# Automatisierungstechnik nach internationaler Norm programmieren (20)

## Mächtiges Werkzeug Byte- und Wortbearbeitung

Ulrich Becker

**Folge 19 schloss das Thema »IT in der Automatisierungstechnik« mit Untersuchungen zum Mailversand in lokalen Automatisierungsnetzen ab. In dieser vorletzten Folge der Serie werden an einem Beispiel Methoden der Byte- und Wortverarbeitung dargelegt. Es wird weiter zum Ausgangspunkt der Folge zurückgekehrt: Darunter war die Frage, wie weit man das Wissen von Step7 erweitern muss, wenn Komponenten eingesetzt werden sollen, welche nach IEC 61131-3 zu programmieren sind.**

Jahrzehntelang konnte Steuerungstechnik fast ausschließlich nur mit Kontakten von Relais und Schützen betrieben werden. Reihenschaltung von Kontakten führten zu UND-Verknüpfungen, Parallelschaltungen zu ODER und mit Öffnerkontakten wurden Negationen realisiert. Alle Signale waren parallel verarbeitete binäre Signale. Die Boolesche Algebra lieferte hilfreiche Regeln für die Verknüpfung derartiger Signale. Verständlich sind auch Bestrebungen, durch Vereinfachung von Schaltfunktionen möglichst viele Relaiskontakte einzusparen.

### Der Klassiker: Verknüpfungssteuerungen und binäre Logik

Die Zeiten haben sich deutlich gewandelt. Viele Aufgaben der Automatisierungstechnik, welche vormals mit binärer Logik und minimierten Schaltnetzen gelöst wurden, sind heute eleganter unter Verwendung von Byte- oder Wort-Operationen zu bearbeiten. Dies wird nachfolgend an einem Beispiel »Fertigungszelle« (**Bild 111**) demonstriert. Dieses ist sehr einfach gewählt und deshalb durchaus auch mit binärer Logik zu lösen. Dem aber stellen wir eine alternative Lösung gegenüber.

In der Fertigungszelle werden drei Typen Teile A, B oder C im Durchlauf bearbeitet.

Bei positiver Flanke des Signals des Endlagenschalters $S_0$ wird der Typ des Bauteils durch Abtasten mit vier Lichtschranken erfasst. Je nach Teileform sind dann die Ventile Y1 ... Y4 sowie die Antriebe M1 und M2 dauerhaft wie in **Tabelle 15** angegeben zu schalten. Eine positive Signalflanke des Endlagenschalters S1 schaltet alle Aggregate aus. In Bild 111 wurde die Belegung der Lichtschranken als Byte im Dualcode (2#) und auch im Hexacode (16#) aufgeführt. Jede abgedunkelte Lichtschranke liefert ein FALSE-Signal.

Klassische Lösungen von Schaltnetzen führten zumeist über Schalttabelle und Schaltgleichung zum Ergebnis. In der Tabelle 15 werden alle Kombinationen der Eingangssignale aufgeführt, zugehörige gültige Ausgangsbelegungen eingetragen und daraus die Schaltgleichung

