IN DIESEM KAPITEL

- Erfahren Sie, was Variablen sind und wie man diese nutzt
- Lernen Sie, welche verschiedenen Typen von Variablen es gibt
- Bekommen Sie vermittelt, wie Zahlenwerte dargestellt werden können
- Erläutere ich Ihnen, wie Zahlen im Speicher abgelegt werden
- Vermittele ich Ihnen, welche Konstrukte es beim Programmieren gibt
- Zeige ich Ihnen verschiedene Objekte, die Ihnen beim Programmieren begegnen können

Kapitel 1
Das 1×1 des Bitverbiegens

Einführung

Wenn Sie noch nie ein Programm erstellt haben, dann vermittelt Ihnen dieses Kapitel eine gute Basis, um selbst erste kleine Programme zu schreiben, auf die Sie künftig aufbauen können. Selbst wenn Sie sich anfangs an der einen oder anderen Stelle schwertun sollten: Bleiben Sie dran! Aus persönlicher Erfahrung sage ich Ihnen: Es lohnt sich!

Bevor es richtig mit diesem Kapitel losgeht, ein Hinweis: Zur Erstellung eines SPS-Programms stehen Ihnen verschiedene Programmiersprachen zur Verfügung, von denen Sie eine oder auch mehrere in Ihrem Programm nutzen können. Näheres zu dem Thema erfahren Sie in Kapitel 3 »Die babylonische Sprachverwirrung«. Hier in diesem Kapitel nutze ich dagegen nur eine textbasierte Programmiersprache, und zwar *ST* oder in Langform *Strukturierter Text* genannt. Im Englischen lautet die Abkürzung ebenfalls *ST* und steht dort entsprechend für *structured text*. Bei Siemens gibt es auch eine textbasierte Programmiersprache, diese wird mit SCL abgekürzt, was in der Langform für Structured Control Language steht, eine deutsche Variante des Namens gibt es bei Siemens nicht.

Was sind Variablen und wie nutzt man sie?

Die Hauptaufgabe einer SPS liegt darin, Daten zu empfangen, die Zustände einer Anlage einzulesen, diese zu verarbeiten und anschließend entsprechend der Verarbeitung Ausgänge zu setzen oder Daten zu verschicken. Bei den Zuständen können dies *binäre Werte* sein wie beispielsweise der Zustand einer Lichtschranke, die genau zwei (daher »binär«) Zustände, nämlich *auf* (offen bzw. `1`) oder *zu* (geschlossen bzw. `0`) haben kann, oder *Messwerte* wie zum Beispiel eine Temperatur. Diese Werte müssen für die weitere Bearbeitung nach dem Einlesen erst mal abgelegt werden. Dies erfolgt in sogenannten *Variablen*.

Eine Variable kann man als einzelnen Behälter betrachten, der einen Namen (den Variablennamen), eine Adresse im Speicher der SPS und, je nach Typ, eine unterschiedliche Größe, sprich einen unterschiedlich großen Speicherbedarf, hat. Eine Ausnahme davon bilden die sogenannten *Pointer*, die immer dieselbe Größe haben. Doch dazu später in diesem Kapitel mehr.

Wie Variablen deklariert werden

Bevor ich mit der eigentlichen Erklärung beginne, hier ein wichtiger Hinweis: Dieses Buch ist herstellerübergreifend angelegt, allerdings werden im Laufe des Buchs einige Hersteller explizit erwähnt, zum Beispiel Siemens mit der Entwicklungsumgebung TIA. Neben Siemens werden weitere Hersteller erwähnt, die alle etwas gemeinsam haben, deren Systeme basieren auf der Software CODESYS.

Gerade wurden die Variablen als Behälter beschrieben, in dem Daten aufbewahrt werden können. Doch bevor eine Variable als Behälter dienen kann, muss sie erst einmal erstellt werden.

Bei SPS-Programmen ist eine Variable nicht einfach so vorhanden und kann genutzt werden. Sie muss zunächst angelegt werden, was als *Deklaration* bezeichnet wird.

In Abbildung 1.1 ist die Deklaration mehrerer Variablen bei einer CODESYS-basierten Steuerung zu sehen und in Abbildung 1.2 dasselbe für TIA von Siemens.

