Automatisierung und
Informationssysteme
TU München

Vorlesung Automatisierungstechnik 1

im Wintersemester 2018/2019

Prof. Dr.-Ing. Birgit Vogel-Heuser

Die heutige VL im Überblick

Projektphasen (zeitliche Sicht)

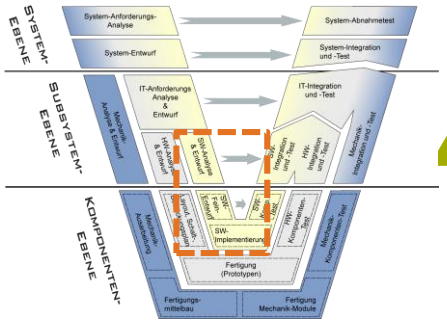
Requirements Analysis | Design | Basic Engineering | Detail Engineering | Implementation / Construction | Commissioning | Production / Operation

Design & Engineering

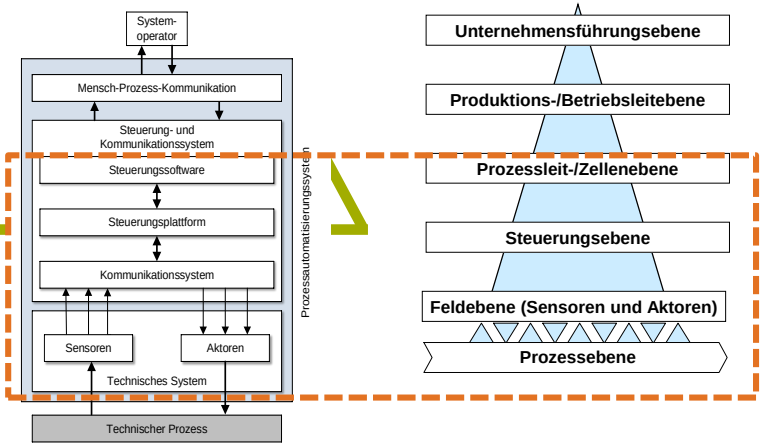
Ziel:
Automatisierte Anlage

V-Modell (Vorgehenssicht)

Software Entwurf & Implementierung



Automatisierungssystem/-pyramide (Automatisierungssicht)

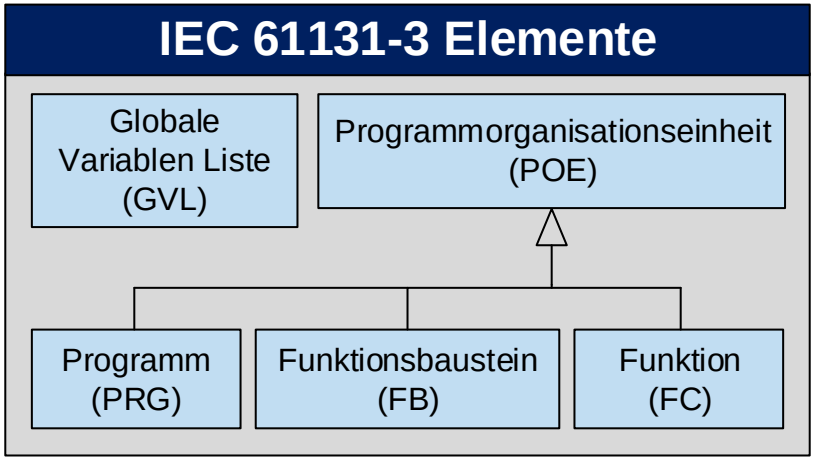


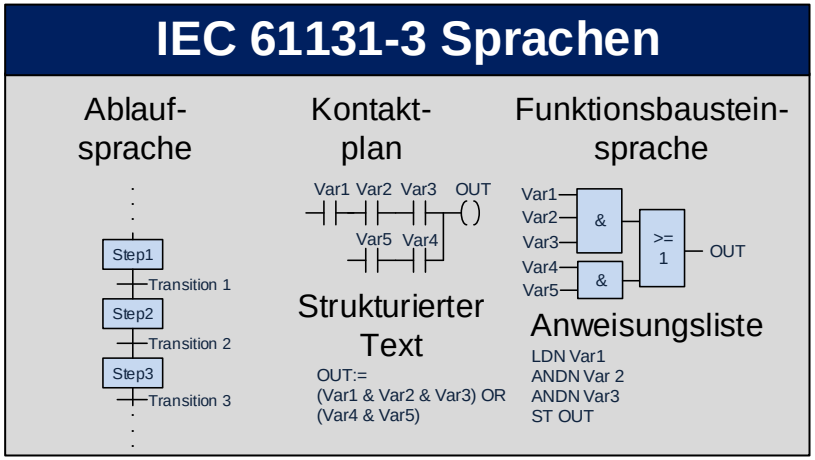
Steuerungssoftware / Steuerungsplattform

Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detail Engineering (Fokus: AT-Hardware)
- Kapitel 5 Detail Engineering (Fokus: AT-Software)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
    - Programmiersprache Kontaktplan (KOP)
    - Programmiersprache Funktionsbausteinsprache (FBS)
    - Programmiersprache Anweisungsliste (AWL)
  - **Objektorientierung und weitere Programmiersprachen der IEC 61131-3**
    - Programmiersprache Ablaufsprache (AS)
    - Programmiersprache Strukturierter Text (ST)

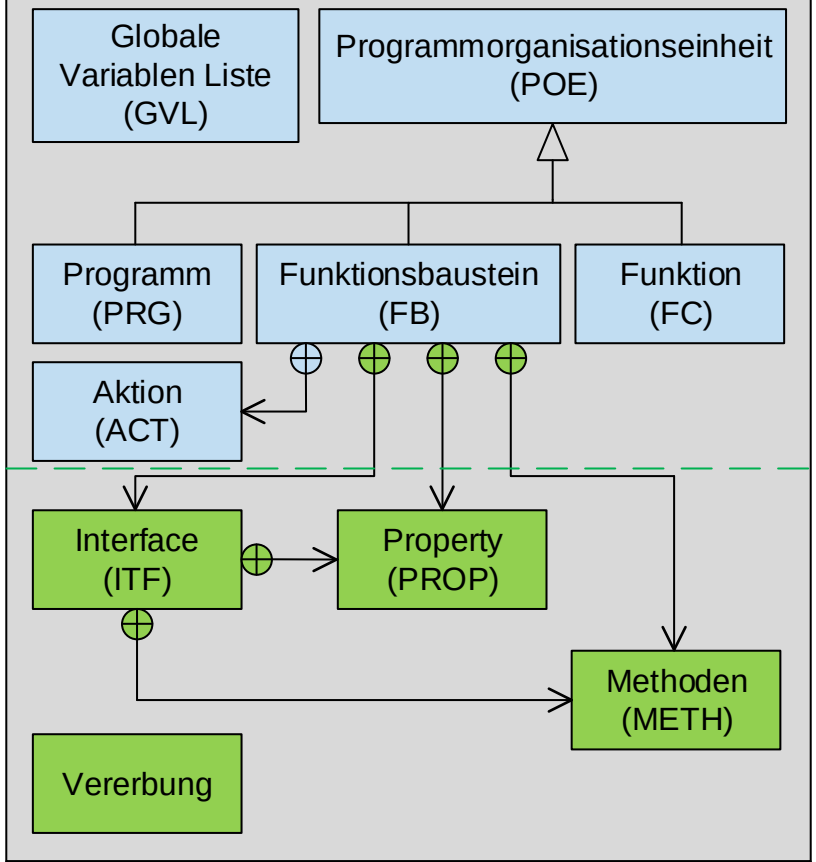
Rückblick: Ursprüngliche IEC 61131-3
Elemente und Sprachen





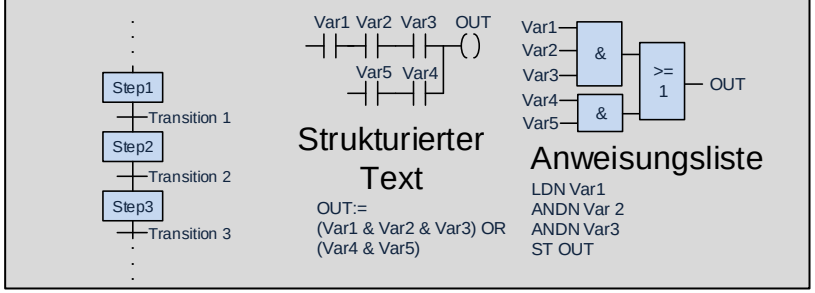
Einführung: Zusätzliche Elemente in der objektorientierten (OO) Erweiterung der IEC 61131-3

IEC 61131-3 Elemente



Zusätzliche Elemente durch die OO-Erweiterung der IEC 61131-3

IEC 61131-3 Sprachen



- Überarbeitung der Programmiernorm (IEC 61131-3, 3rd Edition) im Jahr 2013
  - von vielen Programmiersystemen schon vor Normierung der Elemente unterstützt
- Steuerungssoftware wird immer komplexer, neue Konstrukte zur Unterstützung nötig
- Einführung von Konzepten objektorientierter Programmiersprachen (wie bspw. C++)
- Bessere Möglichkeiten zur Modularisierung, Wiederverwendbarkeit und Wartbarkeit
- Erweiterung unterstützt beispielsweise von

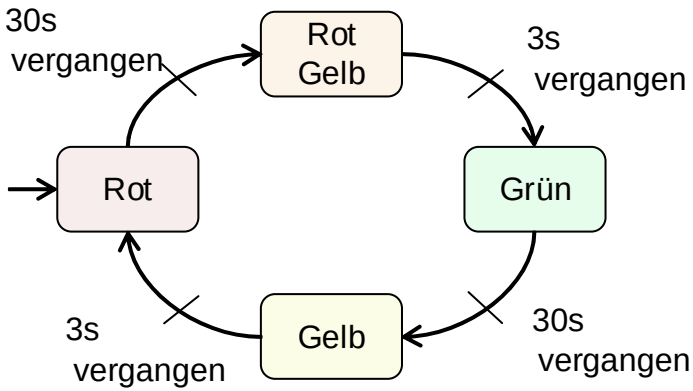
CODESYS
BECKHOFF (TwinCAT3)
Schneider Electric (SoMachine)
Rexroth Bosch Group (IndraWorks)

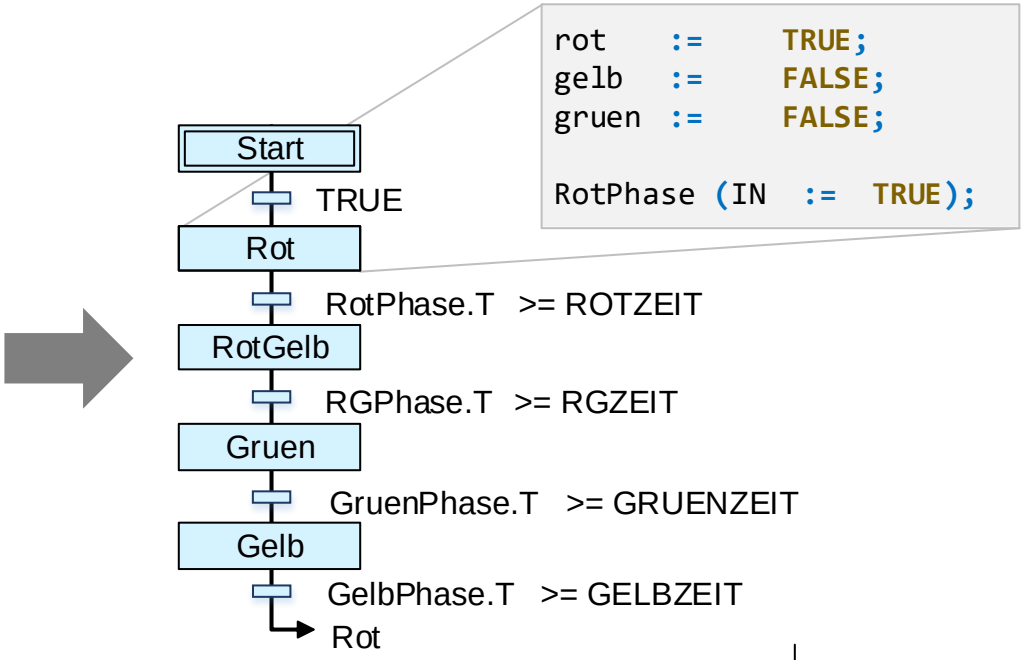
Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detail Engineering (Fokus: AT-Hardware)
- Kapitel 5 Detail Engineering (Fokus: AT-Software)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
    - Programmiersprache Kontaktplan (KOP)
    - Programmiersprache Funktionsbausteinsprache (FBS)
    - Programmiersprache Anweisungsliste (AWL)
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - **Programmiersprache Ablaufsprache (AS)**
    - Programmiersprache Strukturierter Text (ST)

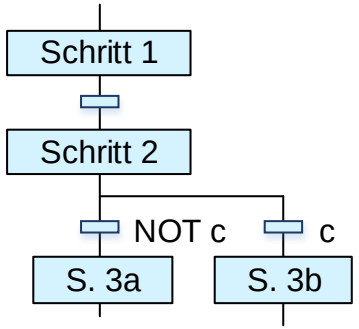
IEC 61131-3 Programmiersprachen – Einführung Ablaufsprache (AS) / Sequential Function Chart (SFC)

- Vor allem für Programmierung von Ablaufsteuerungen (Bei Siemens „S7 GRAPH“ genannt)
- Vergleichbar mit Petri-Netzen
- Zykluswechsel nach Abarbeitung eines Schrittes und prüfen der (gültigen) nachfolgenden Transition





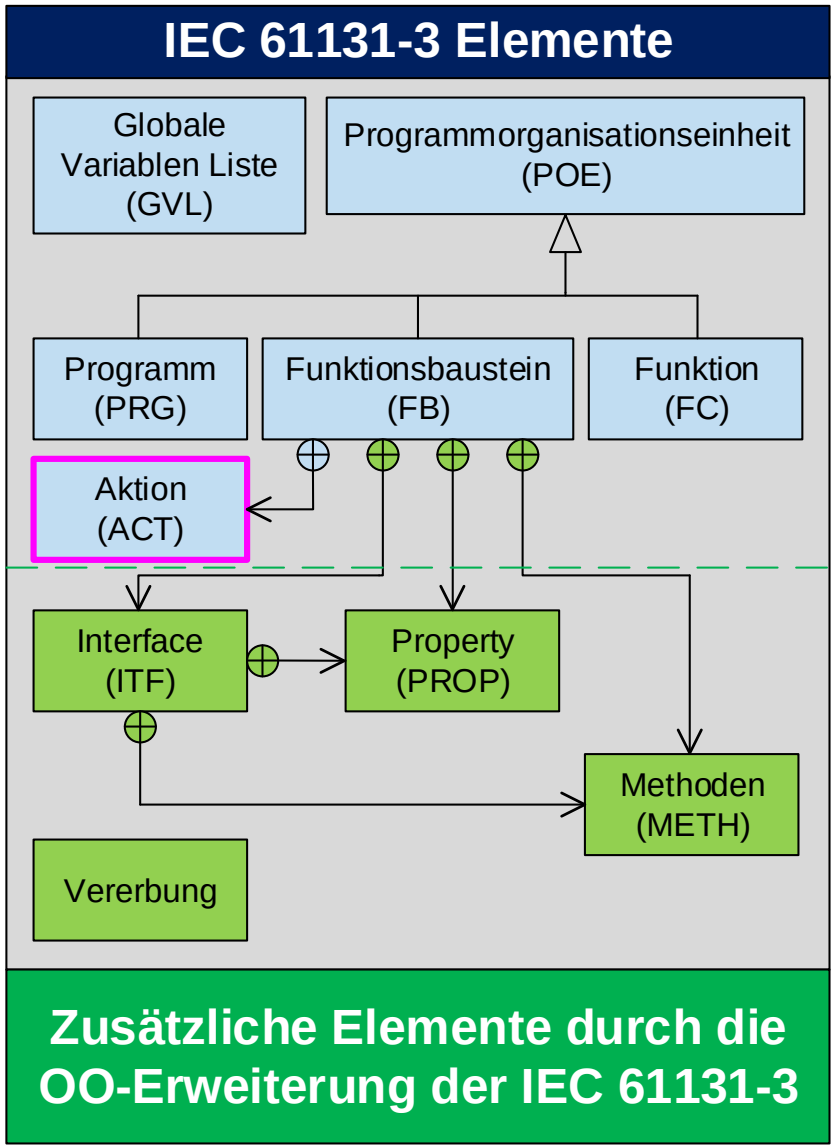
Verzweigungen und parallele Abläufe möglich



Einbinden von Programmcode in einzelne Schritte möglich:

- *Entry-Action*: Wir einmalig beim Betreten des Schrittes ausgeführt
- *During-Action*: Wird bei jedem Zyklus ausgeführt, zu dem der Schritt aktiv ist
- *Exit-Action*: Wird einmalig beim Verlassen des Schrittes ausgeführt

Elemente in der IEC 61131-3 – Aktionen (ACT)



- Allgemeine Code-Abschnitte eines Funktionsbausteins (FB)
- Ein FB kann beliebig viele Aktionen haben
- Nur Implementierung und kein separater Deklarationsteil, aber…
- … voller Zugriff auf die Variablen im FB-Deklarationsteil und auf globale Variablen
- Kein Rückgabewert beim Aufruf einer ACT
- Deklaration zwischen Deklarations- und Implementierungsteil eines FBs

```
FUNCTION_BLOCK FB_Durchfluss
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR

ACTION BeispielAktion :
// Implementierungsteil Aktion
END_ACTION
// Implementierungsteil FB
END_FUNCTION_BLOCK
```