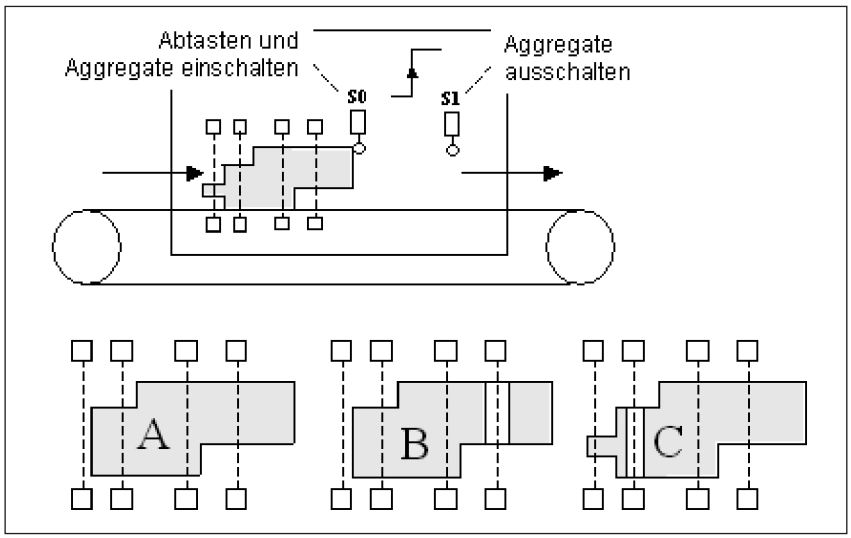


**Bild 111: Aufgabenstellung Fertigungszelle**

**Aggregatzustände**

| TYP A | Typ B | Typ C |
|---|---|---|
| Y0 = FALSE | Y0 = FALSE | Y0 = TRUE |
| Y1 = TRUE | Y1 = TRUE | Y1 = FALSE |
| Y2 = TRUE | Y2 = FALSE | Y2 = TRUE |
| Y3 = FALSE | Y3 = FALSE | Y3 = TRUE |
| M0 = TRUE | M0 = FALSE | M0 = TRUE |
| M1 = TRUE | M3 = TRUE | M1 = TRUE |
| **Belegung des Ausgangsbytes:** | | |
| 2# 0011_0110 | 2# 0010_0010 | 2# 0011_1101 |
| 16# 36 | 16# 24 | 16# 3D |

**Tabelle 15: Gewünschte Schaltung der Aggregate bei der Bearbeitung**

Dr. *Ulrich Becker*, Fachzentrum Automatisierungstechnik und vernetzte Systeme, BTZ-Rohr-Kloster, HWK Südthüringen
ulrich.becker@btz-rohr.de

**Fortsetzung aus »de« 22/2006, S. 66 ff.**

entwickelt. Bei vier Eingangssignalen muss man immerhin 16 Kombinationen berücksichtigen. Im übersichtlichen Beispiel aber lassen sich die Schaltgleichungen unmittelbar aus Tabelle 15 ablesen. So muss Ventil $Y_0$ nur eingeschaltet werden, wenn ein Werkstück vom Typ C erkannt wurde. Ventil $Y_1$ muss dagegen für die Typen B und C geschaltet werden. Beispielhafte Ergebnisse der klasischen Lösung zeigt **Bild 112.** $E_0$ bis $E_3$ stehen für die vier Lichtschrankensignale.

## Eine alternative Lösung

Während bei der klassischen Lösung jedes Ausgangsbit einzeln bearbeitet wird, erlauben Byte- und Wortbefehle das gleichzige Bearbeiten von acht, 16 oder sogar 32 Bit im Byte-, Wort- oder Doppelwortformat.

In den weiteren Ausführungen dieser Folge wird die Programmierung nach CoDeSys derjenigen nach Step7 gegenübergestellt. Damit kommrn wir auf die Frage zurück, inwieweit das verbreitete Wissen über Step7 bei Einsatz anderer Komponenten erweitert werden muss (siehe Folge 1). In den Programmbeispielen ist die Step7-Syntax blau gekennzeichet.

Die Byte- bzw. Wortbearbeitung von Ein- und Ausgangssignalen gelingt solange problemlos, wie die Kartenbaugruppen oder Busklemmen problemorientiert genau für diese Formate zur Verfügung stehen. Das ist aber eher die Ausnahme.

Für das Beispiel »*Fertigungszelle*« (**oberer Kasten, S. 64**) seien die vier Lichtschranken auf Busklemmen mit Adresse %IX1.0 bis 1.3 gelegt. Die Aktoren seien an Ausgängen %QX 4.0 bis 4.5 angeschaltet. Die Byteoperation LD %IB1 würde dann mit den Bits 6 und 7 auch die Signale weiterer Busklemmen erfassen, welche für die Fertigungszelle nicht relevant sind. Genauso würde die Byteoperation ST %AB4 auch nicht relevante Ausgangsbusklemmen beschreiben. Deshalb müssen die Nutzsignale in Zwischenspeicher der gewünschten Formate eingetragen werden, um sie dann dort zu bearbeiten.