FB_ST_01

```
1  FUNCTION_BLOCK PUBLIC FB_ST_01
2  VAR_INPUT
3  END_VAR
4  VAR_OUTPUT
5  END_VAR
6  VAR
7      xTaste01    : BOOL;
8      bFlags01    : BYTE;
9      uiCounter01 : UINT;
10     sText01     : STRING(10) :='Test123';
11 END_VAR
```

Abbildung 1.1: Variablendeklaration bei CODESYS-basierten Steuerungen

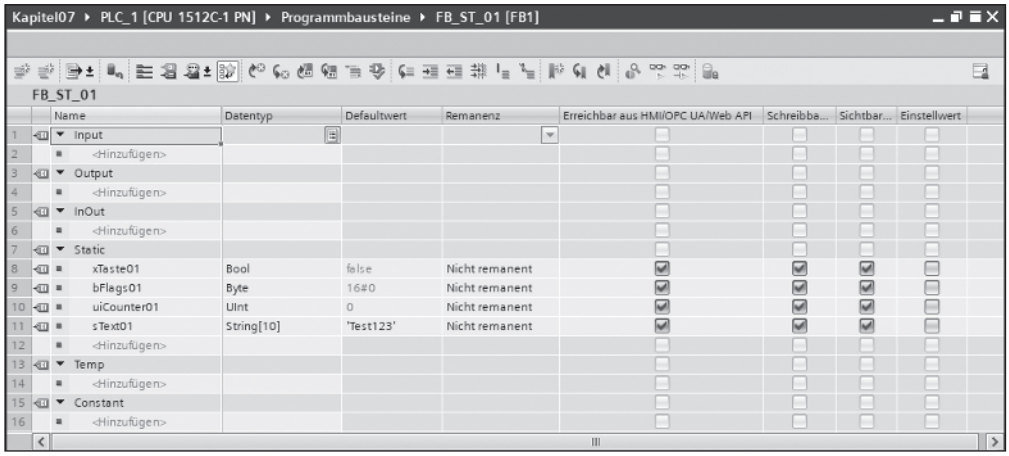


Abbildung 1.2: Variablendeklaration in Siemens TIA

In beiden Abbildungen werden die gleichen Variablen deklariert. Bei CODESYS-basierten Steuerungen und auch bei Siemens in TIA kann eine Deklaration in Text- oder Tabellenform erfolgen. Bei CODESYS nutze ich die Textform, bei Siemens TIA die tabellarische Form.

Bei CODESYS-basierten Steuerungen ist eine Deklaration wie folgt aufgebaut:

1. Zunächst wird der Name der zu deklarierenden Variablen angegeben und
2. danach folgt der Typ.
3. Bei Bedarf kann die Variable anschließend noch einen Startwert bekommen, den diese bei jedem Start der SPS erhält.

Bei Siemens in TIA wird in der ersten Spalte der Tabelle zunächst

1. der Name der Variablen vergeben, in der nächsten Spalte
2. dann der Typ und danach
3. der Startwert.

Welche Variablentypen es gibt und welche Eigenschaften sie haben, erfahren Sie im Verlauf dieses Kapitels im Abschnitt »Vieles zum Thema Variablentypen«.

In beiden Beispielen wurden jeweils vier Variablen deklariert. Die Variable `xTaste01` vom Typ BOOL, die Variable `bFlags01` vom Typ BYTE, die Variable `uiCounter01` vom Typ UINT und die Variable `sText01` vom Typ STRING. Die letzte deklarierte Variable wurde mit dem Startwert »Test123« initialisiert.

Variablen Werte zuweisen

Neben der Zuweisung eines Startwerts bei der Deklaration einer Variablen können Sie einer Variablen natürlich auch im Programm einen Wert zuweisen, sonst wäre das Ganze ja auch relativ sinnlos. Schließlich steht der Begriff »Variable« ja für veränderlich …

Sowohl bei CODESYS-basierten Steuerungen in ST als auch bei Siemens TIA in SCL erfolgt die Zuweisung eines Werts zu einer Variablen mit dem sogenannten Zuweisungsoperator :=. Sie können einer Variablen auch den aktuellen Wert einer anderen Variablen zuweisen.