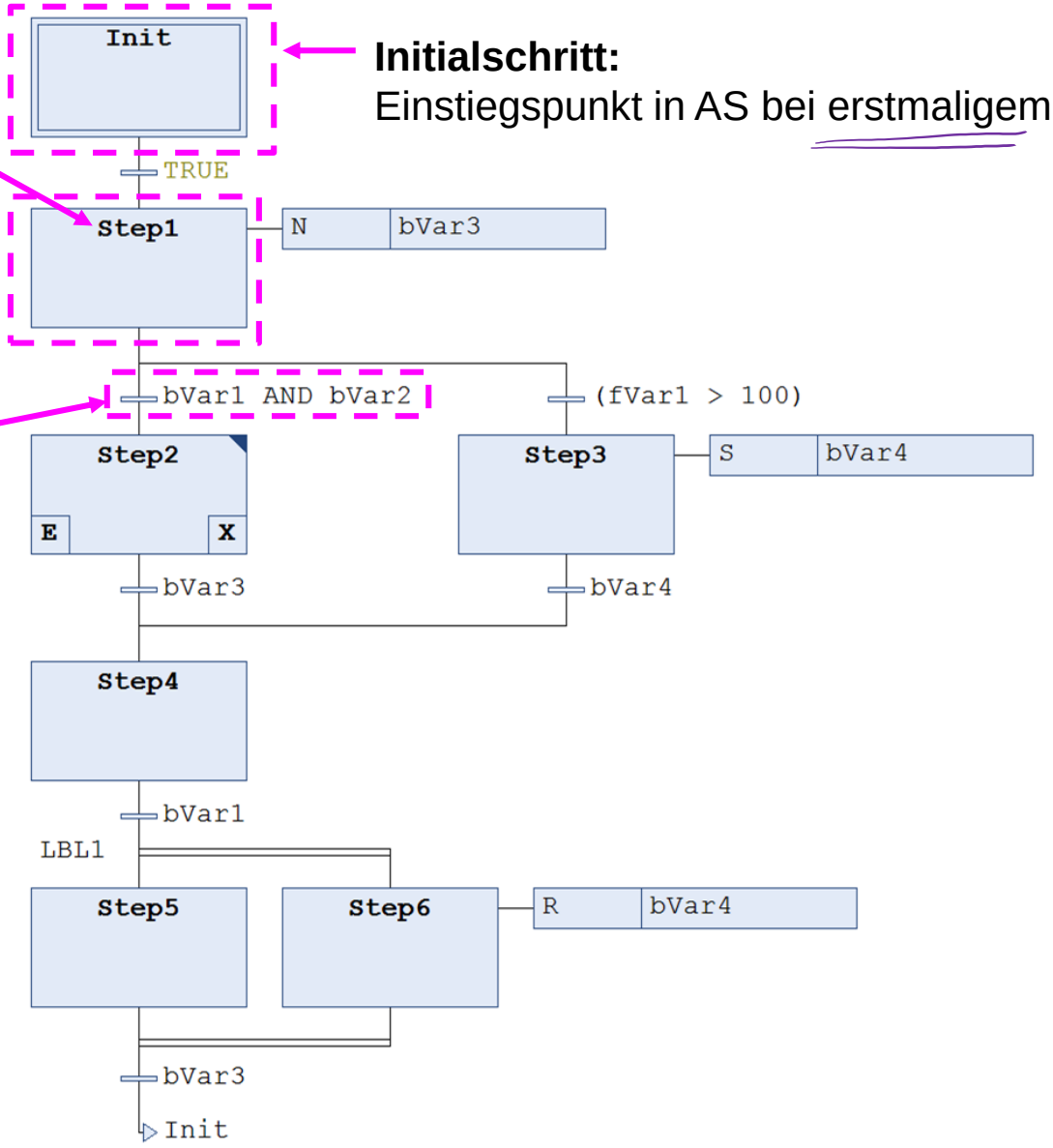
Beispiel für eine Aktion

Beispiel für die Notationselemente in einer Implementierung in Ablaufsprache (AS) (1)

Initialschritt:
Einstiegspunkt in AS bei erstmaligem Aufruf

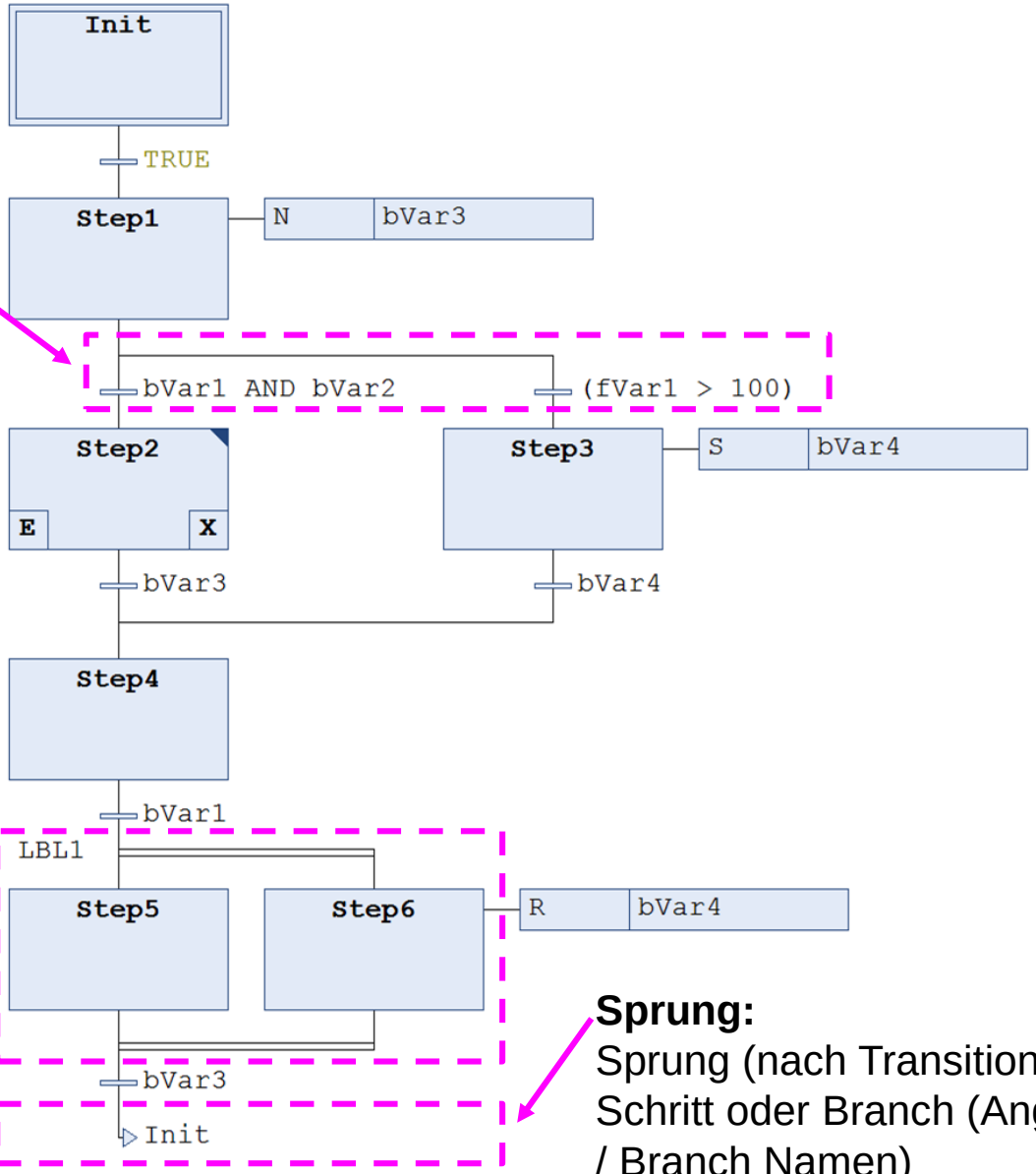
Schritt:
Einzelschritt eines implementierten Ablaufs, Name (Step1) zur Identifikation

Transition:
Weiterschaltbedingung zwischen Schritten, Boolescher Ausdruck, meist in ST



Beispiel für die Notationselemente in einer Implementierung in Ablaufsprache (AS) (2)

Verzweigung:
Alternativen für die Programmbearbeitung, bspw. gut geeignet für Rezeptsteuerungen

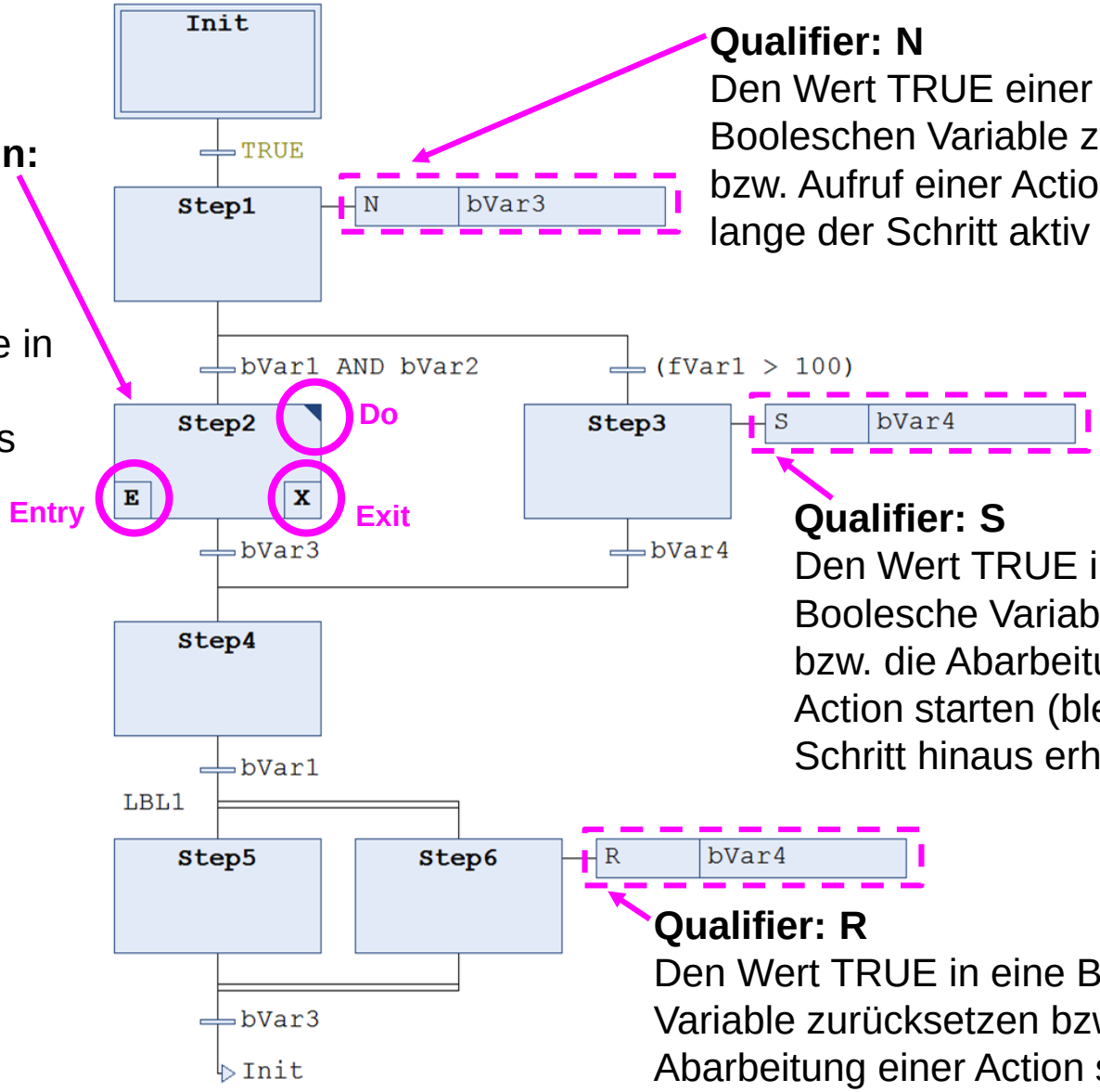


Branch:
Paralleler Aufruf von mehreren Schritten, Aufrufreihenfolge oft von links nach rechts, besitzt Namen zur Identifikation

Sprung:
Sprung (nach Transition) in anderen Schritt oder Branch (Angabe Schritt / Branch Namen)

Beispiel für die Notationselemente in einer Implementierung in Ablaufsprache (AS) (3)

Entry-/Do-/Exit-Action:
Aufruf einer Aktion
Entry: bei Eintritt des Codes in den Schritt
Do: so lange der Code in diesem Schritt ist
Exit: bei Verlassen des Schrittes



Qualifier: N
Den Wert TRUE einer Booleschen Variable zuweisen bzw. Aufruf einer Action, so lange der Schritt aktiv ist

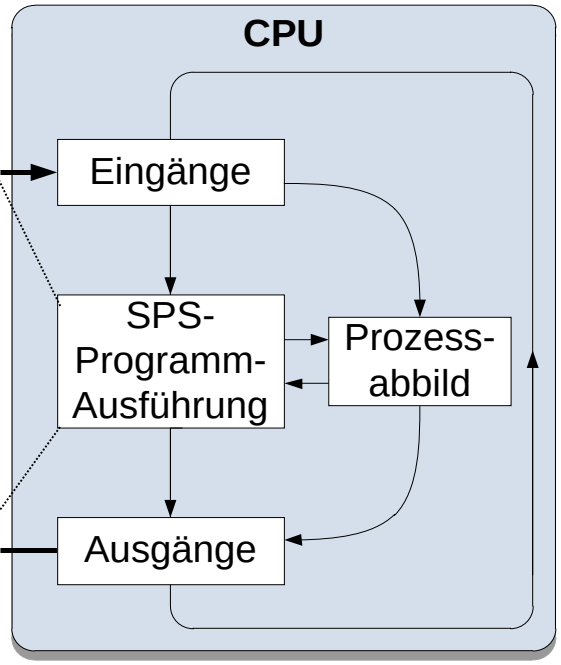
Qualifier: S
Den Wert TRUE in eine Boolesche Variable setzen bzw. die Abarbeitung einer Action starten (bleibt über Schritt hinaus erhalten)

Qualifier: R
Den Wert TRUE in eine Boolesche Variable zurücksetzen bzw. die Abarbeitung einer Action stoppen

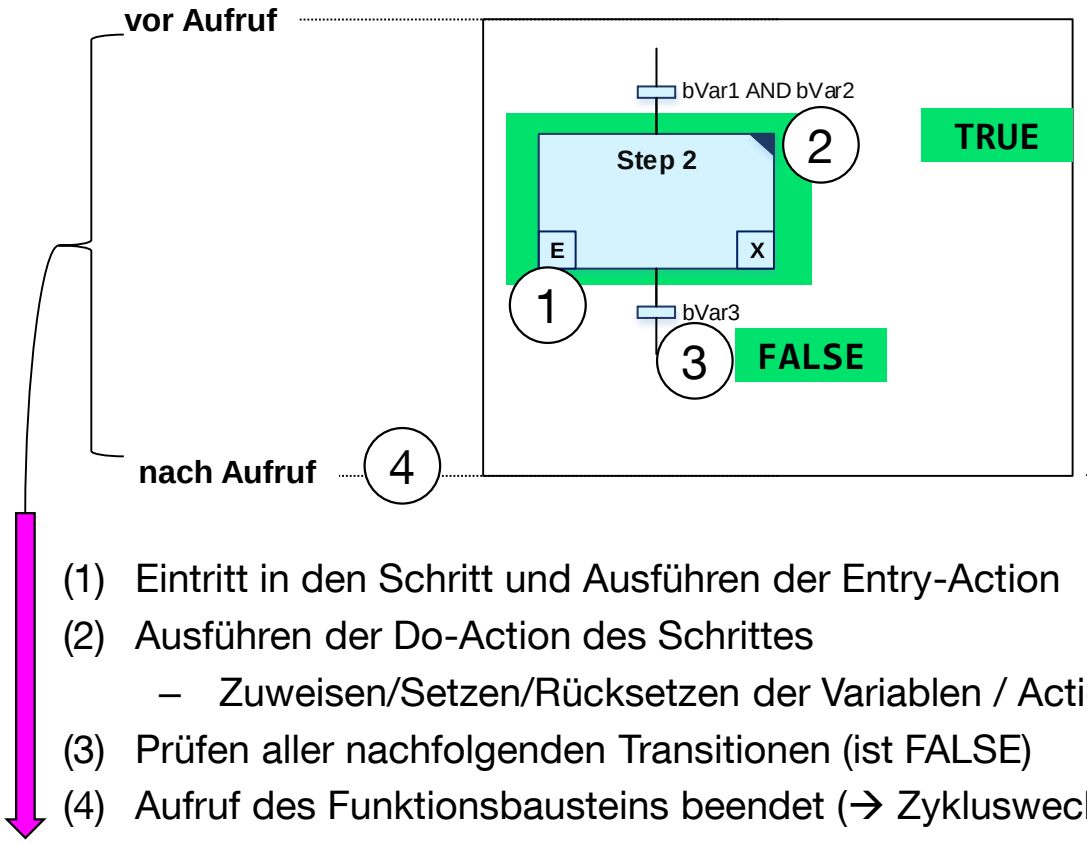
Abarbeitung von IEC 61131-3 Code in der Ablaufsprache (AS) (1)

```
PROGRAM Main
VAR
    fbDemoAS : FB_DemoAS;
END_VAR
```

```
VAR_GLOBAL
    fVal  : REAL;
END_VAR
```



Aktueller SPS-Zyklus: n



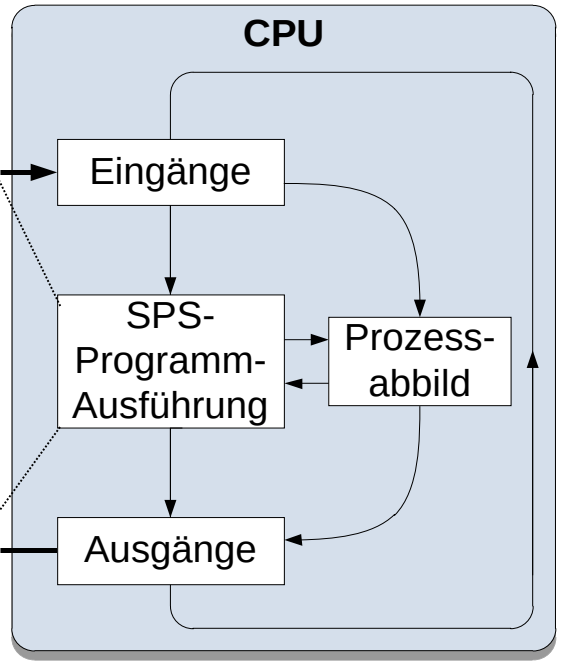
(1) Eintritt in den Schritt und Ausführen der Entry-Action
(2) Ausführen der Do-Action des Schrittes
  – Zuweisen/Setzen/Rücksetzen der Variablen / Actions
(3) Prüfen aller nachfolgenden Transitionen (ist FALSE)
(4) Aufruf des Funktionsbausteins beendet (→ Zykluswechsel)

