iTemp >= 20 THEN
            state := run;
        END_IF
    run:
        // handle run
        ....
        // next state
        IF iTemp > 100 THEN
            // overtemperature protection
            state := error;
        END_IF
    error:
        // handle error
        ....
        IF bReset = TRUE THEN
            state := init;
        END_IF
ELSE
    state := init;
END_CASE
```

7 Beispiel Code

Ein paar Beispiel Codes können hier entnommen werden.

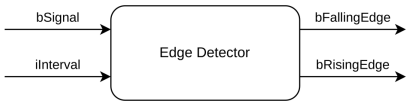
7.1 Flankendetektor

Damit ein Button nur einmal als gedrückt wahrgenommen wird, muss eine Flankendetektion implementiert werden. So kann auch darauf reagiert werden, wenn die Taste losgelassen wird. Für die Lesbarkeit, wird ein Enumerationstyp definiert.

7.1.1 Enumeration

```
// fuer die Button Flanken
TYPE sButtonState :
STRUCT
  bState : BOOL;
  bRisingEdge : BOOL;
  bFallingEdge : BOOL;
END_STRUCT
END_TYPE
```

7.1.2 Funktionsblock




```
FUNCTION_BLOCK EdgeDetector
VAR_INPUT
  bSignal : BOOL;
  iInterval : INT;
END_VAR
VAR_OUTPUT
  bFallingEdge : BOOL;
  bRisingEdge : BOOL;
END_VAR
VAR
  bPrevSignal : BOOL;
  state : eStateEdgeDetect;
  interval : INT;
END_VAR

IF bPrevSignal AND NOT bSignal THEN
  // falling edge detected
  bFallingEdge := 1;
  bRisingEdge := 0;
  state := eStateEdgeDetect.one_press;
ELSIF NOT bPrevSignal AND bSignal THEN
  // rising edge detected
  bFallingEdge := 0;
  bRisingEdge := 1;
  state := eStateEdgeDetect.one_press;
ELSE
  // edge is nor falling or rising
  bFallingEdge := 0;
  bRisingEdge := 0;
END_IF

// store signal for next cycle
bPrevSignal := bSignal;
```

7.2 PID-Regler

7.2.1 Parameter Struct

```
TYPE sPID_Params :
STRUCT
  Kp      : REAL; // Proportional Faktor
  Ti      : REAL; // Nachstellzeit
  Td      : REAL; // Vorhaltzeit
  Tr      : REAL; // ARW Zeitkonstante
  N       : UINT; // Filterkonstante
  ARW     : BOOL; // ARW Enable/Disable
  umin    : REAL; // usat minimal
  umax    : REAL; // usat maximal
END_STRUCT
END_TYPE
```

7.2.2 System Modus Enum

```
TYPE eSystem_Modus :
(
  System_Init := 0,
  System_Run := 1
);
END_TYPE
```

7.2.3 Funktionsblock

```
FUNCTION_BLOCK fbPID
VAR_INPUT
  ref       : REAL;
  y         : REAL;
  Mode      : eSystem_Modus;
  T         : REAL;
  sPID_Par  : sPID_Params;
END_VAR
VAR_IN_OUT

END_VAR
VAR_OUTPUT
  u    : REAL;
  usat : REAL;
END_VAR
VAR
  rParamI  : REAL;
  rParamD1 : REAL;
  rParamD2 : REAL;
  rE       : REAL;
  rEm      : REAL;
  rUp      : REAL;
  rUi      : REAL;
  rUim     : REAL;
  rUd      : REAL;
  rUdm     : REAL;
  u_arw    : REAL;
END_VAR

rE := ref - y;    // momentary error
CASE Mode OF
  eSystem_Modus.System_Init:
    // Parameter for I-Part
    rParamI := sPID_Par.Kp*T / sPID_Par.Ti;
    // Parameter for D-Part
    rParamD1 := (
    sPID_Par.N*sPID_Par.Kp*sPID_Par.Td ) /
    (sPID_Par.Td + sPID_Par.N*T);
    rParamD2 := ( sPID_Par.Td ) / (sPID_Par.Td +
    sPID_Par.N*T);
    u := 0;

  eSystem_Modus.System_Run:
    // P - Part
    rUp := sPID_Par.Kp * rE;
    // I - Part
    rUi := rUim + rParamI * rE;
    // D - Part
    rUd := rParamD1 * (rE - rEm) + rParamD2 * rUdm;
    u := rUp + rUi + rUd;
    // saturating u if out of bounds
    usat := LIMIT(u, sPID_Par.umin, sPID_Par.umax);
    IF sPID_Par.ARW THEN
        rUi := rUi + (1.0 / sPID_Par.Tr) * (usat -
    u);
    END_IF
    rUim := rUi;
        rUdm := rUd;
        rEm := rE;
END_CASE
```

7.3 PT1 Prozess

```
FUNCTION_BLOCK PT1
VAR_INPUT
  signal    : REAL;
  fg        : REAL;
END_VAR
VAR_OUTPUT
  out       : REAL;
END_VAR
VAR
  T         : REAL;
  tau       : REAL;
  prev_out  : REAL;
END_VAR

tau := 1/(2*pi*fg);
T:= 0.01;

out := (T/(T+tau))*signal + (tau/(T+tau))*prev_out;

prev_out := out;
```

7.4 PT2 Prozess

7.4.1 Parameter Struct

```
TYPE dPT1T2_Params :
STRUCT
  K  : REAL;
  T1 : REAL;
  T2 : REAL;
END_STRUCT
END_TYPE
```

7.4.2 Funktionsblock

```
FUNCTION_BLOCK PT1T2_Prozess
VAR_INPUT
  u     : REAL;
  Modus : eSystem_Modus;
  Param : dPT1T2_Params;
```

```
  T     : REAL;
END_VAR
VAR_OUTPUT
  out : REAL;
END_VAR
VAR
  b0    : REAL;
  alph1 : REAL;
  alph2 : REAL;
  a1    : REAL;
  a2    : REAL;
  outm  : REAL;
  outmm : REAL;
END_VAR

CASE Modus OF
  eSystem_Modus.System_Init:
    // alphas
    alph1 := T + Param.T1;
    alph2 := T + Param.T2;
    // funct. parameters
    b0 := (Param.K * T * T) / (alph1*alph2);
    a1 := -( (Param.T1/alph1) + (Param.T2/alph2) );
    a2 := (Param.T1*Param.T2)/(alph1*alph2);

  eSystem_Modus.System_Run:
    out := -a1*outm - a2*outmm + b0 * u;
    outmm := outm;
    outm := out;
END_CASE
```

8 Matlab-Befehle

Für die Auslegung und modellierung wurde in IAUT Matlab verwendet. Dabei wurden Tools wie der 'PID-Tuner' oder 'Siso-Tool' für die Auslegung der Regler eingesetzt. Grundsätzlich folgt der Reglerentwurf diesem Schema:

1. Prozess messen
2. Prozessmodellierung (Sprungantwort parametrisieren, etc.)
3. Regler mit Tool auslegen

9 Hilfsblätter