Im Folgenden sind hier mal ein paar Beispiele aufgeführt, wie die Zuweisung eines Werts zu einer Variablen erfolgen kann.

```
xBoolVar01    := TRUE;
iIntVar01     := -512;
iIntVar02     := iIntVar03;
```

Die Deklaration der Variablen bei CODESYS-basierten Steuerungen erfolgte zwischen den Schlüsselwörtern VAR und END_VAR, bei Siemens im Bereich STATIC. In beiden Fällen handelt es sich um sogenannte lokale Variablen. Diese sollten auch nur lokal verwendet werden, also nur innerhalb des Objekts, zum Beispiel eines Funktionsbausteins, in dem sie deklariert wurden. Näheres zum Thema Objekte erfahren Sie in diesem Kapitel im Abschnitt »Weitere Objekte, beispielsweise Funktionen«.

Bei CODESYS-basierten Steuerungen ist von außerhalb nur ein lesender, aber kein schreibender Zugriff möglich, man kann also den Wert zwar woanders verarbeiten, aber nicht von woanders ändern. Ein Versuch, ein Programm zu übersetzen, das einen schreibenden Zugriff enthält, führt zu einer Fehlermeldung. Bei Siemens ist sowohl der lesende als auch der schreibende Zugriff theoretisch möglich, beides sollten Sie aber nicht tun.

Was sind globale Variablen und globale Datenbausteine

Soll von verschiedenen Objekten auf bestimmte Variablen zugegriffen werden, sollten Sie sogenannte *globale Variablen* verwenden. Diese Variablen stehen in Listen, in denen eine oder mehrere Variablen hinzugefügt werden können.

Bei CODESYS-basierten Steuerungen heißt diese Liste »Globale Variablenliste«. Es können mehrere Listen vorhanden sein, wobei jede Liste einen individuellen Namen hat.

In Abbildung 1.3 ist eine solche Liste zu sehen.

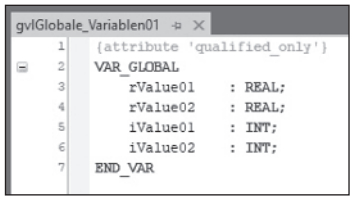


Abbildung 1.3: Globale Variablenliste

Die globale Variablenliste in Abbildung 1.3 hat den Namen GVLGLOBALE_VARIABLEN01 und enthält vier Variablen. Der Befehl in Zeile 1 ist ein sogenanntes *Pragma*. Es gibt Pragmas für verschiedene Aufgaben. Es gibt zum Beispiel Pragmas, mit denen erreicht werden kann, dass nur bestimmte Teile eines Programms übersetzt werden. Das Pragma in diesem

Beispiel bewirkt, dass neben dem Variablennamen auch der Name der Liste angegeben werden muss, wenn Sie auf eine Variable aus dieser Liste zugreifen möchten. Um zum Beispiel der Variablen `rValue01` den Wert 5.0 zuzuweisen, müssten Sie in Ihrem Programm die folgende Zeile hinzufügen:

```
gvlGlobale_Variablen01.rValue01 := 5.0;
```

Wäre das Pragma nicht vorhanden, würde der folgende Code ausreichen:

```
rValue01 := 5.0;
```

Bei Siemens in TIA wird für diese Liste ein sogenannter *globaler Datenbaustein* verwendet. In Abbildung 1.4 ist ein solcher Datenbaustein zu sehen, mit denselben Variablen wie in der globalen Variablenliste in Abbildung 1.3 für CODESYS-basierte Steuerungen.

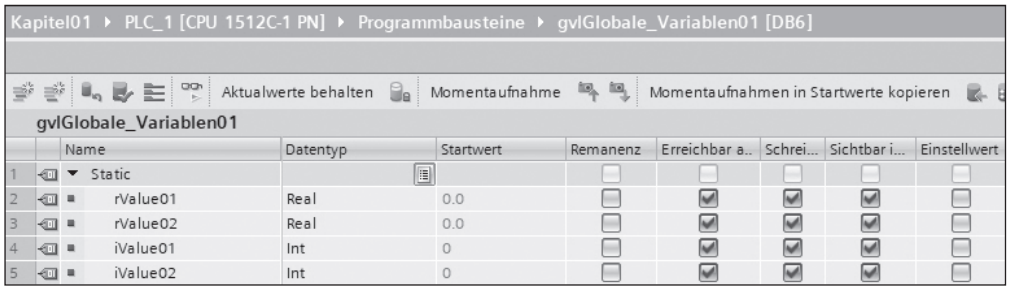


Abbildung 1.4: Globaler Datenbaustein

Um auch hier der Variablen `rValue01` wieder den Wert 5.0 zuzuweisen, müssten Sie den folgenden Code Ihrem Programm hinzufügen:

```
"gvlGlobale_Variablen01".rValue01 := 5.0;
```

Datenbausteine werden auch noch in anderen Situationen verwendet. Wo, das erkläre ich weiter unten in diesem Kapitel bei der Erklärung des Objekts Funktionsbaustein.