Aktueller Schritt / Wert

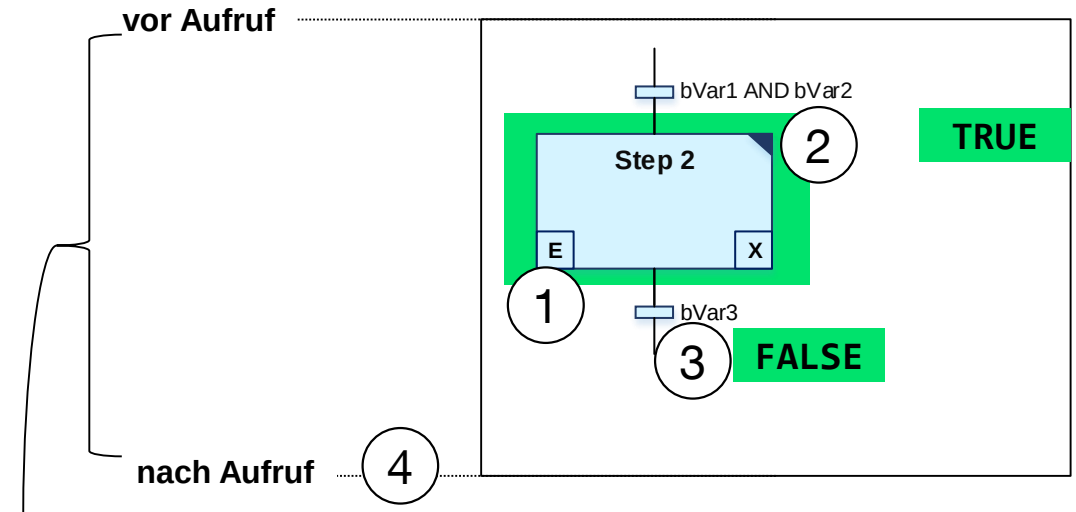
Abarbeitung von IEC 61131-3 Code in der Ablaufsprache (AS) (2)

```
PROGRAM Main
VAR
   fbDemoAS : FB_DemoAS;
END_VAR
```

```
VAR_GLOBAL
   fVal  : REAL;
END_VAR
```



Aktueller SPS-Zyklus: n + 1



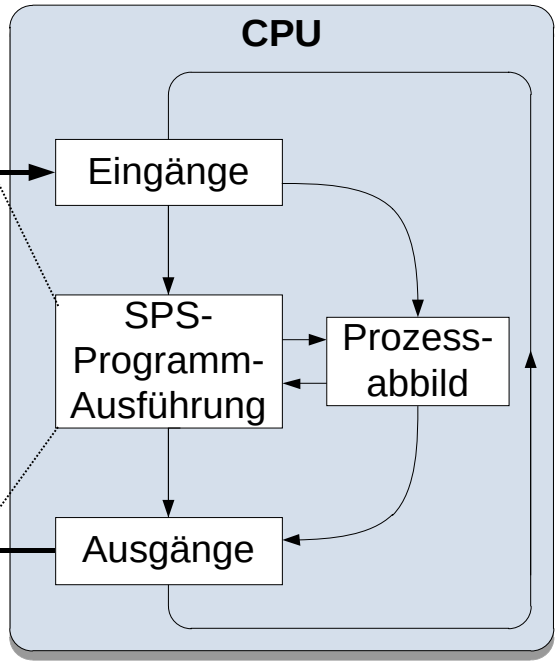
(1) Schritt bereits aktiv → Entry-Action wird nicht ausgeführt
(2) Ausführen der Do-Action des Schrittes
  – Zuweisen/Setzen/Rücksetzen der Variablen / Actions
(3) Prüfen aller nachfolgenden Transitionen (ist FALSE)
(4) Aufruf des Funktionsbausteins beendet (→ Zykluswechsel)

Aktueller Schritt / Wert

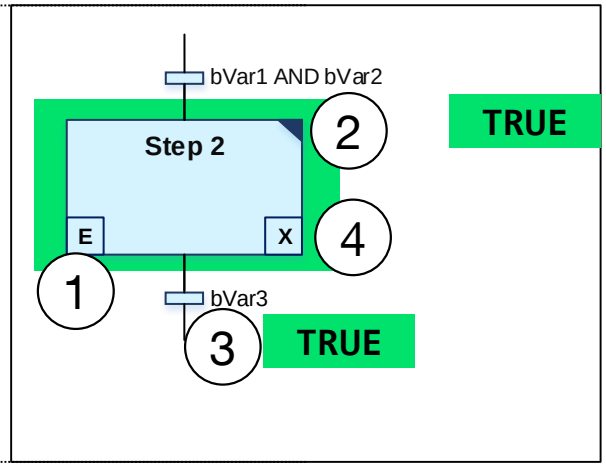
Abarbeitung von IEC 61131-3 Code in der Ablaufsprache (AS) (3)

```
PROGRAM Main
VAR
    fbDemoAS : FB_DemoAS;
END_VAR
```

```
VAR_GLOBAL
    fVal  : REAL;
END_VAR
```



Aktueller SPS-Zyklus: n + 2



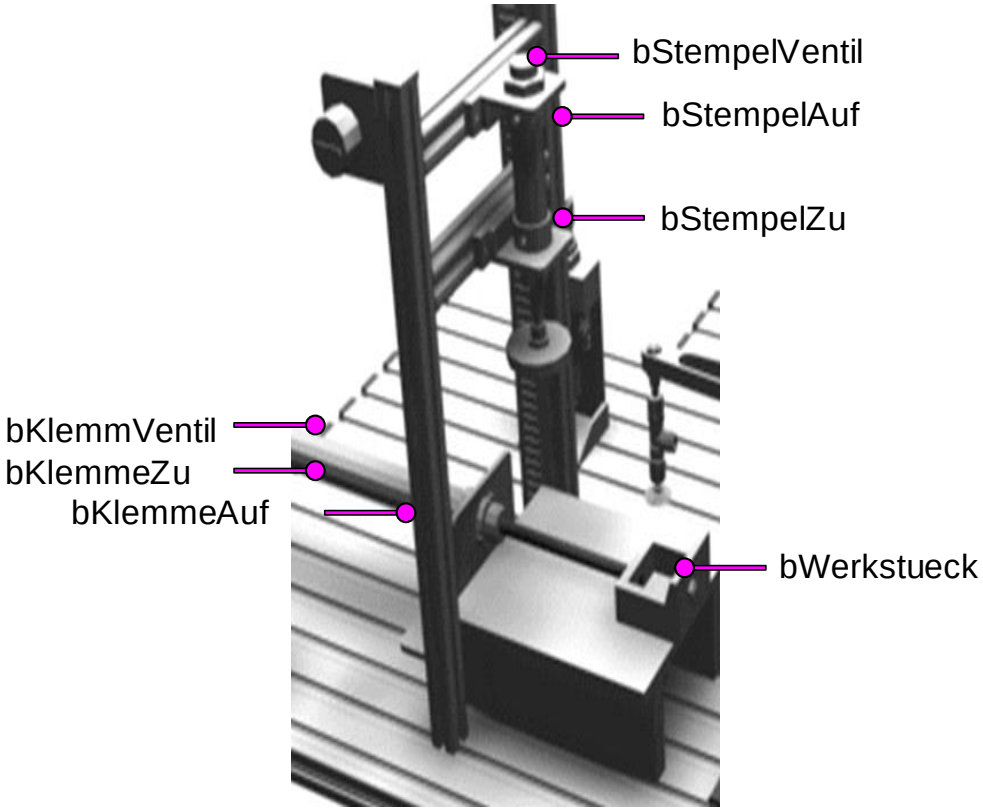
(1) Schritt bereits aktiv → Entry-Action wird nicht ausgeführt
(2) Ausführen der Do-Action des Schrittes
  – Zuweisen/Setzen/Rücksetzen der Variablen / Actions
(3) Prüfen aller nachfolgenden Transitionen (ist TRUE)
(4) Ausführen der Exit-Action des Schrittes
(5) Aufruf des Funktionsbausteins beendet (→ Zykluswechsel)

Aktueller Schritt / Wert

Programmierung eines FBs in Ablaufsprache (AS) – Stempeleinheit der extended Pick & Place Unit (xPPU)

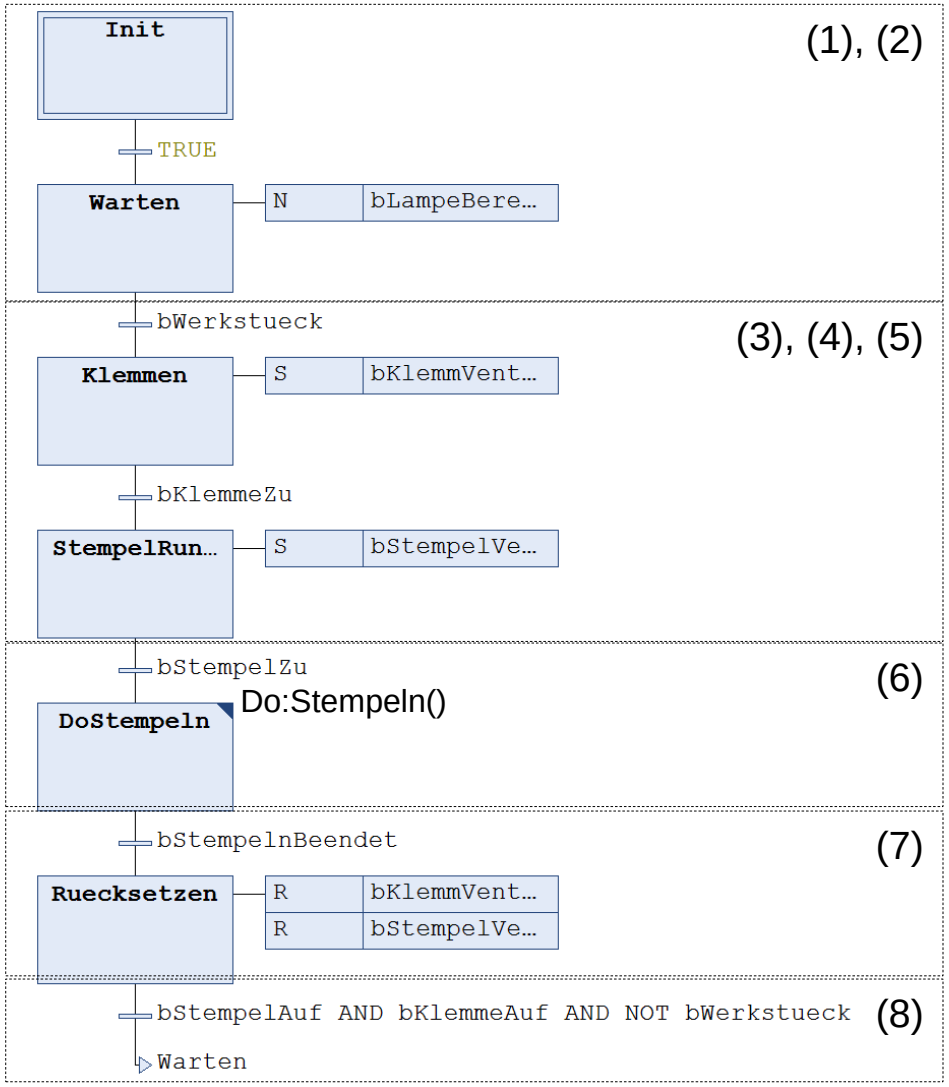
- Für die Stempeleinheit der xPPU soll ein FB in Ablaufsprache (AS) erstellt werden
- Implementierungsteil und Ablaufbeschreibung bereits vorgegeben

```
FUNCTION_BLOCK FB_Stempeleinheit

VAR
      bLampeBereit        AT%Q*   :  BOOL;
      // Zeigt Bereitstatus an
      bKlemmVentil        AT%Q*   :  BOOL;
      // TRUE für Klemmzylinder einfahren
      bStempelVentil      AT%Q*   :  BOOL;
      // TRUE für Stempelzylinder ausfahren

      bKlemmeAuf          AT%I*   :  BOOL;
      bStempelAuf         AT%I*   :  BOOL;
      // TRUE = offene Stellung
      bStempelZu          AT%I*   :  BOOL;
      bKlemmeZu           AT%I*   :  BOOL;
      // TRUE = geschlossene Stellung
      bWerkstueck         AT%I*   :  BOOL;
      // TRUE = Werkstück vorhanden
      bStempelnBeendet            :  BOOL;
      //TRUE = Aktion Stempeln fertig
END_VAR

ACTION Stempeln :
(* Implementierungsteil der Aktion zum
Stempeln von Werkstücken *)
END_ACTION
```



Programmierung eines FBs in Ablaufsprache (AS) – Stempeleinheit der extended Pick & Place Unit (xPPU)

Lösung



Angabe

(1) Nach der Initialisierung soll die Stempeleinheit in einen Wartezustand gehen

(2) Im Wartezustand, soll die Lampe für den Bereitstatus leuchten.

(3) Wird ein Werkstück in der Aufnahme erkannt, soll der Klemmzylinder eingefahren werden

(4) Wenn der Klemmzylinder eingefahren (zu) ist, soll der Stempelzylinder ausgefahren werden

(5) Die Zylinder sollen beide wenn ausgefahren (zu) permanent mit Druck beaufschlagt werden

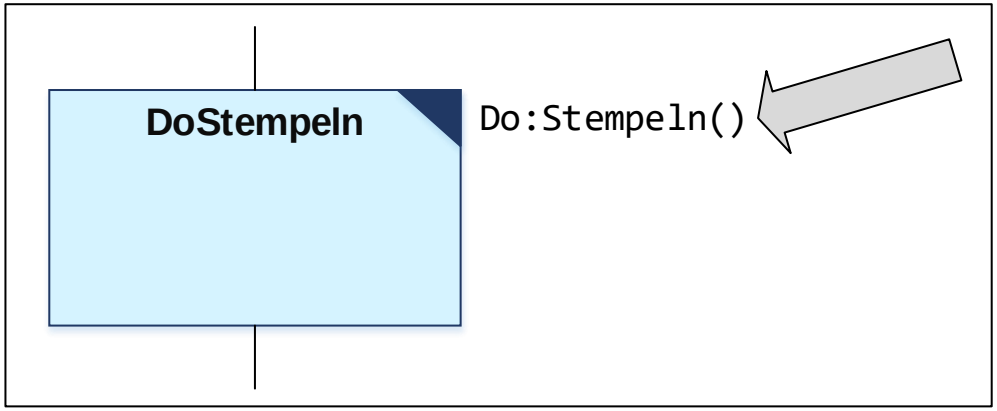
(6) Ist der Stempelzylinder ausgefahren (zu), soll während des nächsten Schritts die Aktion „Stempeln“ ausgeführt werden

(7) Ist die Aktion Stempeln beendet, sollen im nächsten Schritt beide Zylinder wieder ausgefahren werden (auf)

(8) Sind beide Zylinder ausgefahren (auf) und ist kein Werkstück mehr in der Aufnahme, soll der Baustein wieder in den Wartezustand gehen

Einschub: Konvention zur Angabe von Entry-/Exit-/Do-Actions in dieser Lehrveranstaltung

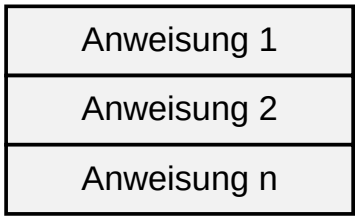
- Welche Aktion als Entry-/Do-/Exit-Aktion aufgerufen werden soll, ist im Diagramm (grafischen Editor) meist nicht direkt sichtbar (oft nur im „Eigenschaften“-Fenster der Programmierumgebung)
- Für diese Lehrveranstaltung wird daher folgende Konvention getroffen: Name der aufzurufenden Aktion mit Schlüsselwort und gefolgt von Klammern „()“ an die entsprechende Ecke des Schrittes schreiben (siehe Beispiel unten: Do-Aktion)



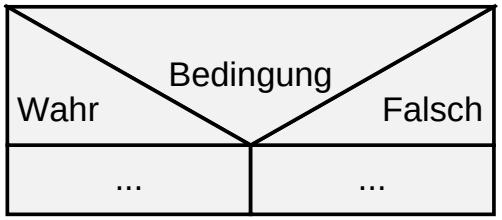
Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detail Engineering (Fokus: AT-Hardware)
- Kapitel 5 Detail Engineering (Fokus: AT-Software)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
    - Programmiersprache Kontaktplan (KOP)
    - Programmiersprache Funktionsbausteinsprache (FBS)
    - Programmiersprache Anweisungsliste (AWL)
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - Programmiersprache Ablaufsprache (AS)
    - **Programmiersprache Strukturierter Text (ST)**

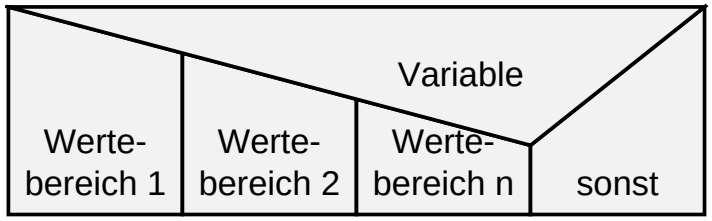
Exkurs: Struktogramm bzw. Nassi-Shneiderman-Diagramm Notationselemente



Folge (Sequenz):
Ausführung mehrerer Anweisungen hintereinander. Abarbeitung von oben nach unten.



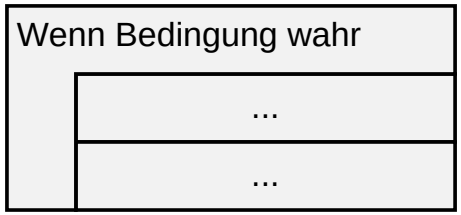
Selektion (einseitige und zweiseitige Auswahl):
Durch die Bedingung bzw. den Ausdruck wird eine Auswahl der auszuführenden Anweisungen vorgenommen. Ist die Bedingung wahr, werden die Anweisungen hinter dem Ausdruck wahr ausgeführt, sonst hinter falsch. Bei einer einseitigen Auswahl steht im falsch-Zweig keine Anweisung.



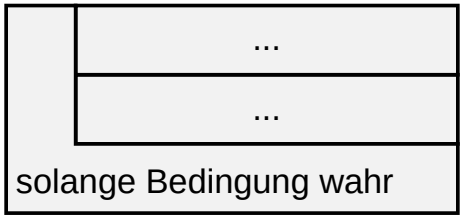
Selektion (Mehrfachauswahl):
Muss zwischen mehr als zwei Möglichkeiten ausgewählt werden, dann wird die Mehrfachauswahl verwendet.

Quelle: Klima, Selberherr, 2007

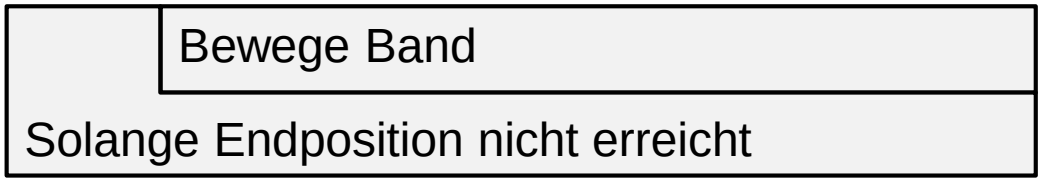
Exkurs: Struktogramm bzw. Nassi-Shneiderman-Diagramm Notationselemente

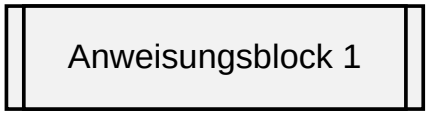


Wiederholung (Schleife), vorprüfend:
Die Abfrage erfolgt vor jedem Schleifendurchlauf.



Wiederholung (Schleife), nachprüfend:
Die Abfrage erfolgt nach jedem Schleifendurchlauf.