Der Kasten »*Programm ›Fertigungszelle‹*« (nächste Seite) zeigt die Variablendeklaration des Programmbeispiels. Als Zwischenspeicher dienen die Variablen mit Namen »*Lichtgitter*« und »*Aktoren*« von Byteformat (fett hervorgehoben). Die Endlagenschalter $S_0$ und $S_1$ wurden an Busklemmen mit Adressen %IX0.6 und 0.7 geschaltet.

Zunächst zeigt **Bild 113** eine unvollständige Gegenüberstellung von Bit- und Byteverarbeitung. Wichtig ist: Bei CoDeSys unterscheiden die Operationen *Laden* (LD), *Logik* (AND, OR, XOR und NOT) sowie *Zuweisen* (ST) nicht zwischen Bit- und Byte-/Wort-/Doppelwortformat, während man dies bei Step7 vielfach sorgfältig unterscheiden muss. Wie erhalten die Zwischenspeicher genau die richtigen Biteinträge, mit denen dann die Byte- oder Wortverarbeitung gestartet werden kann? Selbstverständlich kann man mit den Befehlen *Laden* (LD) und *Store* (ST) einzelne Bits schreiben. CoDeSys ermöglicht in einfachster Weise das

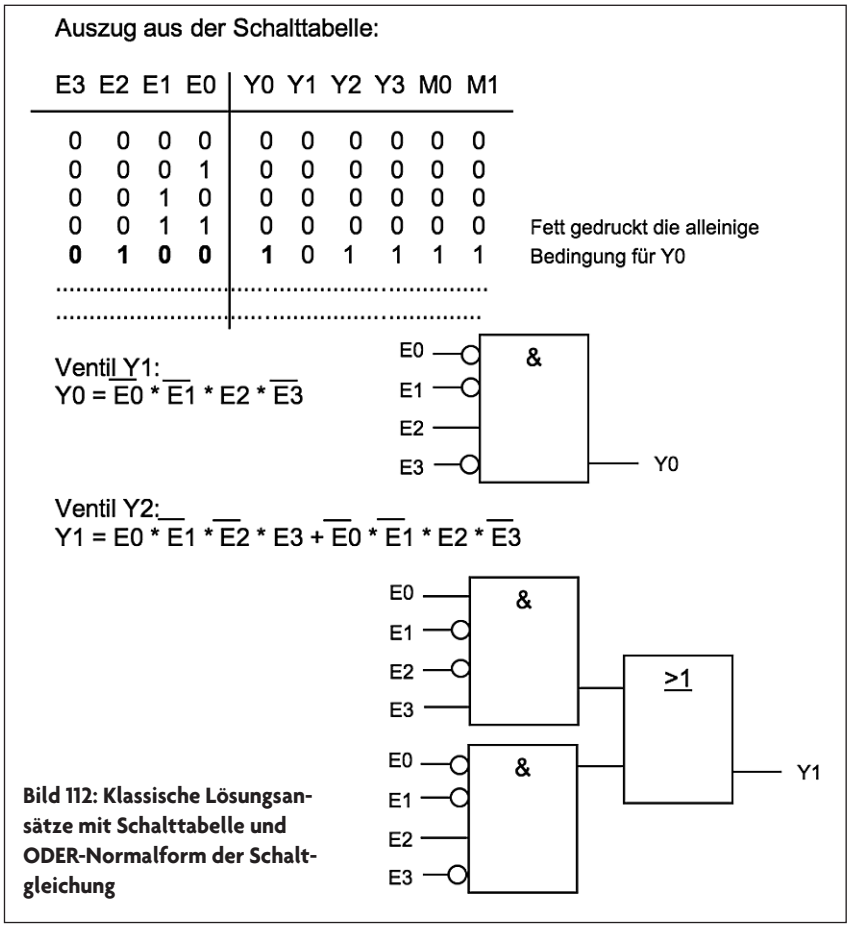