Bei der Verwendung von globalen Variablen sollten Sie den Spruch »So viel wie nötig, so wenig wie möglich« beherzigen.

Bei der Verwendung von globalen Variablen kann man sich leicht verzetteln, speziell das Schreiben globaler Variablen an verschiedenen Stellen sollte unbedingt vermieden werden, da im Fehlerfall eine Suche nach der Ursache ziemlich schwierig werden könnte.

Vieles zum Thema Variablentypen

In der folgenden Liste sind verschiedene Variablentypen mit ihrem Namen, dem – soweit vorhanden – in der Entwicklungsumgebung verwendeten Kürzel, mit ihrem Speicherbedarf und, soweit sinnvoll, ihrem Wertebereich aufgeführt. Ein BYTE beinhaltet dabei 8 Bits, also acht Speicherzellen, die entweder 0 oder 1 beziehungsweise FALSE oder TRUE sein können.

Variablentyp	Größe	Wertebereich
BOOL	? Byte	FALSE/TRUE, 0/1
BYTE	1 Byte	
WORD	2 Byte	
DWORD	4 Byte	
Unsigned Short Integer USINT	1 Byte	0 bis 255
Unsigned Integer UINT	2 Byte	0 bis 65535
Short Integer SINT	1 Byte	–128 bis127
Integer INT	2 Byte	–32768 bis 32767
Real REAL	4 Byte	-3.402823×10^{38} bis 3.402823×10^{38}

Tabelle 1.1: Übersicht über Variablentypen

Sie werden sich vielleicht schon über das Fragezeichen bei den BOOL-Variablen gewundert haben. Ich werde dies im Folgenden erklären, möchte aber schon hier die Warnung aussprechen, dass Ihnen eventuell nach dieser Erklärung der Kopf schwirren könnte und Sie aus der Sache leicht verwirrt rauskommen.

Boolesche Variablen können nur zwei Zustände annehmen, die durch 0 und 1 ausgedrückt werden, und würden entsprechend in ein Bit passen. Dies wird aber meist von den Herstellern nicht genutzt. In den meisten Fällen wird eine boolesche Variable in einem Byte abgelegt, wobei von diesem Byte dann nur das unterste Bit genutzt werden darf. Es würde zu einer Fehlermeldung kommen, wenn man versucht, die anderen Bits des Bytes zu ändern.

Warum eine Variable vom Typ BOOL nicht in ein Bit abgelegt wird, liegt darin begründet, dass der Zugriff auf ein einzelnes Bit an einer beliebigen Stelle innerhalb eines Bytes deutlich aufwendiger ist und somit viel mehr Zeit in Anspruch nimmt. Es gibt aber Steuerungen, bei denen es auch die Möglichkeit gibt, eine boolesche Variable in einem einzelnen Bit abzulegen, sodass 8 boolesche Variablen lediglich 1 Byte im Speicher belegen, anstatt 8 Byte. Dieser Umstand ist ein Grund für das obige Fragezeichen, ein weiterer folgt gleich.

Leider gilt hier der Spruch »Keine Regel ohne Ausnahme«. Es gibt Hersteller, bei denen mehrere Bits in ein Byte abgelegt werden und das jeweilige Bit dann ausmaskiert wird.

Quizfrage: Um wie viel Bytes vergrößert sich ein Programm, wenn man eine Variable vom Typ BOOL hinzufügt, wenn ein BOOL einzeln in einem Byte abgelegt wird?

Sie werden sich jetzt mit Sicherheit fragen, was diese Frage soll – natürlich 1 Byte! Schließlich habe ich ja gerade geschrieben, dass Variablen vom Typ BOOL, bis auf wenige Ausnahmen

ein Byte groß sind. Nun, das ist richtig. Dennoch kann es vorkommen, dass ein Programm nach dem Hinzufügen eines BOOL um mehr als ein Byte größer wird. Aber wie kann das sein?