Unterprogrammaufruf:
Es wird ein Unterprogramm aufgerufen, welches durch einen weiteren Programmablaufplan beschrieben ist.

Quelle: Klima, Selberherr, 2007

IEC 61131-3 Programmiersprachen – Einführung
Strukturierter Text (ST) / Structured Text (ST)

Bietet gute Strukturierungsmöglichkeiten, geeignet für Abläufe und Verknüpfungen, vergleichbar mit der Programmiersprache Pascal/C/C++

```
if (a && !b && (c || d))
      e = true;
else
      e = false;
```

```
//Vereinfacht
e = a && !b && (c || d);
```

```
IF a AND NOT b AND (c OR d) THEN
      e := TRUE;
ELSE
      e := FALSE;
END_IF
```

```
//Vereinfacht
   e := a AND NOT b AND (c OR d);
```

- Schleifen, erweiterte Kontrollstrukturen

```
FOR i := 0 TO 5 DO
      ( *…* )
END_FOR
```

```
CASE a OF
      1:    x := 25;
      2:    x := 50;
END_CASE
```

- Erweiterte mathematische Funktionen

```
c := SQRT (a**2 + b**2)
```

- Funktions- / FB-Aufrufe möglich

```
FB_Pythagoras(h1 := a, h2 := b1);
c := FB_Pythagoras.Erg;
```

Operatoren in ST

Operator	Beispiel	Wert	Beschreibung	Priorität
( )	(2+3) * (4+5)	45	Klammerung	Höchste
**	3**4	81	Potenzierung	
-	-10	-10	Negation	
NOT		NOT TRUE	Falsch / Komplement	
*	10 * 3	30	Multiplikation	
/	6 / 2	3	Division	
MOD	17 MOD 10	7	Modulo	
+	2 + 3	5	Addition	
-	4 - 2	2	Subtraktion	
<, >, <=, >=	4 > 12	FALSE	Vergleich	
=	T#26h = T#1d2h	TRUE	Gleichheit	
<>	8 <> 16	TRUE	Ungleichheit	
&, AND	TRUE & FALSE	FALSE	Boolesches UND	
XOR	TRUE XOR FALSE	TRUE	Boolesches XOR	
OR	TRUE OR FALSE	TRUE	Boolesches ODER	Niedrigste

Anweisungen in ST

Schlüsselwort	Beispiel
RETURN	`RETURN;`

IF	`IF a < b THEN`
	`  c := 1;`
	`ELSIF a = b THEN`
	`  c := 2;`
	`ELSE`
	`  c := 3;`
	`END_IF`

CASE	`CASE f OF`
	`  1:  a := 3;`
	`  2:  a := 4;`
	`ELSE  a := 0;`
	`END_CASE`

FOR	`FOR a := 1 TO 10 BY 2`
	`DO`
	`  f[a] := b;`
	`END_FOR`

WHILE	`WHILE b > 1 DO`
	`  b := b / 2;`
	`END_WHILE`

REPEAT	`REPEAT`
	`  a := a * b;`
	`UNTIL  a > 10000;`
	`END_REPEAT`

Exit	`EXIT;`

Beispielaufgabe: Erstellung und Verwendung einer Funktion (FC) in der Programmiersprache ST – Angabe

(1) Die **Funktion** soll aus der **Eingabe von Sensorwerten** (induktiver Sensor, optischer Sensor) den **Typ eines Werkstücks** bestimmen und als Rückgabewert liefern
- induktiv = TRUE → metallisch,
- induktiv = FALSE & optisch = TRUE → weiß,
- induktiv = FALSE & optisch = FALSE → schwarz,

(2) Im Programm **Main** soll die **Funktion** anschließend mit den im Programm vorhandenen Sensorwerten als Parameterwerte aufgerufen werden und das Ergebnis in der entsprechenden Variable gespeichert werden

```
TYPE E_WsType :
(
  metall      := 0,
  weis        := 1,
  schwarz     := 2
);

END_TYPE
```

Funktion zur Bestimmung Typ des Werkstücks

(1)

```
FUNCTION FC_WsType : E_WsType

VAR_IN
      bSensInd      : BOOL;
      bSensOpt      : BOOL;
END_VAR
// (* Implementierungsteil *)









END_FUNCTION
```

Programm Main (Erkannter Werkstücktypen Variable zuweisen)

(2)

```
PROGRAM Main

VAR
      bInduktiv      AT%I*      : BOOL;
      bOptisch       AT%I*      : BOOL;
      eWsType                   : E_WsType;
END_VAR
// (* Implementierungsteil *)









END_PROGRAM
```

Beispielaufgabe: Erstellung und Verwendung einer Funktion (FC) in der Programmiersprache ST – Lösung

(1) Die **Funktion** soll aus der **Eingabe von Sensorwerten** (induktiver Sensor, optischer Sensor) den **Typ eines Werkstücks** bestimmen und als Rückgabewert liefern
- induktiv = TRUE → metallisch,
- induktiv = FALSE & optisch = TRUE → weiß,
- induktiv = FALSE & optisch = FALSE → schwarz,

(2) Im Programm **Main** soll die **Funktion** anschließend mit den im Programm vorhandenen Sensorwerten als Parameterwerte aufgerufen werden und das Ergebnis in der entsprechenden Variable gespeichert werden

```
TYPE E_WsType :
(
  metall      := 0,
  weis        := 1,
  schwarz     := 2
);

END_TYPE
```

Funktion zur Bestimmung Typ des Werkstücks

(1)

```
FUNCTION FC_WsType : E_WsType

VAR_IN
      bSensInd      : BOOL;
      bSensOpt      : BOOL;
END_VAR

IF bSensInd THEN
      FC_WsType := E_WsType.metall;
ELSE
      IF bSensOpt THEN
            FC_WsType := E_WsType.weis;
      ELSE
            FC_WsType := E_WsType.schwarz;
END_IF

END_FUNCTION
```

Programm Main (Erkannter Werkstücktypen Variable zuweisen)

(2)

```
PROGRAM Main

VAR
      bInduktiv       AT%I*       : BOOL;
      bOptisch        AT%I*       : BOOL;
      eWsType                     : E_WsType;
END_VAR

eWsType := FC_WsType(
                  bSensInd := bInduktiv,
                  bSensOpt := bOptisch,
                  );

END_PROGRAM
```

Hinweis: *Die Anweisung nach ELSE wird nur ausgeführt, wenn die IF-Bedingung nicht wahr (nicht TRUE) ist.*

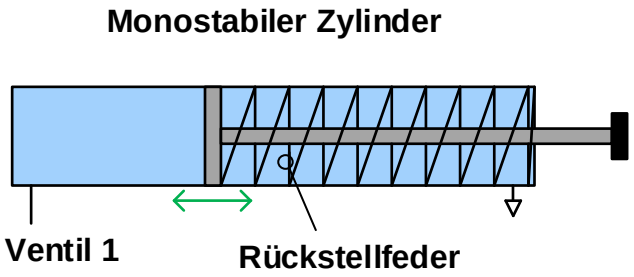
Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detailplanung von AT-Systemen (Hardware-Auslegung)
- Kapitel 5 Detailplanung von AT-Systemen (Software-Auslegung)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - Programmiersprache Ablaufsprache (AS)
    - Programmiersprache Strukturierter Text (ST)
      - **Funktionsbausteine und Methoden**
      - Verwendung von Interfaces
      - Vererbung in der IEC 61131-3
      - Properties von Funktionsbausteinen

Einschub – mono- und bistabile Zylinder

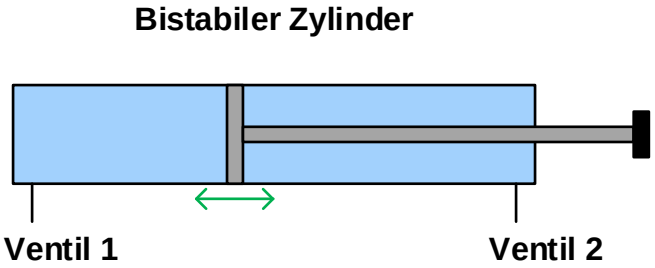
Monostabiler Zylinder

- Verfügt über einen Ventileingang
- Feder versetzt Zylinder automatisch wieder in Ausgangsposition
- Für anhaltenden Ausfahrzustand muss Ventil1 durchgehend mit Druck beaufschlagt werden

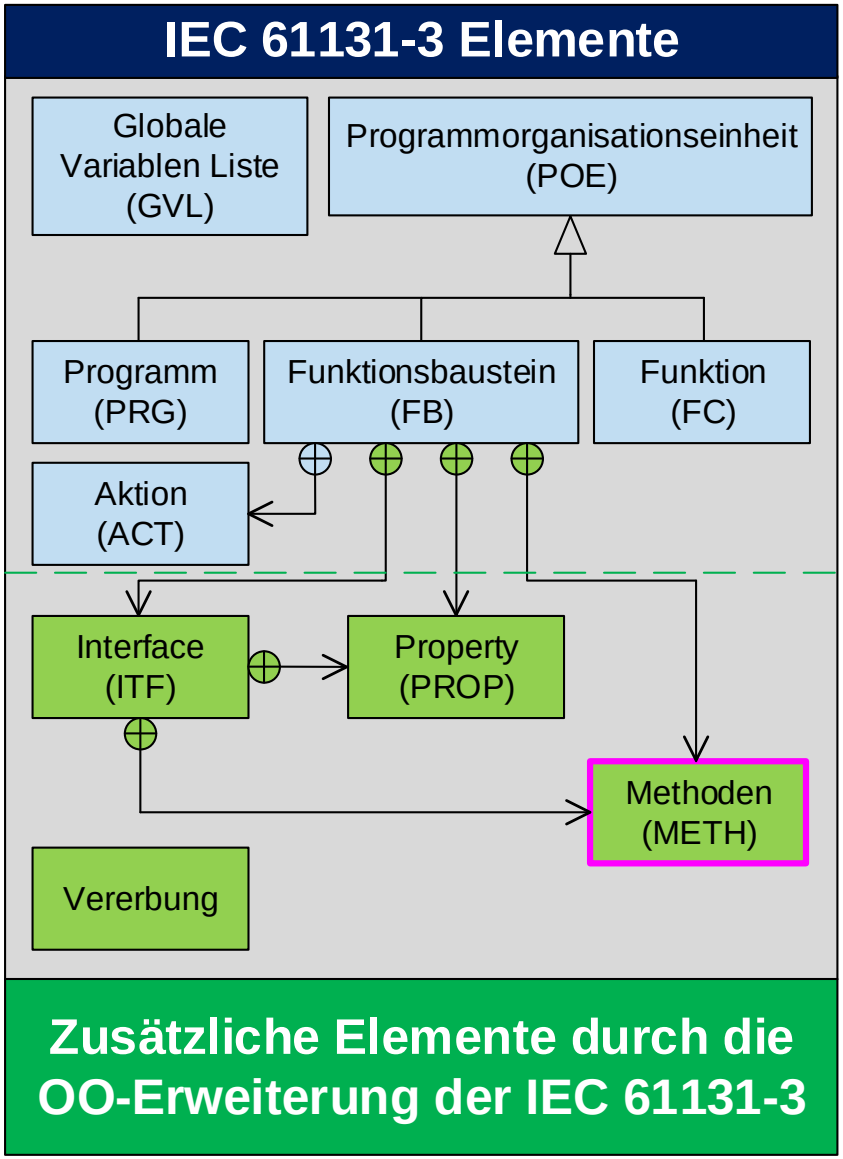


Bistabiler Zylinder

- Verfügt über zwei Ventileingänge
- Ansteuern von Ventil 2 versetzt Zylinder wieder in Ausgangsposition (Ventil 1 darf dabei nicht gleichzeitig angesteuert werden)
- Für anhaltenden Ausfahrzustand muss Ventil1 nicht durchgehend mit Druck beaufschlagt werden



Zusätzliche Elemente in der OO IEC 61131-3 – Methode (METH)



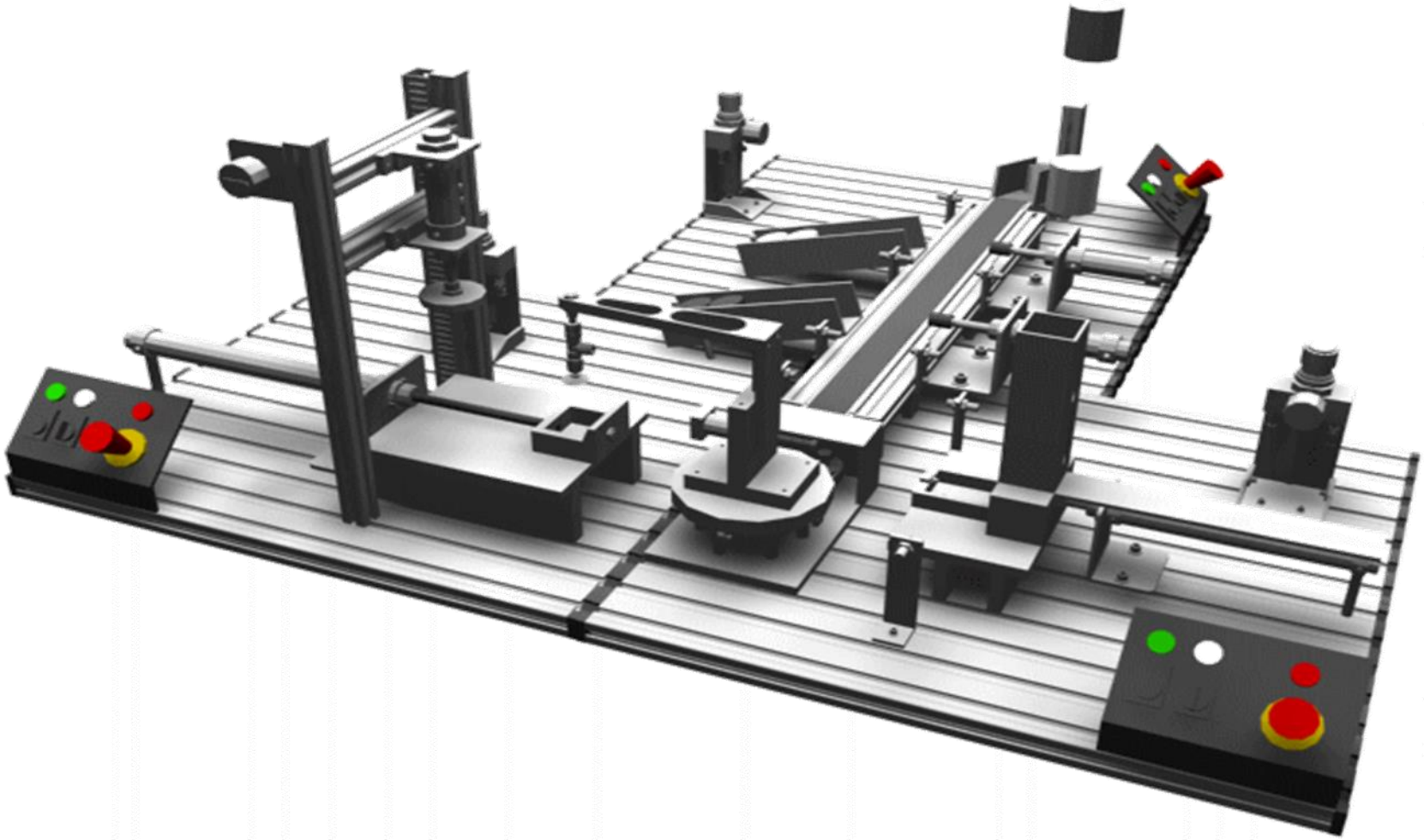
- Spezielle Code-Abschnitte eines FB zur Implementierung von Einzel-Funktionalitäten
- Ein FB kann beliebig viele Methoden haben
- Ein FB mit Methoden muss nicht zwingend einen eigenen Implementierungsteil haben
- Eigener Deklarationsteil zusätzlich zu dem des Funktionsbausteins
- Definierter Rückgabewert beim Aufruf (kann optional verwendet werden)
- Deklaration zwischen Deklarationsteil und Implementierungsteil eines FB (wie Aktion)

Beispiel für Deklaration einer Methode

```
METHOD METH_Ausfahren : BOOL

VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
// Implementierungsteil der Methode
END_METHOD
```

Nutzung von Methoden in IEC 61131-3 Steuerungscode – Ideensammlung an der PPU (1)

- Kapselung von mehreren verschiedenen (Einzel-)Funktionalitäten, die auf gleiche Hardware und Variablenraum (Deklarationsteil) zugreifen, innerhalb eines Funktionsbausteins mit Methoden

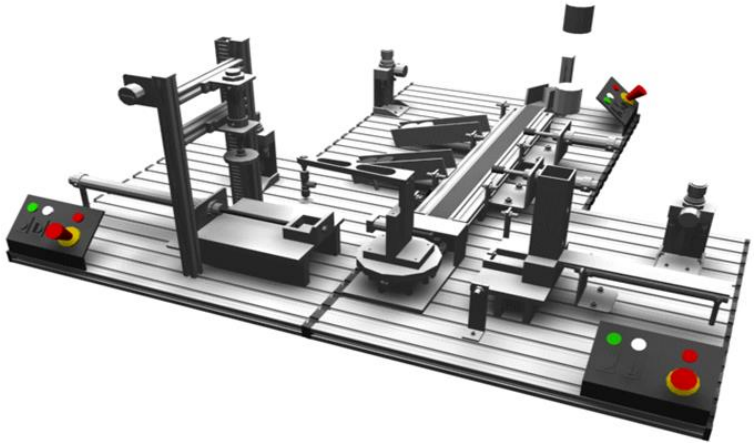


Nutzung von Methoden in IEC 61131-3 Steuerungscode – Ideensammlung an der PPU (1)

- Kapselung von mehreren verschiedenen (Einzel-)Funktionalitäten, die auf gleiche Hardware und Variablenraum (Deklarationsteil) zugreifen, innerhalb eines Funktionsbausteins mit Methoden
- Zusätzlich eigener Deklarationsteil → Parametrierung der umgesetzten Funktionalität

Stempel:

Sortierband:



Kran:

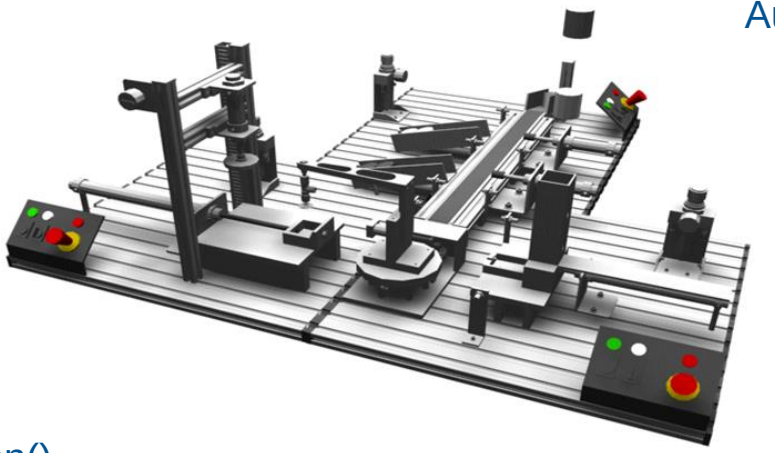
Stapel:

Nutzung von Methoden in IEC 61131-3 Steuerungscode – Ideensammlung an der PPU (2) – Lösungsvorschlag

- Kapselung von mehreren verschiedenen (Einzel-)Funktionalitäten, die auf gleiche Hardware und Variablenraum (Deklarationsteil) zugreifen, innerhalb eines Funktionsbausteins mit Methoden
- Zusätzlich eigener Deklarationsteil → Parametrierung der umgesetzten Funktionalität

Stempel:
Einspannen(),
Stempeln() → Parameter: Druck

Sortierband:
Aussortieren() → Parameter Ziel



Kran:
Heben(), Senken(),
Ansaugen(), Ablassen(),
Drehen() → Parameter: Ziel

Stapel:
Ausschieben(),
Rücksetzen()

IEC 61131-3 Methoden – Beispiel Pneumatikzylinder
1) Kontinuierlicher Methodenaufruf

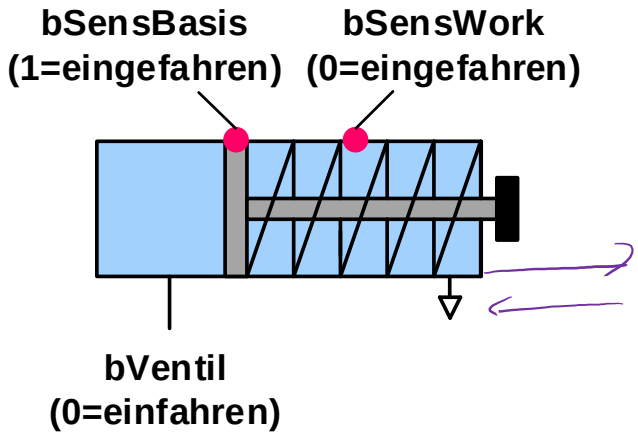
Im Beispiel hier:

- Zylinder wird über boolesche Variable bVentil angesteuert
- Zwei Sensoren erfassen die Endlagenpositionen (bSensWork ≙ Ausgefahren / bSensBasis ≙ Eingefahren)
- Zylinder stellt die Funktionalitäten (Einfahren/Ausfahren) zur Verfügung
- Definierte Rückgabewerte als Status des Zylinders bei Methodenaufruf: Ready (Bereit), Busy (Beschäftigt), Done (Beendet)

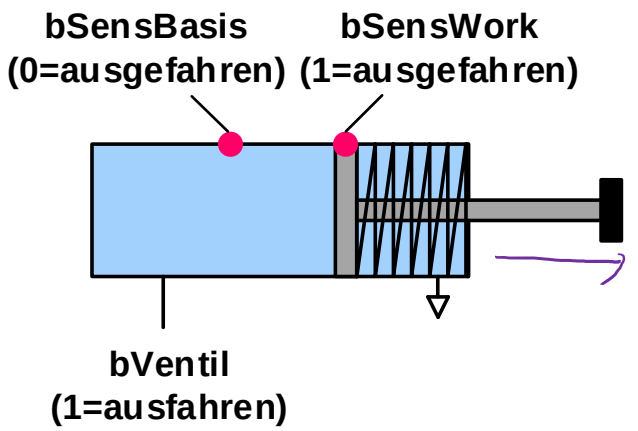
Aufgabenstellung:

- Für den Pneumatikzylinder soll zunächst ein FB erstellt und Methoden für das Ein- u. Ausfahren angelegt werden
- Methode für Ausfahren soll vollständig implementiert werden
- Durch Drücken eines Tasters Methode über ein Programm aufgerufen werden und Zylinder soll ausfahren. Dabei soll stets der aktuelle Status rückgemeldet werden

Zylinder eingefahren



Zylinder ausgefahren



IEC 61131-3 Methoden – Beispiel Pneumatikzylinder
Schritt 1: Deklaration des Funktionsbausteins

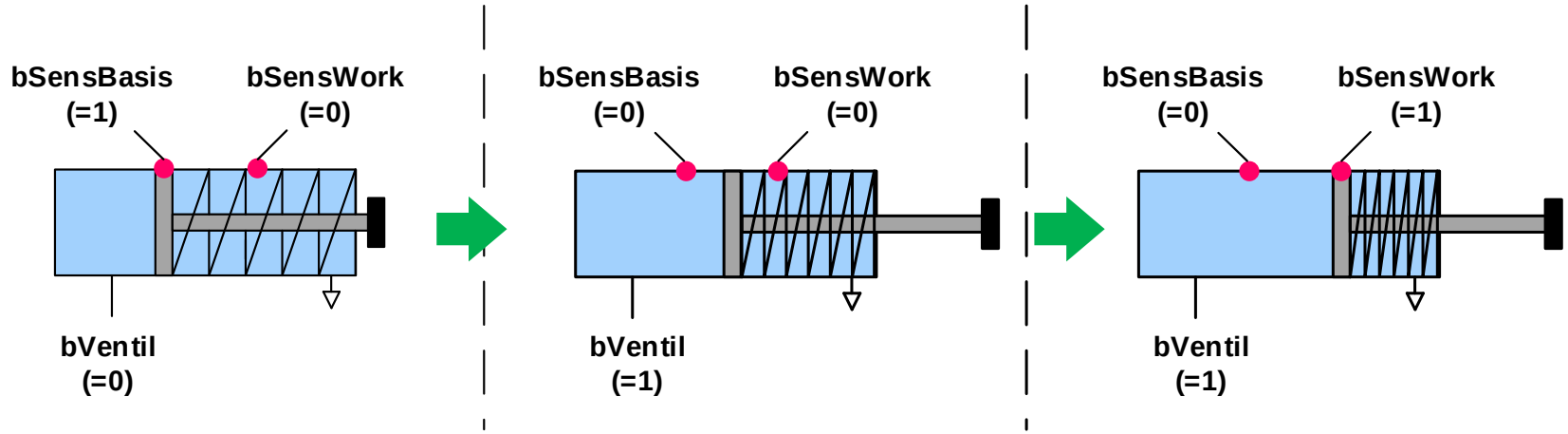
```
FUNCTION_BLOCK  FB_Zylinder1V2S
VAR
      bVentil      AT%Q*  : BOOL;
      bSensBasis   AT%I*  : BOOL;
      bSensWork    AT%I*  : BOOL;
END_VAR
METHOD  METH_Ausfahren : E_Status
// Deklaration und Implementierung
END_METHOD
METHOD  METH_Einfahren : E_Status
// Deklaration und Implementierung
END_METHOD
END_FUNCTION_BLOCK
```

Gemeinsamer Variablenraum (Zylinder)

```
TYPE E_Status :
(Ready:=0, Busy:=1, Done:=2);
END_TYPE
```

Eigener Datentyp für Rückgabewert der Methoden

Ausfahrvorgang:



Rückgabe: Ready | Rückgabe: Busy | Rückgabe: Done

IEC 61131-3 Methoden – Beispiel Pneumatikzylinder
Schritt 1: Deklaration des Funktionsbausteins

```
FUNCTION_BLOCK  FB_Zylinder1V2S
VAR
      bVentil      AT%Q*  : BOOL;
      bSensBasis   AT%I*  : BOOL;
      bSensWork    AT%I*  : BOOL;
END_VAR
METHOD  METH_Ausfahren : E_Status
// Deklaration und Implementierung
END_METHOD
METHOD  METH_Einfahren : E_Status
// Deklaration und Implementierung
END_METHOD
END_FUNCTION_BLOCK
```

Gemeinsamer Variablenraum (Zylinder)

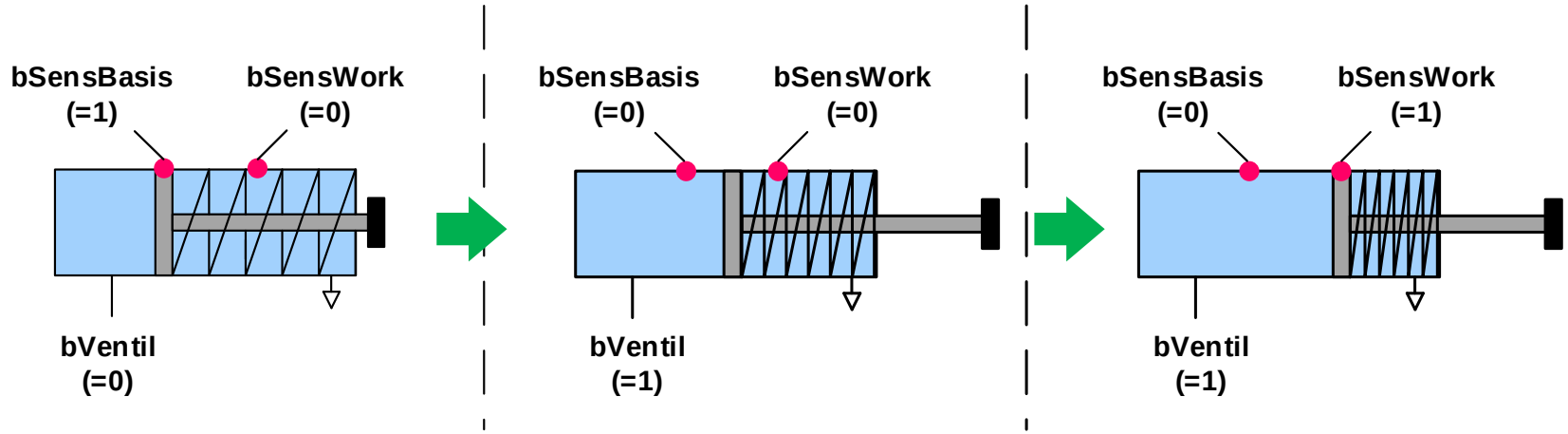
```
TYPE E_Status :
(Ready:=0, Busy:=1, Done:=2);
END_TYPE
```

Eigener Datentyp für Rückgabewert der Methoden

Mögliche Fehlerzustände beim Ausfahren:

- Der Zylinder befindet sich aktuell nicht in Basisposition (NOT bSensBasis)
- Der Zylinder wird aufgefordert einzufahren bevor das Ausfahren abgeschlossen ist
- Ein weiterer Fehler tritt auf (z.B. Zylinder klemmt)

Ausfahrvorgang:



Rückgabe: Ready | Rückgabe: Busy | Rückgabe: Done

IEC 61131-3 Methoden – Beispiel Pneumatikzylinder
Schritt 2: Deklaration und Implementierung der Methoden

```
FUNCTION_BLOCK  FB_Zylinder1V2S
VAR
      bVentil      AT%Q*  : BOOL;
      bSensBasis   AT%I*  : BOOL;
      bSensWork    AT%I*  : BOOL;
END_VAR
METHOD  METH_Ausfahren : E_Status
// Deklaration und Implementierung
END_METHOD
METHOD  METH_Einfahren : E_Status
// Deklaration und Implementierung
END_METHOD
END_FUNCTION_BLOCK
```

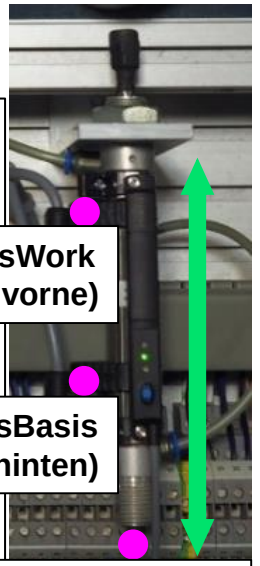
```
TYPE E_Status :
(Ready:=0, Busy:=1, Done:=2);
END_TYPE
```

Eigener Datentyp für Rückgabewert der Methoden

Gemeinsamer Variablenraum (Zylinder)

Lösung:

```
METHOD METH_Ausfahren : E_Status
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
METH_Ausfahren := E_Status.Ready;

bVentil := TRUE;

IF bSensWork THEN
      METH_Ausfahren := E_Status.Done;
ELSIF NOT (bSensBasis OR bSensWork) THEN
      METH_Ausfahren := E_Status.Busy;
END_IF

END_METHOD
```



Angabe für Implementierung:

- Die Methode soll dem Ausgang für das Ventil den Wert „TRUE“ zuweisen
- Für den Rückgabewert soll folgender Zusammenhang implementiert werden:
  - Zylinder hinten → Rückgabe: ‚Ready‘
  - Zylinder vorne → Rückgabe: ‚Done‘
  - undefiniert → Rückgabe ‚Busy‘

Hinweis:
Hier keine Behandlung von Fehlerzuständen

IEC 61131-3 Methoden – Beispiel Pneumatikzylinder
Schritt 3: Ansteuerung des Zylinders über Methoden

```
FUNCTION_BLOCK  FB_Zylinder1V2S
VAR
      bVentil      AT%Q*  : BOOL;
      bSensBasis   AT%I*  : BOOL;
      bSensWork    AT%I*  : BOOL;
END_VAR
METHOD  METH_Ausfahren : E_Status
// Deklaration und Implementierung
END_METHOD
METHOD  METH_Einfahren : E_Status
// Deklaration und Implementierung
END_METHOD
END_FUNCTION_BLOCK
```

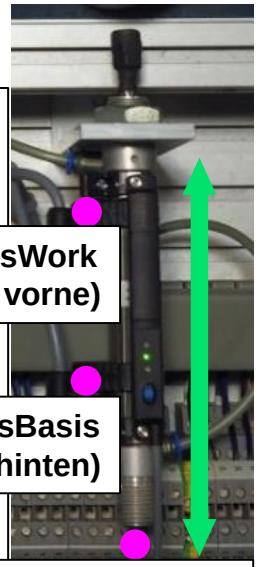
```
TYPE E_Status :
(Ready:=0, Busy:=1, Done:=2);
END_TYPE
```

Eigener Datentyp für Rückgabewert der Methoden

Lösung:

```
PROGRAM Main
VAR
      fbZyl              :  FB_Zylinder1V2S;
      bTaster  AT%IX0.0  :  BOOL;
      eStatus            :  E_Status;
END_VAR

IF bTaster THEN
eStatus := fbZyl.METH_Ausfahren();
END_IF

END_PROGRAM
```

bTaster



bSensWork (1 = vorne)

bSensBasis (1 = hinten)

bVentil (1 = ausfahren

Angabe für Implementierung:

- Wird der Taster gedrückt, soll die Methode ‚METH_Ausfahren' der Instanz ‚fbZyl' aufgerufen werden
- Der Rückgabewert soll in die Variable ‚eStatus' geschrieben werden

Notation Methodenaufruf in ST:

Auslesen des Rückgabewerts

Name der FB-Instanz' → ‚Punktnotation' → ‚Name der Methode'

IEC 61131-3 Methoden – Beispiel Kranmodul der PPU
Erstellung und Aufruf von Funktionsbaustein und Methoden

```
FUNCTION_BLOCK FB_Kran
VAR
      bDreheLinks  AT%Q*   :  BOOL;
      bDreheRechts AT%Q*   :  BOOL;
      iSensPos     AT%I*   :  INT;
END_VAR
METHOD  METH_FahreZu : E_Status
// Deklaration und Implementierung
END_METHOD
END_FUNCTION_BLOCK
```

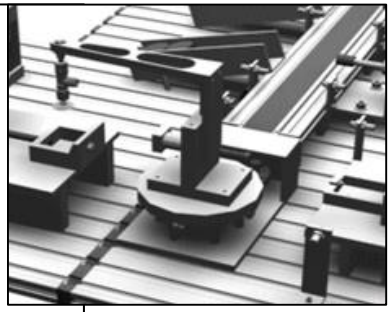
```
TYPE E_Status :
(Ready:=0, Busy:=1, Done:=2);
END_TYPE
```

Eigener Datentyp für Rückgabewert der Methoden

1

Lösung (Methode):

```
METHOD METH_FahreZu : E_Status
VAR_INPUT
      iZielPos : INT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
// Implementierung der Funktionalität
END_METHOD
```



4

Angabe für Implementierung (Methode):

- Die Methode soll bei Aufruf über eine Eingangsvariable parametriert werden können, um anzugeben welches Ziel angefahren werden soll

Angabe für Implementierung (Programm):

- Die Methode des FBs soll aufgerufen werden und der Wert ‚90' an die Eingabevariable übergeben werden
- Der Rückgabewert der Methode soll nicht verwendet werden

Vorgehen bei der Programmierung

2

3

Lösung (Programm):

```
PROGRAM Main
VAR
      fbKran : FB_Kran;
END_VAR
fbKran.METH_FahreZu(iZielPos := 90);
END_PROGRAM
```

Aufrufreihenfolge im IEC 61131-3 Code

Parametrierung der Methode über Eingangsvariable

Einschub – Entprellen von Schaltern und Tastern

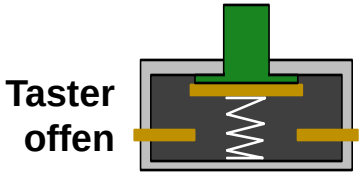
Prellen:

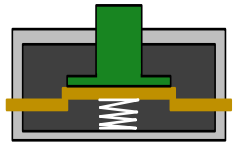
- Mechanisch ausgelöster Störeffekt
- Betätigung des Schalters ruft mehrfaches Öffnen und Schließen des Kontakts hervor
- Ursache meist elastisches Zurückprallen der Schalterfederung

Auswirkungen:

- Auftreten unerwünschter Mehrfachereignisse

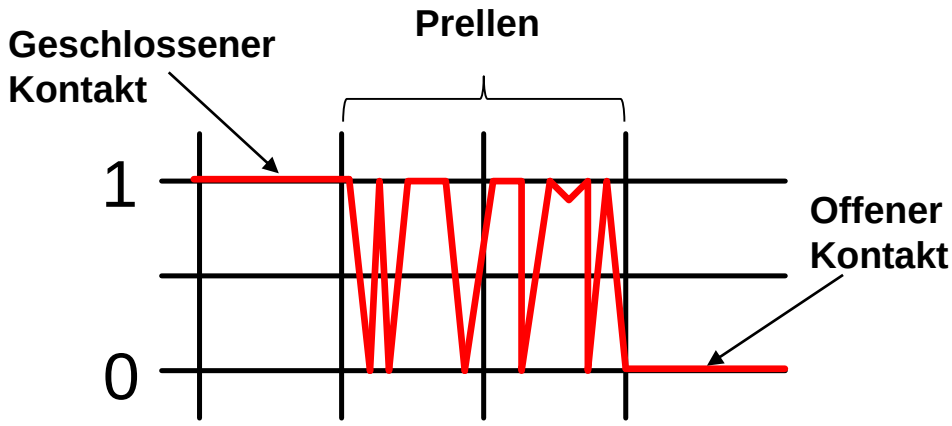
Bsp.: Tastatur (Ohne Entprellung würde Tastenanschlag fälschlicherweise als mehrfacher Anschlag registriert werden)