**Bild 112: Klassische Lösungsansätze mit Schalttabelle und ODER-Normalform der Schaltgleichung**

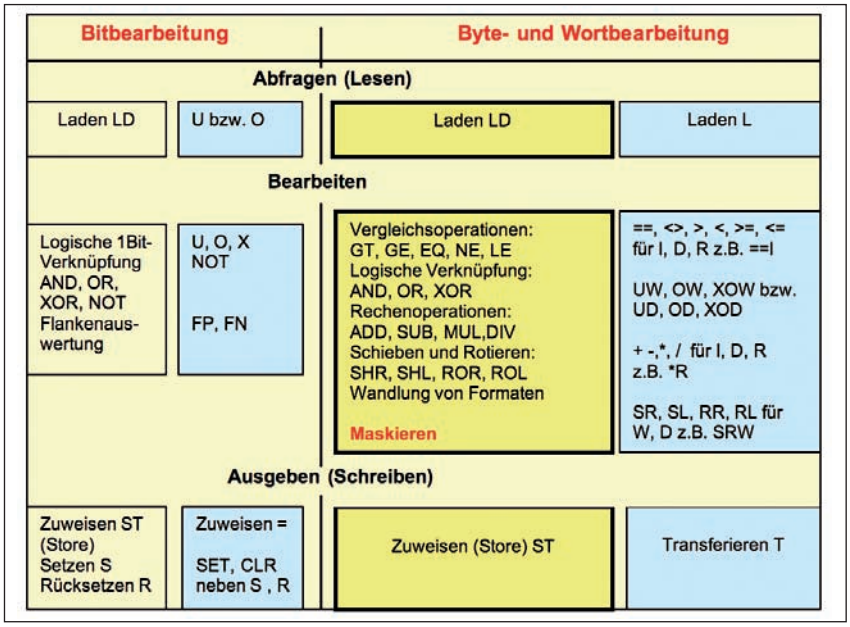


**Bild 113: Grundsätzliche Methodik der Bit- und Byte- bzw. Wortverarbeitung**

## PROGRAMM »FERTIGUNGSZELLE«

```
VAR
S0 AT %IX0.6: BOOL;
S1 AT %IX0.7: BOOL;
Eingangsbyte AT %IB1:BYTE;
Lichtgitter:BYTE;
Aktoren:BYTE;
Ausgangsbyte AT %QB4:BYTE
posFl_0:R_TRIG;
posFl_1:R_TRIG;
END_VAR
```

Direkter Vergleich wenig sinnvoll; siehe dafür Symboltabelle 16.

Für Lichtgitter und Aktoren wäre die Deklaration von **Statischen Variablen** mit Byteformat zweckmäßig.

Im Beispiel aber wird mit dem **Merkerbereich** gearbeitet. Alternativ wären auch Bytes eines globalen Datenbausteins möglich.

## EIN- UND AUSGANGSSIGNALE

**Bitweiser Eintrag von Ein- und Ausgangs-Signalen (Beispiele, nicht weiter verfolgt)**

| | **absolut** | **symbolisch** |
|---|---|---|
| LD Eingangsbyte.0 | U E1.0 | U »Lichtschranke_0« |
| ST Lichtgitter.0 | = M100.0 | = »Lichtgitter_0« |
| oder | oder | |
| LD Aktoren.3 | U M102.3 | U »Aktoren_3« |
| ST Ausgangsbyte.3 | = A4.3 | = »Y3« |