Der Grund hierfür liegt im sogenannten Alignment. Schon wieder so ein neudeutscher Begriff, aber was bedeutet er? Das Wort Alignment bedeutet übersetzt *Ausrichtung*, und genau darum geht es, nämlich um die Art, wie Daten, zum Beispiel Variablen, im Speicher ausgerichtet werden. Dies erfolgt nämlich nicht immer, wie man es eigentlich erwarten würde, lückenlos, sondern es werden, je nach Aneinanderreihung von Variablen, Füllbytes hinzugefügt. Im folgenden Kasten finden Sie ein Beispiel dazu.

Ein BOOL belegt hier ein Byte, wenn nun also zwei BOOL im Speicher hintereinander abgelegt werden, ergibt das einen Speicherverbrauch von zwei Bytes. An dieser Stelle sei nochmals angemerkt, dass es auch Hersteller gibt, bei denen ein BOOL tatsächlich nur ein Bit einnimmt. Ein WORD belegt zwei Bytes. Wenn nun zunächst ein BOOL und anschließend ein WORD im Speicher abgelegt werden, müsste dies einen Speicherverbrauch von 3 Bytes ergeben. Das tut es aber, wie erwähnt, nicht. Die Startadresse, an der ein WORD abgelegt werden darf, muss – oder besser *sollte* – gerade sein. Daher wird nach dem BOOL ein Füllbyte eingefügt, im englischen Pad-Byte genannt. Solche Daten werden auch Fülldaten genannt. Bei einem DWORD, das 4 Bytes belegt, ist die Sache übrigens noch etwas umfangreicher. Ein DWORD sollte zwar auch an einer geraden Speicheradresse beginnen, gleichzeitig muss diese dann aber auch noch ein Vielfaches von vier sein, also zum Beispiel 0, 4, 16, 20 und so weiter.

An dieser Stelle kann man jetzt den Titel eines Lieds des Musikers Herbert Grönemeyer zitieren, nämlich »Was soll das?« Es gibt hierfür nur einen Grund: Geschwindigkeit. Damit zum Beispiel ein WORD innerhalb eines Takts eingelesen werden kann, muss es an bestimmten Adressen im Speicher starten, eben an geraden Adressen. Da heutzutage Speicher meist reichlich zur Verfügung steht, ist diese Verschwendung von Speicher nicht weiter problematisch. Es kann aber Situationen geben, wo diese Lücken stören würden. Es gibt deshalb Möglichkeiten, die Lückenbildung in solchen Situationen auszuschalten.

Warum gibt es für manche Variablentypen keine Bereiche?

Falls Sie schon Programmierkenntnisse besitzen oder bereits vorab etwas von den Typen BYTE und WORD gehört haben, dann werden Sie sich oben vielleicht gewundert haben, warum ich bei diesen Typen keine Wertebereiche angegeben habe. Wie ich bereits schrieb, erfolgt die Angabe eines Wertebereichs nur da, wo es sinnvoll ist. Bei den Typen BYTE und WORD ergibt die Angabe eines Wertebereichs jedoch schlicht keinen Sinn. Ein Wertebereich beschreibt einen Zahlenbereich, den eine Variable annehmen kann, die für die Speicherung einer Zahl gedacht ist. Es gibt Variablentypen für Zahlen, zum Beispiel REAL oder INTEGER, und welche für andere Dinge, beispielsweise zum Ablegen von Buchstaben. Zu letzterem Beispiel komme ich später noch.

Variablen vom Typ BYTE oder WORD sind keine Typen für Zahlen. Vermeiden Sie die Nutzung dieser Variablentypen für Zahlenwerte.

Aber wenn Sie nicht zum Speichern von Zahlen oder Buchstaben genutzt werden, wofür sind sie denn dann da? Nun, ganz einfach, sie dienen zur Ablage von Nullen und Einsen, sprich Bits, weswegen sie auch als *Bit-Datentypen* bezeichnet werden. Die Typen BYTE und WORD können als Containerformate für eine bestimmte Anzahl von Bits betrachtet werden. Im Falle eines Bytes zum Beispiel zur Ablage von 8 Bits und im Falle eines WORD zur Ablage von 16 Bits.