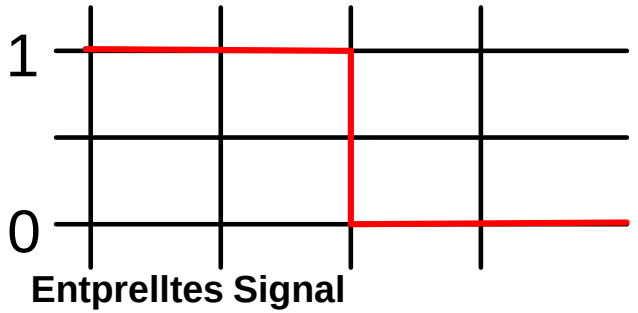




Beispiel: Gegenmaßnahme Entprellen

- Verwendung eines RC-Gliedes als Tiefpassfilter (→Unterdrückung von hochfrequenten Signalanteilen zufolge Kontaktprellen)
- Zustandsänderung wird erst registriert, wenn eine bestimmte Zeit (Entprellzeit) vorliegt. Umsetzung bspw. unter Verwendung einer Eingangsverzögerung (TON)
- Entprellung unter Verwendung einer Flankenerkennung (R_TRIG, F_TRIG)

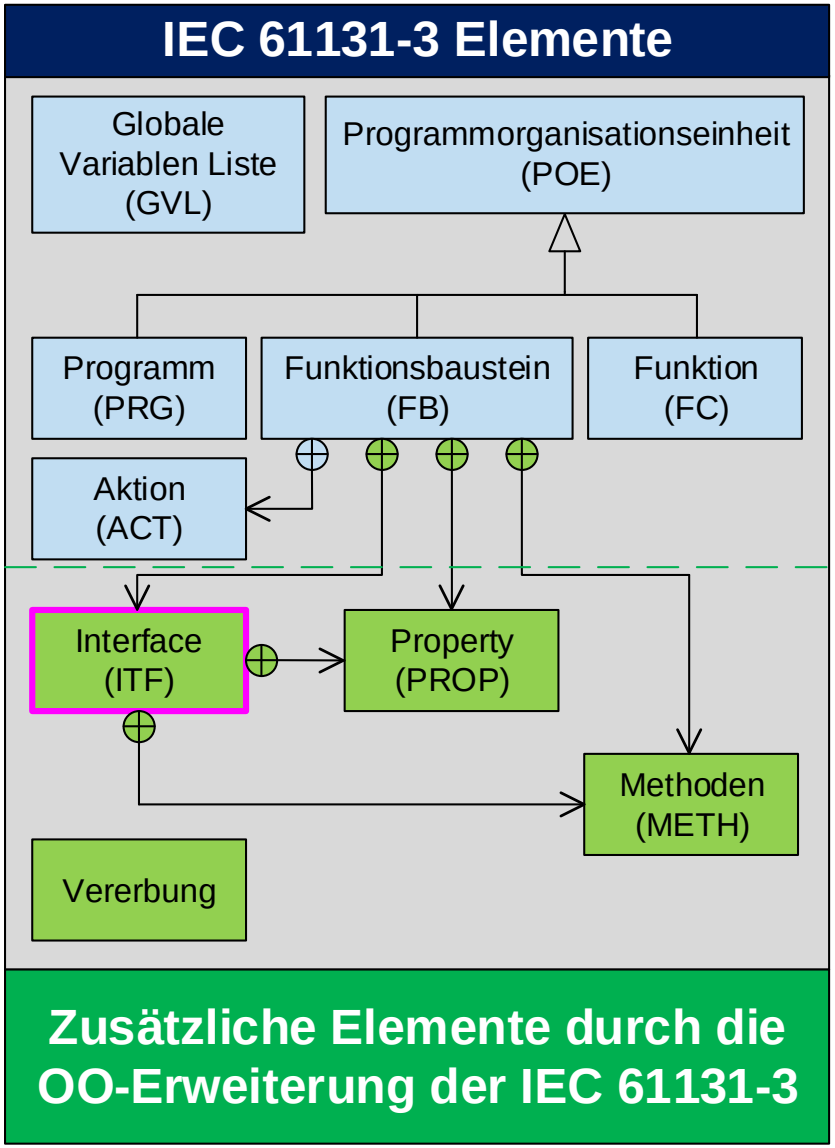




Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detailplanung von AT-Systemen (Hardware-Auslegung)
- Kapitel 5 Detailplanung von AT-Systemen (Software-Auslegung)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - Programmiersprache Ablaufsprache (AS)
    - Programmiersprache Strukturierter Text (ST)
      - Funktionsbausteine und Methoden
      - **Verwendung von Interfaces**
      - Vererbung in der IEC 61131-3
      - Properties von Funktionsbausteinen

Zusätzliche Elemente in der OO IEC 61131-3 – Interface (ITF)



- Verbesserung der Zusammenarbeit mehrerer Entwickler in einem IEC 61131-3 Projekt
  - Möglichkeit zur Spezifikation einer definierten Schnittstelle von FBs
- Deklariert Prototypen für Methoden und Properties, die ein FB haben muss
- Nur Deklarationsteile; Implementierungsteile werden spezifisch für jeden FB separat erstellt
- Implementierung eines Interfaces durch einen FB über Schlüsselwort „*Implements*“

```
INTERFACE ITF_1
METHOD METH_Beispiel : BOOL;
VAR_INPUT
END_VAR
VAR_OUPUT
END_VAR
VAR
END_VAR
//Keine Implementierung der Funktionalität!
END_METHOD
END_INTERFACE
```

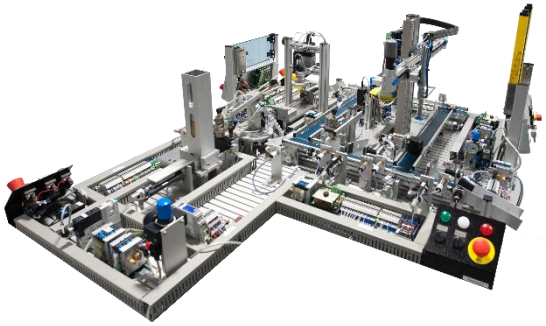
Beispiel:
Deklaration (links) und Nutzung eines Interfaces durch einen FB (unten, mehrere Interfaces durch Komma getrennt)

```
FUNCTION_BLOCK FB_Beispiel IMPLEMENTS ITF_1, ITF_2
//...
```

Interfaces bei gemeinsamer Codeerstellung – Beispiel: Einfach-/Doppelt-Wirkender Zylinder

Hauptprogramm

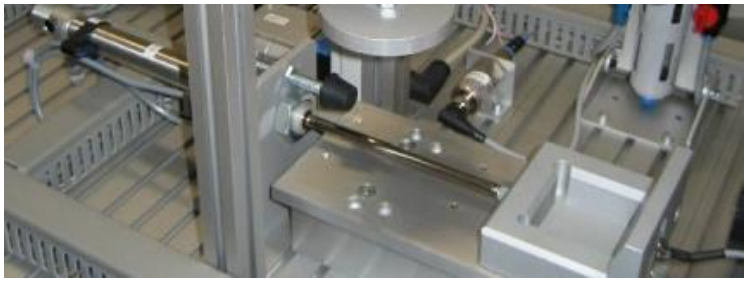
```
PROGRAM Main
VAR
      fbMyZyl1 : FB_Zyl1V2S;
      fbMyZyl2 : FB_Zyl2V2S;
END_VAR
fbMyZyl1.METH_Ausfahren();
fbMyZyl2.METH_Ausfahren();
END_PROGRAM
```



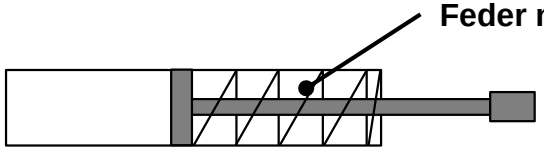
```
INTERFACE ITF_Zyl
METHOD METH_Ausfahren : E_Status
//Keine Implementierung der Funktionalität!
END_METHOD
END_INTERFACE
```

Einfach wirkender Zylinder
- Ausfahren

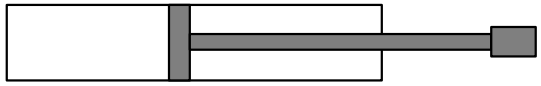
Doppelt wirkender Zylinder
- Ausfahren



Feder mit Rückhub



Ventil1 → TRUE

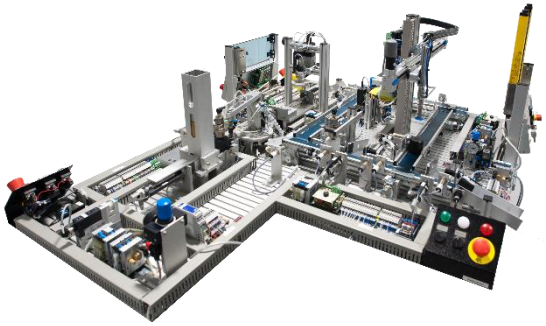


Ventil1 → TRUE **Ventil2 → FALSE**

Interfaces bei gemeinsamer Codeerstellung – Beispiel: Einfach-/Doppelt-Wirkender Zylinder

Hauptprogramm

```
PROGRAM Main
VAR
      fbMyZyl1 : FB_Zyl1V2S;
      fbMyZyl2 : FB_Zyl2V2S;
END_VAR
fbMyZyl1.METH_Ausfahren();
fbMyZyl2.METH_Ausfahren();
END_PROGRAM
```



Unterschiedliche Anzahl Aktoren
aber gleiches Interface zur
Ansteuerung

```
INTERFACE ITF_Zyl
METHOD METH_Ausfahren : E_Status
//Keine Implementierung der Funktionalität!
END_METHOD
END_INTERFACE
```

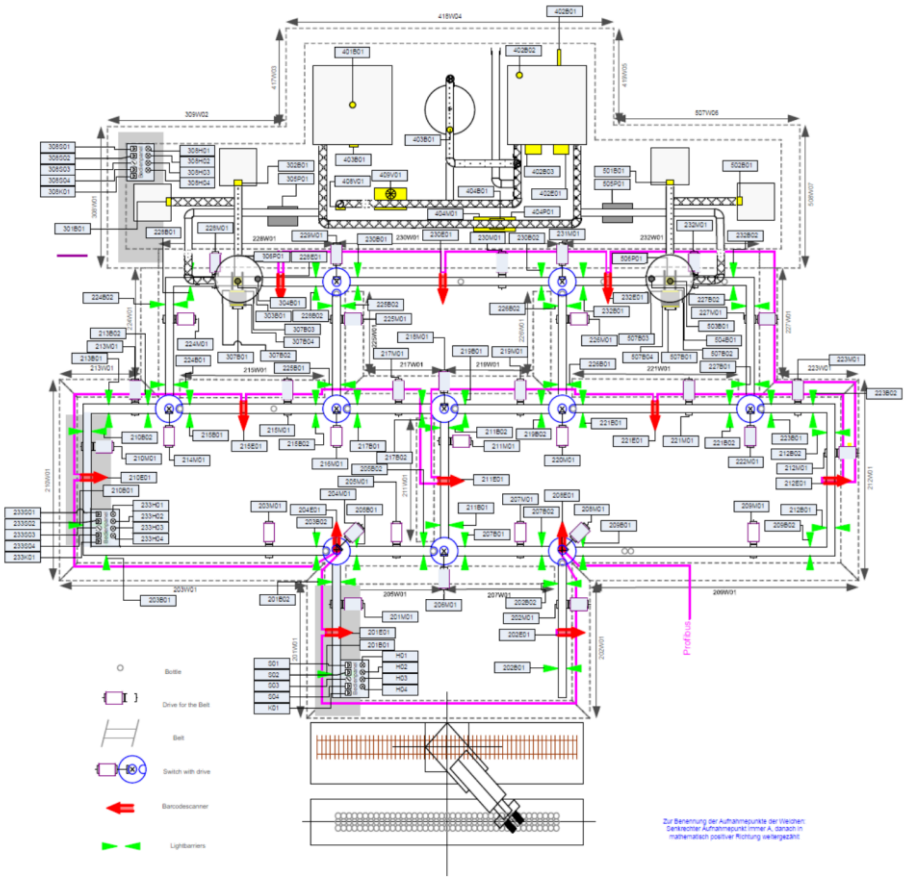
Einfach wirkender Zylinder

```
FUNCTION_BLOCK FB_Zyl1V2S IMPLEMENTS ITF_Zyl
VAR
      bVentil1  AT%Q* : BOOL;  //Ein Ventil

END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit einem Ventil
END_METHOD
END_FUNCTION_BLOCK
```

Doppelt wirkender Zylinder

```
FUNCTION_BLOCK FB_Zyl2V2S IMPLEMENTS ITF_Zyl
VAR
      bVentil1  AT%Q* : BOOL;  //...
      bVentil2  AT%Q* : BOOL;  //Zwei Ventile
END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit zwei Ventilen
END_METHOD
END_FUNCTION_BLOCK
```

Motivation: Mehrere Funktionsbausteine über die gleiche Schnittstelle ansteuern



- In realen automatisierten Anlagen der Industrie oft einige hundert Module

➢ Implementiert als FBs um Modularität und Wiederverwendung zu ermöglichen

- Üblicherweise müssen Anlagen (und deren Module) mehrere Betriebsarten besitzen
  - Bspw. Automatik, Manuell, Tippbetrieb, etc.

➢ Implementiert als Methoden in einem definierten Interface für alle FBs der Module

- Sollen im Programm alle Module in die gleich Betriebsart gebracht werden

➢ Aufruf der jew. Methode von allen FBs → hunderte von Codezeilen benötigt, oder?

Layout der Laboranlage Hybrides Prozessmodell:
22 Transportbänder, 12 Weichen, 1 Roboter, 2 Abfüllstationen, 1 Verfahrenstechnikstation
→ vergleichsweise sehr klein gegenüber realen Anlagen, bspw. Materialflusssystemen (Distributionszentrum), dort hunderte von Modulen

Einschub: Deklaration und Verwendung von Arrays anhand von zwei Beispielen

- Arrays dienen der Sammlung von mehreren Variablen des gleichen Datentyps
- Vorteil: Können im Programmcode durch Programmschleifen bearbeitet werden

Bsp. 1: Eindimensionales Array

Schlüsselwörter

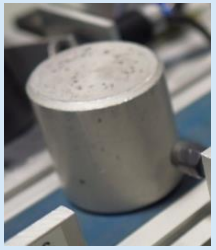
```
aWsSpeicher : ARRAY[1..3] OF E_WsType;
```

Name

Datentyp

Grenzen des Array
Hier: 1 bis 3 → 3 Elemente

```
TYPE E_WsType :
(none:=0, weis:=1, schwarz:=2, metall:=3);
END_TYPE
```

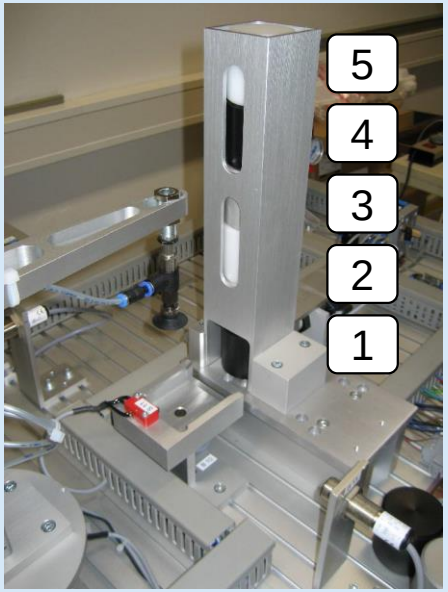


Element:	1	2	3	4	5
Wert:	schwarz	weis	metall	schwarz	weis

Bsp. 2: Arrays-Verarbeitung mittels einer FOR-Schleife, um Informationen der Werkstücke (WS) in einem Stapel zu speichern.

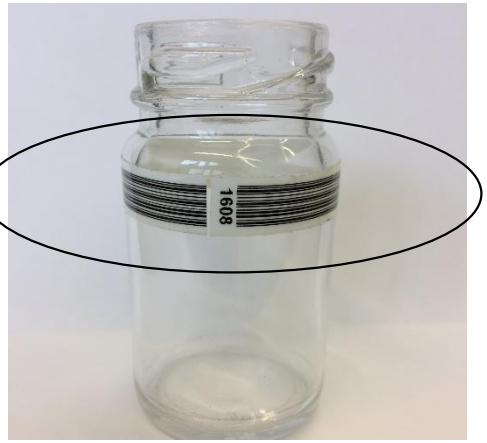
Hinweis: Initialisierung eines Arrays, d. h. hier Vorbelegung als schwarze WS
Wird nach Identifikation der WS-Art überschrieben

```
FUNCTION_BLOCK FB_Stapel
VAR
      aWsSpeicher  : ARRAY[1..5] Of E_WsType;
      iVar         : BOOL;
END_VAR
FOR iVAr := 1 TO 5 DO
      aWsSpeicher[iVar] := E_WsType.schwarz;
END_FOR
// ...
END_FUNCTION_BLOCK
```



Einschub: Deklaration und Verwendung von Arrays

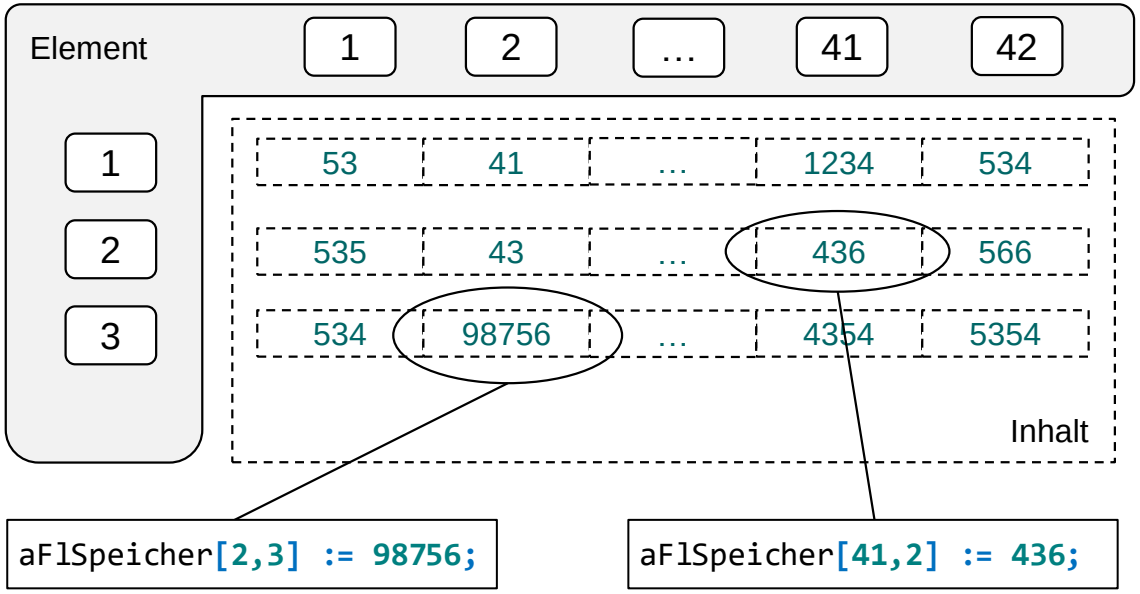
Ganzzahl als Barcode → Datentyp INT



Bsp.: Mehrdimensionales Array – Position eines Fläschchen im Roboterlager

```
aFlSpeicher : ARRAY[1..42,1..3] OF INT;
```