## ANWENDUNG EINER MASKE

**Anwendung einer Maske für die Bearbeitung der Eingänge**

| | | **Belegung des Byte** | **absolut** | **symbolisch** |
|---|---|---|---|---|
| LD | Eingangsbyte | a a a a _ n n n n | L EB1 | L »Eingangsbyte« |
| AND | 16#0F | 0 0 0 0 _ 1 1 1 1 (Maske) | L B#16#0F | L B#16#0F |
| ST | Lichtgitter | 0 0 0 0 _ n n n n | UW | UW |
| | | | T MB100 | T »Lichtgitter« |

**Anwendung einer Maske für die Bearbeitung der Eingänge**

| | **Belegung des Byte** | **absolut** | **symbolisch** |
|---|---|---|---|
| LD Ausgangsbyte | a a x x _ x x x x | L AB4 | L »Ausgangsbyte« |
| AND 16#C0 | 1 1 0 0 _ 0 0 0 0 (Maske) | L B#16#C0 | L B#16#C0 |
| (ST Ausgangsbyte) | a a 0 0 _ 0 0 0 0 | UW | UW |
| OR Aktoren | 0 0 n n _ n n n n | OR MB102 | OR »Aktoren« |
| ST Ausgangsbyte | a a n n _ n n n n | T AB4 | T »Ausgangsbyte« |

## Zuordnungstabelle

| **Element** | **Operand** | **Symbol** |
|---|---|---|
| Schalter S0 | E 0.6 | »S0« |
| Schalter S1 | E0.7 | »S1« |
| Hilfsbit Flankenauswertung 0 | M104.0 | »Flankenmerker_0« |
| Hilfsbit Flankenauswertung 1 | M104.1 | »Flankenmerker_1« |
| Eingangsbyte1 | EB1 | »Eingangsbyte« |
| Zwischenspeicher Lichtgitter | MB100 | »Lichtgitter« |
| Bit 0 von Lichtgitter | M100.0 | »Lichtgitter_0« |
| Zwischenspeicher Aktoren | MB102 | »Aktoren« |
| Bit 3 von Aktoren | M102.3 | »Aktoren_3« |
| Ausgangsbyte 4 | AB4 | »Ausgangsbyte« |

**Tabelle 16: Zuordnungstabelle mit Symbolen für Step7**

Selektieren der Bits von Variablen, wie nacholgend beispielhaft gezeigt (**Kasten** *»Ein- und Ausgangssignale«*, **links**). Bei Step7 löst man Bytes oder Worte über Merker oder globale Datenbits auf.

Grundsätzlich ist aber das Arbeiten mit Masken und wortweiser UND-, ODER- bzw. auch Exklusiv-ODER-Verknüpfung eleganter. Nach Maskierung liegen im Byte *»Lichtgitter«* genau die vier Bits der Lichtschranken. Hierbei bezeichen *»n«* die Bits der Nutzsignale und *»a«* die Bits anderer denkbarer Programmteile, welche auf Eingänge 4 bis 7 des Eingangsbytes zugreifen mögen.

Bei der Befehlsausgabe ist ähnlich vorzugehen. Zunächst werden mit der Maske Bit 6 und 7 selektiert, welche andere Programmteile in das Ausgangsbyte schreiben mögen (**Kasten:** *»Anwendung einer Maske«*, **links unten**). Danach werden die Nutzsignale Bit 0 bis 5 über ODER-Verknüpfung »beigefügt«. Auf den in Klammer gesetzten Befehl *»ST Ausgangsbyte«* kann hier auch verzichtet werden. Mit ihm wird jedoch die Belegung des Bytes im Zwischenschritt sichtbar.

Besondere Beachtung erfordern hier die Sprungbefehle. Warum sind Sprünge unverzichtbar? Binäre Logik erlaubt stets, Ausgangsbits abhängig von Bedingungen zu schreiben, denn die Befehle Zuweisen (=), Setzen (S) und Rücksetzen (R) sind »VKE-abhängig«. VKE bezeichnet das Verknüpfungsergebnis (1 Bit) der vorgelagerten Logik. Das Schreiben von Bytes oder Worten mit Store (ST) bzw. Step7: Transferiere (T) ist dagegen nicht abhängig vom VKE. Deshalb muss das Schreiben unterschiedlicher Werte in gleiche Adressaten mit Sprungbefehlen gezielt organisiert werden. Wird dies unterlassen, entstehen de facto die gefürchteten »Mehrfachzuweisungen«.

Wichtig ist weiter, dass trotz der Sprünge der Zyklus der Bearbeitung stets gesichert ist. Dies wird durch Beendigung des Programms an bestimmten Stellen und Rücksprung mit den Befehlen *RET* bzw. *BEA* bewirkt. Hier muss auch überlegt werden, wo zyklisch abzuarbeitende Programmteile und wo spezielle nicht zyklische Teile anzuordnen sind. Im Programm *Fertigungszelle* (oberer Kasten, links) wird deshalb die Bearbeitung der Ausgänge wie die der Eingänge an den Anfang des Programms gesetzt. Weiter ist dafür zu sorgen, dass die Lichtschranken das Werkstück nicht zyklisch, sondern einmalig abtasten. Das wird durch Flankenauswertung der End-

lagenschalter und gezielte Sprünge zu den Marken M1 und M2 bewirkt. Für die im »*Step7-Programm*« (**Kasten oben**) verwendeten Symbole gilt die Zuordnungstabelle **Tabelle 16, S. 64**.

## Umdenken Step7 – CoDeSys IEC 61131-3: ein Problem?

Die unterschiedliche Syntax bestimmter Operationen im IEC-61131-Programmiersystem CoDeSys und im System Step7 stellt bei einiger Routine und Verwendung der jeweiligen Referenzlisten sicher kein eigentliches Problem dar. Wohl aber sind Details der Datenverwaltung und der unterschiedliche Einsatz von Variablen und Symbolen stets zu beachten. Hierzu nochmals einige Ausführungen:

Sowohl CoDesys als auch Step7 verwenden globale und lokale Variablen. Bei lokalen Variablen besteht Übereinstimmung darin, dass deren Adressen in den Grenzen der verfügbaren Datenspeicher vorteilhaft vom System selbst verwaltet werden. Allerdings bestehen in Details der Nutzung erhebliche Unterschiede (**Bild 114**). Hierzu gehört auch die unterschiedliche Interpretation von Instanz eines Funktionblockes (CoDeSys) und Instanzdatenbaustein eines Funktionsbausteins (Step7).

Im System Codesys werden auch globale Variablen vom System automatisch verwaltet.

Demgegenüber muss der Step7-Programmierer diese Daten selbst adressieren. Insbesondere die Verwendung von Merkern führt durch Mehrfachverwendung und unbeabsichtigten Byte- und gleichzeitig Bit-Zugriff immer wieder zu Fehlern. Sie können eingeschränkt werden, wenn man für alle globalen Daten konsequent Symbole verwendet. In Step7 wird dazu eine Symboltabelle editiert. Formal ergeben Symbole und Variablen ein vergleichbares Bild. Es bleibt aber der grundsätzliche Unterschied, dass hinter jedem Symbol eindeutig eine vom Programmierer festgelegte Adresse in den Speicherbereichen Merker, Datenbausteine, E/A-Peripherie oder Prozessabbild liegt.

Die symbolische Programmierung wird im System Step7 in gleicher Weise erweitert auf die globalen Prozessabbilder der Ein- und Ausgänge. Demgegenüber werden bei IEC 61131-3 die Variablen mit dem Schlüsselwort »*AT*« auf Ein- oder Ausgangsadressen gelegt. Neuere Versionen von CoDeSys im Verbund mit aktuellen Targets der Hersteller erlauben eine komfortable Hardwarekonfiguration ähnlich der von Simatic/Step7. Beim Anschalten von Busklemmen kann nun für jeden Ein- oder Ausgang ein Bezeichner vorgegeben werden, der als globale Variable wirkt. Dies ist der symbolischen Bezeichnung der Ein- und Ausgänge in Step7 sehr ähnlich.