Grenzen des Array
Hier: (1 bis 42) X (1 bis 3)
→ 126 Elemente



Nutzung von Pointer-Arrays auf Interfaces – Erklärung mittels Beispiel-Aufgabe – Angabe

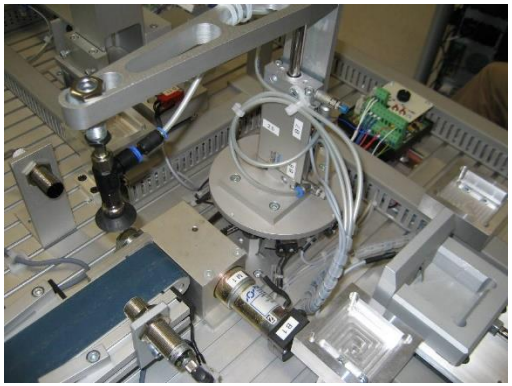
(1) Version der Stempelanlage enthält die Module *Stempel*, *Kran* und *Stapel* → Umsetzung als FB

(2) Funktionsbausteine der Module implementieren alle das Interface *ITF_Module*
   – Betriebsarten (Automatik, Manuell) der Module als Methode, Rückgabewert BOOL

(3) Gesucht wird eine Möglichkeit, in dem Main Programm alle Module, die das Interface implementieren, effizient in die jeweilige Betriebsart zu setzen (d.h. die jeweilige Methode aufzurufen)

Stempel

Kran



Stapel

Nutzung von Pointer-Arrays auf Interfaces – Erklärung mittels Beispiel-Aufgabe – Lösung Funtionsbaustein

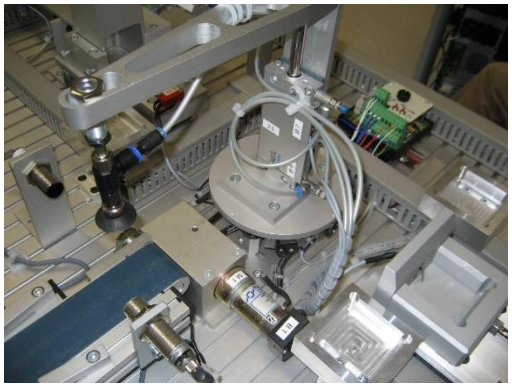
(1) Version der Stempelanlage enthält die Module *Stempel*, *Kran* und *Stapel* → Umsetzung als FB

(2) Funktionsbausteine der Module implementieren alle das Interface *ITF_Module*
- Betriebsarten (Automatik, Manuell) der Module als Methode, Rückgabewert BOOL

(3) Gesucht wird eine Möglichkeit, in dem Main Programm alle Module, die das Interface implementieren, effizient in die jeweilige Betriebsart zu setzen (d.h. die jeweilige Methode aufzurufen)

Stempel

Kran



Stapel

(1)

```
FUNCTION_BLOCK FB_Stempel
// …
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK FB_Kran
// …
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK FB_Stapel
// …
END_FUNCTION_BLOCK
```

Nutzung von Pointer-Arrays auf Interfaces – Erklärung mittels Beispiel-Aufgabe – Lösung Interface

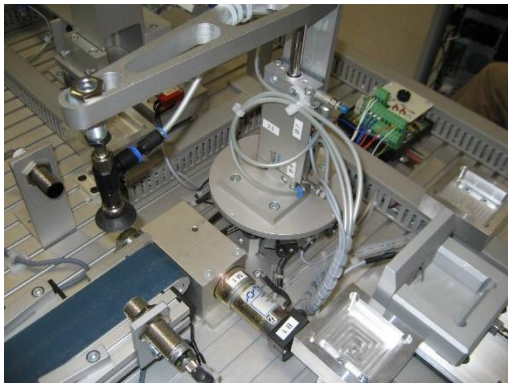
(1) Version der Stempelanlage enthält die Module *Stempel*, *Kran* und *Stapel* → Umsetzung als FB

(2) Funktionsbausteine der Module implementieren alle das Interface *ITF_Module*
- Betriebsarten (Automatik, Manuell) der Module als Methode, Rückgabewert BOOL

(3) Gesucht wird eine Möglichkeit, in dem Main Programm alle Module, die das Interface implementieren, effizient in die jeweilige Betriebsart zu setzen (d.h. die jeweilige Methode aufzurufen)

Stempel

Kran



Stapel

(2)

```
INTERFACE ITF_Module
METHOD METH_Auto :  BOOL END_METHOD
METHOD METH_Manu :  BOOL END_METHOD
END_INTERFACE
```

(1)

```
FUNCTION_BLOCK FB_Stempel  IMPLEMENTS ITF_Module
// …
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK FB_Kran  IMPLEMENTS ITF_Module
// …
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK FB_Stapel  IMPLEMENTS ITF_Module
// …
END_FUNCTION_BLOCK
```

Nutzung von Pointer-Arrays auf Interfaces – Erklärung mittels Beispiel-Aufgabe – Lösung (3)

```
PROGRAM Main
VAR
      fbStempel     :  FB_Stempel;
      fbKran        :  FB_Kran;
      fbStapel      :  FB_Stapel;

      apModules     :  ARRAY[1..3] OF ITF_Module
                    := [fbStempel,
                        fbKran,
                        fbStapel];

      iVar          :  INT;// Hilfsvariable zum Zählen
END_VAR


// ...

FOR iVar := 1 TO 3 DO
      apModules[iVar].METH_Manu();
END_FOR

// …

fbStempel.METH_Manu();
fbKran.METH_Manu();
fbStapel.METH_Manu();


END_PROGRAM
```

(3.1) Instanziierung der FBs der Module

(3.2) Deklaration eines Arrays vom Typ des definierten Interfaces

(3.3) Vorgabe der Instanznamen der FBs als Initialwert der Array-Elemente

(3.4) Jetzt kann im Code durch Iterieren über das Array in einer FOR-Schleife die Betriebsart aller Module gewechselt werden (durch Aufruf der Methode)

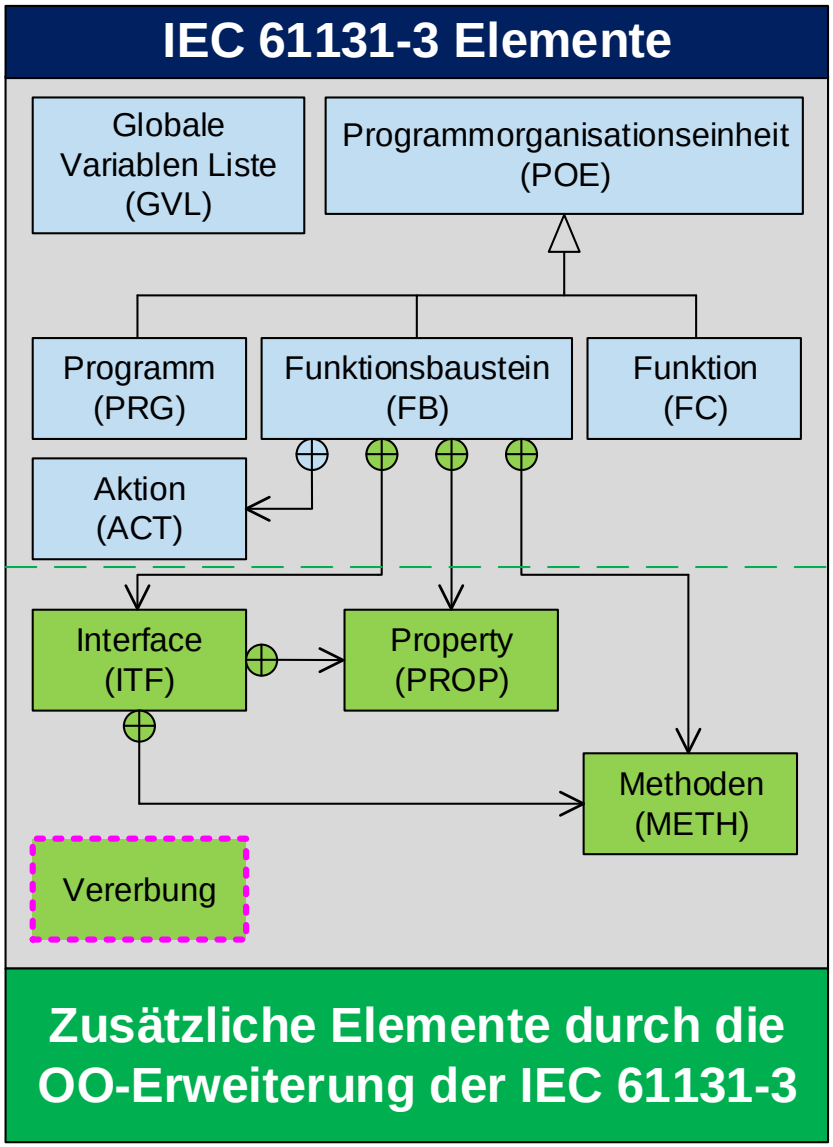
Zum Vergleich: diese Codezeilen haben den gleichen Effekt wie die FOR-Schleife davor

Hinweis:
Im Beispiel nur drei Module, deshalb werden auch ohne dieses Programmierkonstrukt nur drei Codezeilen benötigt;
In großen Materialflusssystemen einige hundert Module vorhanden, dort wären dann jedes Mal mehrere hundert Codezeilen für Wechsel der Betriebsart von allen nötig (wenn gefordert)!

Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detailplanung von AT-Systemen (Hardware-Auslegung)
- Kapitel 5 Detailplanung von AT-Systemen (Software-Auslegung)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - Programmiersprache Ablaufsprache (AS)
    - Programmiersprache Strukturierter Text (ST)
      - Funktionsbausteine und Methoden
      - Verwendung von Interfaces
      - **Vererbung in der IEC 61131-3**
      - Properties von Funktionsbausteinen

Zusätzliche Elemente in der objektorientierten (OO) Erweiterung der IEC 61131-3 – Vererbung



- Erweiterung eines Elements (FB, ITF, oder METH) durch ‚vererben' der Eigenschaften
- Vererbung nur zwischen gleichen Elementen möglich (FB→FB, …)
- Vererbung des Deklarationsteils und Methoden; Implementierungsteil abhängig von Schlüsselwort:
  FB → SUPER(), METH → SUPER
- Erweiterung der geerbten Variablen und Code durch zusätzliche Variablen und Code
- Verwendung des Schlüsselworts ‚EXTENDS'
  - (a) Vererbung zwischen FB, (b) Vererbung zwischen ITF, (c) Vererbung zwischen METH

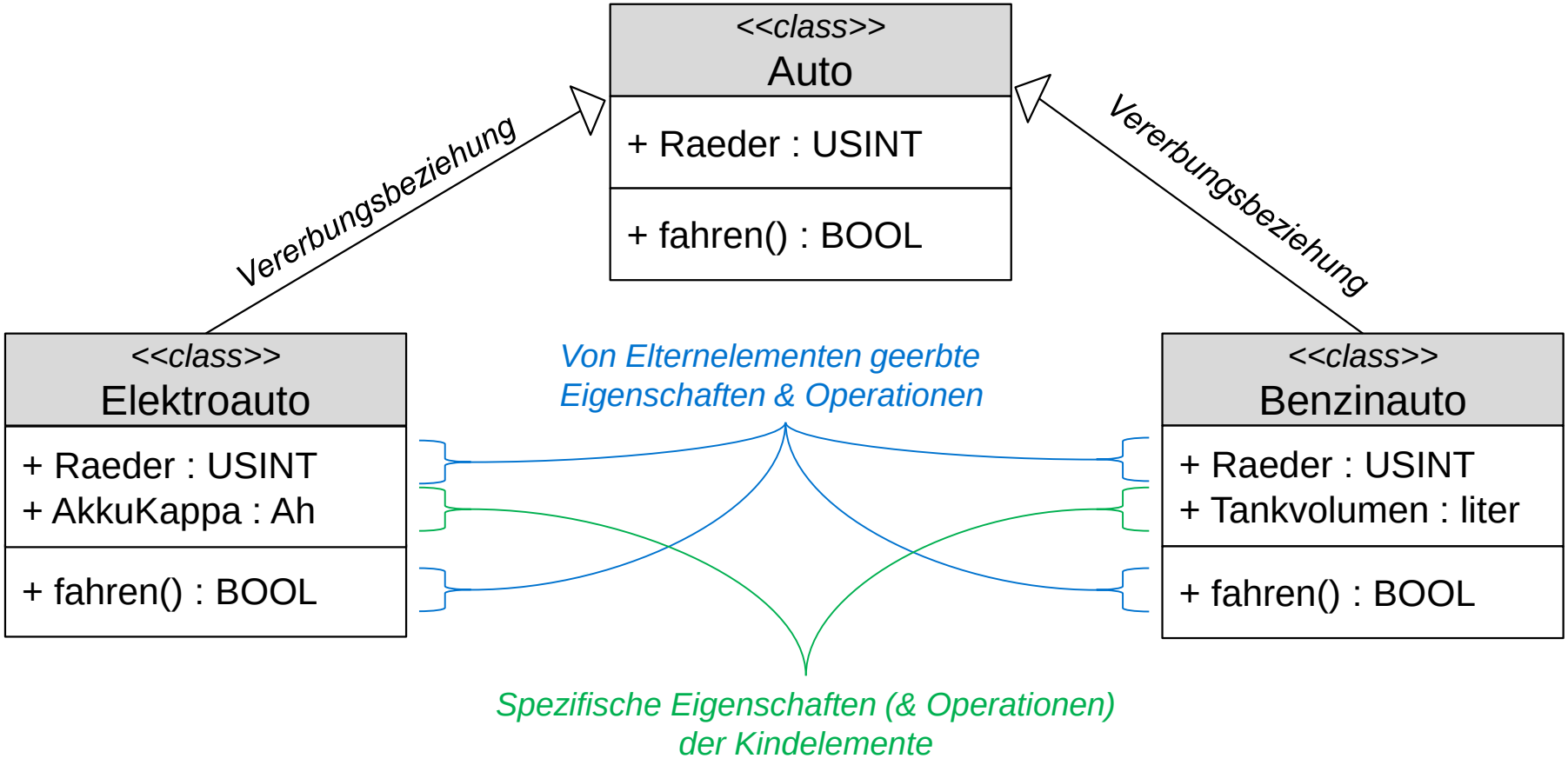
```
FUNCTION_BLOCK FB_Derived EXTENDS FB_Base
//…
END_FUNCTION_BLOCK
```
(a)

```
INTERFACE ITF_Derived EXTENDS ITF_Base
//…
END_INTERFACE
```
(b)

```
METHOD METH_Derived EXTENDS METH_Base
//…
END_METHOD
```
(c)

Exkurs: Vererbung in objektorientierten Sprachen am Beispiel der Unified Modeling Language (UML)

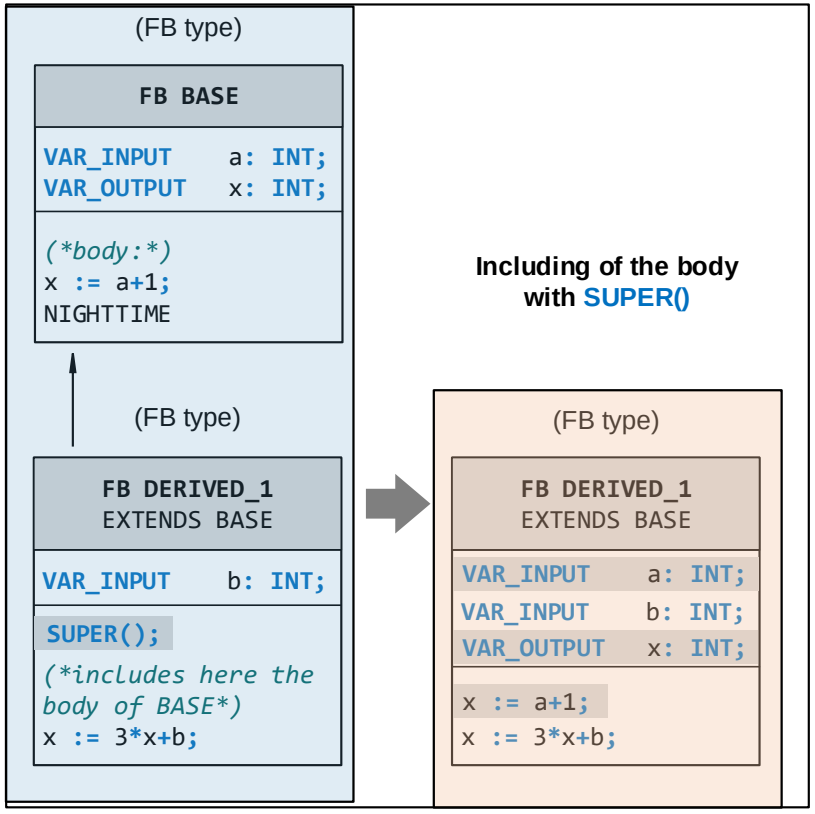
- Mittels Vererbung werden in UML alle Eigenschaften (*Attributes*) und Operationen (*Operations*) einer Klasse (*class*) an eine andere Klasse vererbt
- Durch Aufbau einer Vererbungshierarchie können bei Kind-Elementen so die Elemente der Eltern-Elemente wiederverwendet werden und müssen nicht neu deklariert werden



Beispiel: Zugriff auf den Implementierungsteil eines vererbenden (Elternteil) FBs – *SUPER()*

- Mit Schlüsselwort *SUPER()* kann im Implementierungsteil eines erbenden FBs (Kindelement) der Implementierungsteil des vererbenden FB ausgeführt werden.