### STEP-7-PROGRAMM

**Byteverarbeitung Fertigungszelle**

| | | | |
|---|---|---|---|
| CAL posFl_0 (CLK:= S0)<br>LD posFl_0.Q<br>JMPC M1 | | (*Flankenauswertung S0*) | U »S0«<br>FP »Flankenmerker_0«<br>SPB M1 |
| CAL posFl_1 (CLK:= S1)<br>LD posFl_1.Q<br>JMPC M2<br>RET | | (*Flankenauswertung S1*) | U »S1«<br>FP »Flankenmerker_1«<br>SPB M2<br>BEA |
| M1: | LD Lichtgitter<br>EQ 16# 08<br>JMPC A | (*Abfrage Lichtgitter nach<br>(*Typ A*) | M1: L »Lichtgitter«<br>L B#16#08<br>==I<br>SPB A |
| | LD Lichtgitter<br>EQ 16 # 09<br>JMPC B | (*Abfrage Lichtgitter nach<br>(*Typ B*) | L »Lichtgitter«<br>L B#16#09<br>==I<br>SPB B |
| | LD Lichtgitter<br>EQ 16#04<br>JMPC C<br>RET | (*Abfrage Lichtgitter nach<br>(*Typ C*) | L »Lichtgitter«<br>B#16#04<br>==I<br>SPB C<br>BEA |
| A: | LD 16#36<br>ST Aktoren<br>RET | (*Eintrag der Ausgangsbits<br>(*für Typ A*) | A: L B#16#36<br>T »Aktoren«<br>BEA |
| B: | LD 16#24<br>ST Aktoren<br>RET | (*Eintrag der Ausgangsbits für<br>(*für Typ B*) | B: L B#16#24<br>T »Aktoren«<br>BEA |
| C: | LD 16#3D<br>ST Aktoren<br>RET | (*Eintrag der Ausgangsbits<br>(*für Typ C*) | C: L B#16#3D<br>T »Aktoren«<br>BEA |
| M2: | LD 16#00<br>ST Aktoren<br>RE | (*Ausschalten aller Ausgänge*)<br>(*bei Betätigung von S1*) | M2: L B#16#0<br>T »Aktoren«<br>BE |

| Variablen CoDeSys | | Variablen Step7 | |
|---|---|---|---|
| lokal | global | lokal | global |
| nur im aktuellen Baustein gültig | im gesamten Programm gültig | nur im aktuellen Baustein gültig | im gesamten Programm gültig |
| Adressen werden vom System automatisch verwaltet 1) | | Adressen werden vom System automatisch verwaltet | Adressen müssen vom Programmierer verwaltet werden! |
| deklariert durch Bezeichner und Datentyp | | deklariert durch Bezeichner und Datentyp | Absolute oder symbolische Syntax |
| **Nutzung:**<br>nach den Regeln der Variablendeklaration ohne Einschränkungen<br>Beispiel global oder lokal:Teilezahl:INT;<br>1) Nicht empfehlenswerter Sonderfall: Durch Bezeichnung als Merker können Variablen vom Programmierer addresiert werden. Beispiel:<br>Teilezahl AT %MW10:WORD; | | **Nutzung:**<br>- als Temporäre Variablen im L-Stack<br>- als Statische Variablen im Instanzdatenbaustein eines FB<br>Beispiel:<br>(in der Deklarationstabelle)<br>#Teilezahl (INT) | **Nutzung:**<br>- im Merkerbereich<br>- als Elemente von Datenbausteinen<br>- als Timer und Zaehler<br>Beispiele:<br>"Teilezahl" oder MW10<br>"Status".LED1 oder DB2.DBX4.1<br>"Zeitglied1" oder T12 |

**Bild 114: Datenhaltung in den Programmiersystemen CoDeSys und Step7**

*(Fortsetzung folgt)*