1. Vererbung von FB_Base an FB_Derived1

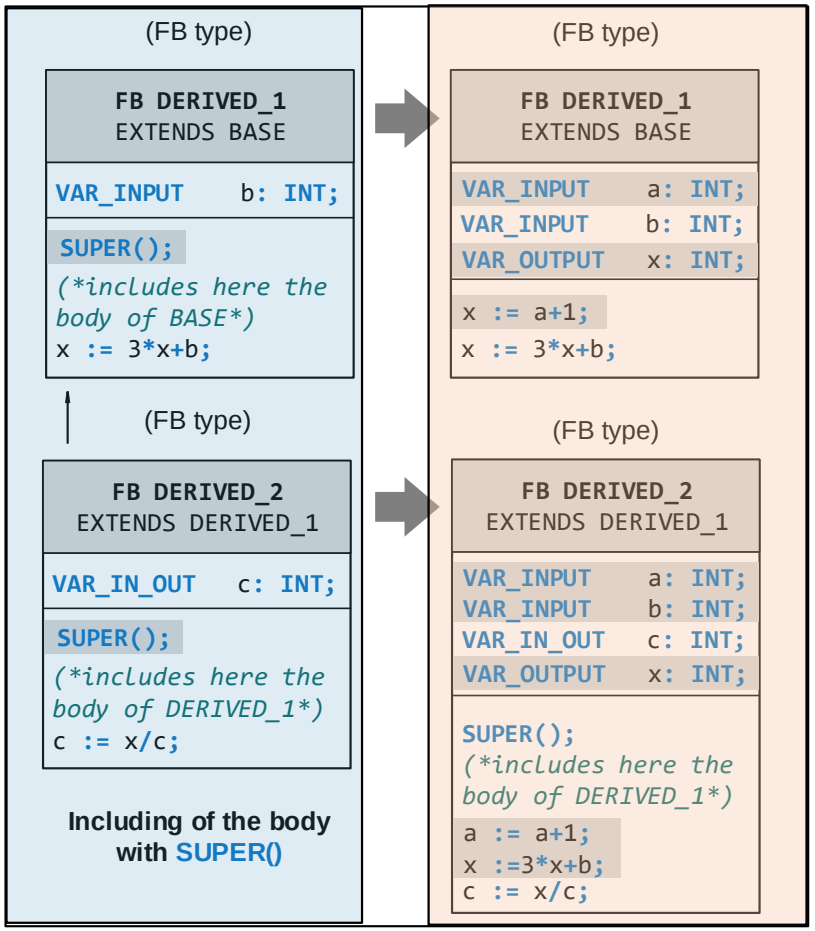


[IEC 61131-3, 3rd Edition]

Menschliche Sicht

Compiler Sicht

2. Vererbung von FB_Derived1 an FB_Derived2



[IEC 61131-3, 3rd Edition]

Beispiel: Zugriff auf den Implementierungsteil einer vererbenden (Elternteil) Methode (0) – *SUPER*

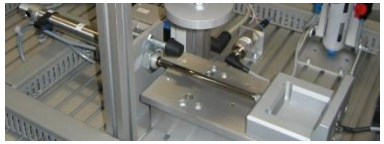
- Mit Schlüsselwort *SUPER* kann im Implementierungsteil einer erbenden Methode (Kindelement) der Implementierungsteil der vererbenden Methode ausgeführt werden

```
INTERFACE ITF_Zyl
METHOD METH_Ausfahren : E_Status END_METHOD
END_INTERFACE
```

Einfach wirkender Zylinder


```
FUNCTION_BLOCK FB_Zyl1V2S IMPLEMENTS ITF_Zyl
VAR
      bVentil1  AT%Q* : BOOL;

END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit einem Ventil
bVentil1 := TRUE;


END_METHOD
END_FUNCTION_BLOCK
```

Doppelt wirkender Zylinder



```
FUNCTION_BLOCK FB_Zyl2V2S
VAR

      bVentil2  AT%Q* : BOOL;
END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit zwei Ventilen



END_METHOD
END_FUNCTION_BLOCK
```

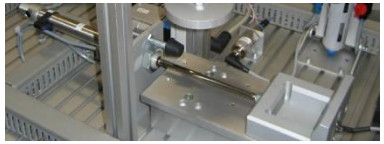
- Beide Zylinder von einem Programmierer zu entwickeln, effiziente Codeerstellung durch Wiederverwendung angestrebt

Beispiel: Zugriff auf den Implementierungsteil einer vererbenden (Elternteil) Methode (1) – *SUPER*

- Mit Schlüsselwort *SUPER* kann im Implementierungsteil einer erbenden Methode (Kindelement) der Implementierungsteil der vererbenden Methode ausgeführt werden

```
INTERFACE ITF_Zyl
METHOD METH_Ausfahren : E_Status END_METHOD
END_INTERFACE
```

Einfach wirkender Zylinder



Doppelt wirkender Zylinder

```
FUNCTION_BLOCK FB_Zyl1V2S IMPLEMENTS ITF_Zyl
VAR
      bVentil1  AT%Q* : BOOL;

END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit einem Ventil
bVentil1 := TRUE;


END_METHOD
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK FB_Zyl2V2S EXTENDS FB_Zyl1V2S
VAR
      (*Vererbung der Variable bVentil1*)
      bVentil2  AT%Q* : BOOL;
END_VAR
METHOD METH_Ausfahren : E_Status        (1)
// Implementierung mit zwei Ventilen
(*Methode wird durch EXTENDS geerbt, nicht
deren Implementierung*)

END_METHOD
END_FUNCTION_BLOCK
```

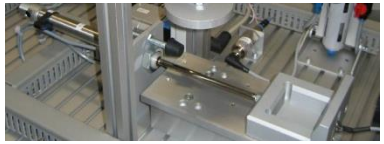
(1) **Vererben der Variablen** von FB_Zyl1V1S2 durch das **Schlüsselwort *EXTENDS***
Dadurch wird die Variable bVentil1 geerbt → Muss nicht deklariert werden
Auch werden alle Methoden geerbt, die erweitert werden können

Beispiel: Zugriff auf den Implementierungsteil einer vererbenden (Elternteil) Methode (2) – *SUPER*

- Mit Schlüsselwort *SUPER* kann im Implementierungsteil einer erbenden Methode (Kindelement) der Implementierungsteil der vererbenden Methode ausgeführt werden

```
INTERFACE ITF_Zyl
METHOD METH_Ausfahren : E_Status END_METHOD
END_INTERFACE
```

Einfach wirkender Zylinder



Doppelt wirkender Zylinder

```
FUNCTION_BLOCK FB_Zyl1V2S IMPLEMENTS ITF_Zyl
VAR
      bVentil1  AT%Q* : BOOL;

END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit einem Ventil
bVentil1 := TRUE;


END_METHOD
END_FUNCTION_BLOCK
```

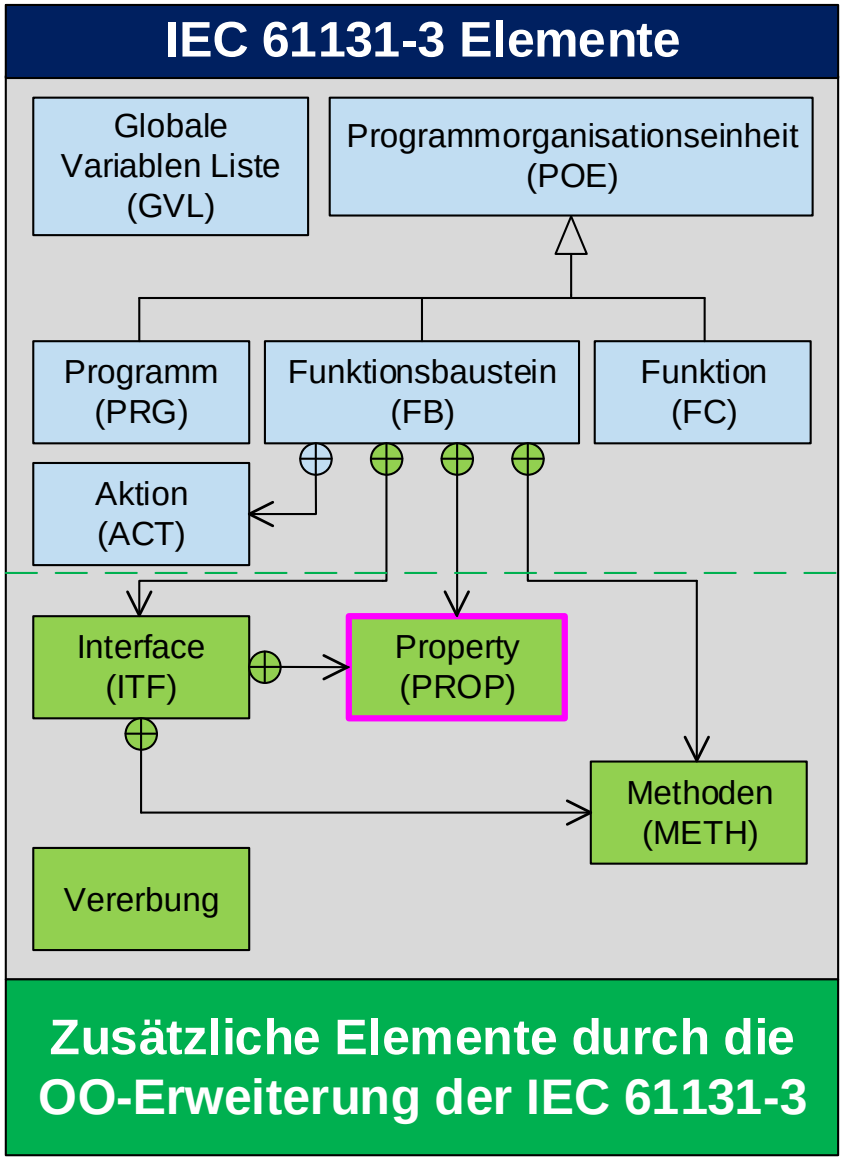
```
FUNCTION_BLOCK FB_Zyl2V2S EXTENDS FB_Zyl1V2S
VAR
      (*Vererbung der Variable bVentil1*)
      bVentil2  AT%Q* : BOOL;
END_VAR
METHOD METH_Ausfahren : E_Status
// Implementierung mit zwei Ventilen
SUPER;                                        (2)
(*Implementierung wird durch SUPER; geerbt*)
bVentil2 := FALSE; //Erweiterung
END_METHOD
END_FUNCTION_BLOCK
```

(2) **Vererben des Implementierungsteils** der Methode von FB_Zyl1V1S2 durch das **Schlüsselwort SUPER** im Implementierungsteil der Methode des Kindelements (*bVentil1 := TRUE;* wird geerbt)
Erweiterung durch zusätzlichen Code zur Ansteuerung des zweiten Ventils (bVentil2 := FALSE;)
→ Wiederverwendung und Erweiterung von geerbtem Code = Einsparung

Lehrveranstaltung AT 1 - Überblick

- Kapitel … …
- Kapitel 4 Detailplanung von AT-Systemen (Hardware-Auslegung)
- Kapitel 5 Detailplanung von AT-Systemen (Software-Auslegung)
  - Elemente in einem IEC 61131-Projekt
  - Variablendeklaration und Hardwarezugriff
  - Einführung in die Programmiersprachen der IEC 61131-3
  - Objektorientierung und weitere Programmiersprachen der IEC 61131-3
    - Programmiersprache Ablaufsprache (AS)
    - Programmiersprache Strukturierter Text (ST)
      - Funktionsbausteine und Methoden
      - Verwendung von Interfaces
      - Vererbung in der IEC 61131-3
      - **Properties von Funktionsbausteinen**

Zusätzliche Elemente in der OO IEC 61131-3 – Property (PROP)

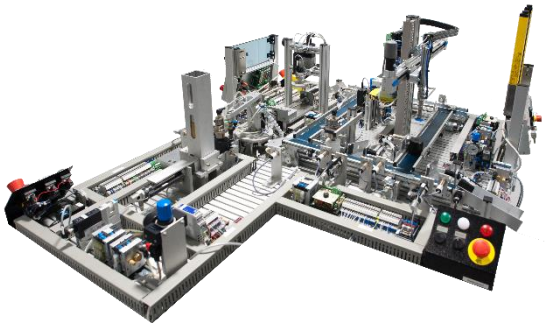


- Elemente zur Angabe eines definierten Zugriffs auf einen Wert (vgl. Methode → Zugriff auf eine Funktionalität)
- Properties werden hauptsächlich in Interfaces definiert (aber auch in FB möglich)
  - Gute Möglichkeit in einem Interface Eigenschaften eines FBs zu definieren Bspw.: Alle Linearaktoren (Zylinder) müssen als Eigenschaft den Wert der Kraft haben, mit der sie ausfahren
- Ein Funktionsbaustein muss den jeweiligen Wert damit nicht direkt als Variable enthalten
- Enthalten ein Methoden-Paar zum Lesen (get-Methode) und Schreiben (set-Methode) des jeweiligen Wertes
- Implementierungsteile dieser Methoden nur für FBs vorhanden, für Interfaces leer
- Nicht im Standard der IEC 61131-3 enthalten aber von manchen Programmiersystemen (bspw. CODESYS) umgesetzt

Interfaces zur Deklaration von gemeinsamen Properties – Beispiel: Pneumatikzylinder und elektrischer Linearantrieb (1)

Hauptprogramm

```
PROGRAM Main
VAR
      fbMyZyl      : FB_Zyl;
      fbMyLinAn    : FB_LinAn;
END_VAR
fbMyZyl.pfKraft    :=53.12;
fbMyLinAn.pfKraft  :=46.57;
END_PROGRAM
```



?

Pneumatikzylinder

Elektrischer Linearantrieb

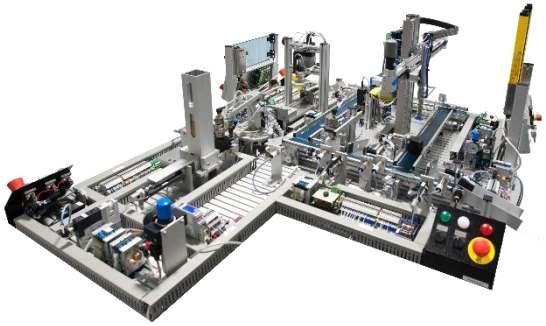
Kraft = Druck * Fläche

Kraft = f(Stromaufnahme)

Interfaces zur Deklaration von gemeinsamen Properties – Beispiel: Pneumatikzylinder und elektrischer Linearantrieb (2)

Hauptprogramm

```
PROGRAM Main
VAR
      fbMyZyl      : FB_Zyl;
      fbMyLinAn    : FB_LinAn;
END_VAR
fbMyZyl.pfKraft    :=53.12;
fbMyLinAn.pfKraft  :=46.57;
END_PROGRAM
```



Unterschiedliche Parameter zur Berechnung aber gleiche Property im Interface zum Zugriff auf Eigenschaft

```
INTERFACE ITF_Example
PROPERTY pfKraft : REAL  END_PROPERTY
// Im Interface keine Implementierung des
// Methoden-Paares Get() und Set()
END_INTERFACE
```

Pneumatikzylinder

Elektrischer Linearantrieb

```
FUNCTION_BLOCK FB_Zyl IMPLEMENTS ITF_Example
VAR
      fFlaeche            : REAL;
      iDruck      AT%Q*   : DINT;
END_VAR
// ...
END_FUNCTION_BLOCK
```

≠

```
FUNCTION_BLOCK FB_LinAn IMPLEMENTS ITF_Example
VAR
      fMaxStrom   AT%Q*   : DINT;
END_VAR
// ...
END_FUNCTION_BLOCK
```

Property – Deklaration und Implementierung am Beispiel ‚Kraft' im Interface für Linearaktoren (Zylinder)

Beispiel - **Deklaration** einer Property „*pfKraft*" in einem Interface:

```
INTERFACE ITF_Example
PROPERTY pfKraft : REAL  END_PROPERTY
// Im Interface keine Implementierung des
// Methoden-Paares Get() und Set()
END_INTERFACE
```

Beispiel – Property Methoden „Get" und „Set" in einem FB **ausimplementiert** (Hinweis: Definiert im Interface, s.o.)

```
FUNCTION_BLOCK FB_Zyl IMPLEMENTS ITF_Example
VAR
      fFlaeche            : REAL;
      iDruck      AT%Q*   : DINT;
END_VAR
PROPERTY pfKraft : REAL
// Implementierung der Methoden erst im FB

      METHOD Get()
        pfKraft := fFlaeche * iDruck;
      END_METHOD

      METHOD Set()
        iDruck  := REAL_TO_INT(pfKraft / fFlaeche);
      END_METHOD

END_PROPERTY
END_FUNCTION_BLOCK
```

Zylinder als Beispiel:

Zylinder im Beispiel berechnet seine Kraft aus Druck und Fläche (seinen Parametern, s. links) weil Pneumatik;

Interface kann auch durch einen elektrisch betriebenen Zylinder (Linearantrieb) implementiert werden, dann andere Berechnung der Kraft

Get-Methode (Lesen) hat immer automatisch die Property (hier: *pfKraft*) als Rückgabewert

Set-Methode (Schreiben) hat immer automatisch die Property (hier: *pfKraft*) als Eingabewert

Hinweis: **Get- und Set-Methode** verwenden abgesehen von der Property nur **lokale Variablen** des FBs

Property – Deklaration und Implementierung am Beispiel ‚Kraft' im Interface für Linearaktoren (Zylinder)

Property in einem FB (Implementiert mit dem Interface):

```
FUNCTION_BLOCK FB_Zyl IMPLEMENTS ITF_Example
VAR
      fFlaeche            : REAL;
      iDruck      AT%Q*   : DINT;
END_VAR
// vollständiger Baustein: siehe vorige Folie
END_FUNCTION_BLOCK
```

Zylinder als Beispiel:

Verwendung der Properties einer Instanz eines FBs:

```
PROGRAM Main
VAR
      fbZyl         : FB_Zyl;
      fIstKraft     : REAL;
END_VAR
// Zuweisung eines Werts
fbZyl.pfKraft       := 2.71;
// automatischer Aufruf der Set-Methode

// Zuweisung eines Werts
fIstKraft           := fbZyl.pfKraft;
// automatischer Aufruf der Get-Methode

END_PROGRAM
```

Schreiben auf einer Property wird vom Compiler automatisch übersetzt zu (sinngemäß):

```
fbZyl.pfKraft.Set(pfKraft := 2.71;);
```

Lesen einer Property wird vom Compiler automatisch übersetzt zu (sinngemäß):

```
fIstKraft := fbZyl.pfKraft.Get();
```

Also:
bei lesendem Zugriff → Aufruf und Ausführung der Get-Methode
bei schreibendem Zugriff → Aufruf und Ausführung der Set-Methode

Rückblick

Mit den Inhalten dieser Vorlesung sollten Sie …

- … wissen welche **Elemente** in **der objektorientierten Erweiterung der IEC 61131-3** hinzugekommen sind und wie diese aufgebaut sind
- … die Grundlagen für die Programmiersprachen **Ablaufsprache und Strukturierten Text** kennen und **einfache Programme erstellen** können
- … wissen, **wie Aktionen, Interfaces, Methoden deklariert werden** können, **wie sie verwendet werden** und **welchen Vorteil** sie bieten
- … wissen was **Arrays** sind und wie diese verwendet werden
- … wissen, wie **Vererbung zwischen Elementen** in einem IEC 61131-3 Projekt funktioniert und wie sie eingesetzt wird
- … in der nun **folgenden Übung aktiv mitarbeiten** können