Verschiedene Darstellungsarten von Werten

Für diese Variablentypen, also BYTE und WORD, gibt es eigentlich nur eine Darstellungsart, nämlich die Binärdarstellung, also eine Anreihung von 0 und 1. Dies entspricht der Art, wie ein Computer und somit auch eine SPS die Daten in einzelnen Speicherzellen ablegt. Eine 1 bedeutet dabei, dass an der zu dieser Speicherzelle zugehörigen Schaltung eine Spannung anliegt, während bei einer 0 keine Spannung anliegt.

Die Bits werden von niederwertigsten zum höchstwertigen von rechts nach links durchgezählt, wobei in den meisten Fällen bei 0 mit der Zählung begonnen wird.

Ist in einem Byte das 0., das 3., 5. und 6. Bit auf 1 und der Rest auf 0, dann sieht das Byte in binärer Schreibweise so aus:

01101001

Ein WORD, das aus 16 Bits besteht, bei dem das 1., 6., 7., 9., 12. und 15. Bit gesetzt ist, sieht entsprechend so aus:

1001001011000010

Das Ganze ist zugegebenermaßen recht schwer zu lesen. Daher bin ich zu einem Kompromiss bereit. Neben dieser Darstellung kann ein BYTE oder WORD auch in *hexadezimaler Schreibweise* dargestellt werden. Eine Darstellung in dezimaler Schreibweise, also in der Art, wie Sie und ich im täglichen Leben Zahlen schreiben, ist nicht wirklich sinnvoll, was ich weiter unten in diesem Kapitel noch beweisen werde. Aber was ist die hexadezimale Schreibweise jetzt wieder für ein neomodisches Zeug? Ich werde versuchen, sie im Folgenden möglichst verständlich zu erläutern.

Bleiben wir zunächst bei den Zahlen und dem Zahlensystem, mit dem wir oft täglich zu tun haben. Die Rede ist von den römischen Zahlen, beispielsweise 1, 5, 8 oder 9 und dem Dezimalsystem als Zahlensystem. Bei diesem Zahlensystem gibt es pro Stelle Ziffern von 0 bis 9. Je weiter rechts eine Stelle steht, desto niederwertiger ist sie. Die Ziffer an der zweiten Stelle von rechts betrachtet hat also einen höheren Wert als die Ziffer ganz rechts. Der Begriff »Dezimal« leitet sich von dem lateinischen Wort *decimus* ab, was für »der Zehnte« steht. Dies beschreibt, dass das Zahlensystem die Basis 10 hat und somit jede Stelle weiter links um den Faktor 10 größer ist.

Beim Hexadezimalsystem ist das ähnlich. Wie Sie vielleicht schon erkannt haben, enthält auch dieser Begriff als einen Bestandteil das Wort »dezimal«. Heißt das also, das auch dieses Zahlenformat etwas mit 10 zu tun hat? Naja fast, denn es kommt auch noch das griechische Wort »hexa« vor, das 6 bedeutet. Bei diesem Zahlenformat ist die Basis nicht 10, sondern 16. Aber wie soll das denn darstellbar sein? Nun, mit den Ihnen bis jetzt bekannten Ziffern von 0 bis 9 auf jeden Fall nicht. Da benötigen Sie noch tatkräftige Unterstützung. Diese erhalten Sie in Form der Buchstaben A bis F.

Um hexadezimale Zahlen von anderen unterscheiden zu können, gibt es verschiedene Möglichkeiten, sie als hexadezimale zu kennzeichnen.

Hier einmal verschiedene Kennzeichnungsformen für hexadezimale Zahlen:

- ✔ $92C2_{Hex}$
- ✔ $92C2_{16}$
- ✔ 0x92C2

Wie soll nun dieses Zahlenformat bitte dafür Sorge tragen, dass Sie eventuell schneller oder besser erkennen können, welche Bits in einem WORD gesetzt sind? Um das zu verstehen, muss ich jetzt auch noch das *Binärsystem* zerpflücken. Das Binärsystem wird auch *Dualsystem* genannt, was sich vom lateinischen Wort »dualis« ableitet, das »zwei enthaltend« bedeutet. Bei diesem Zahlensystem ist die Basis 2. Im Gegensatz zum Dezimalsystem, wo jede Stelle weiter links um den Faktor 10 größer ist, oder um den Faktor 16 bei hexadezimalen Zahlen, ist beim Binärsystem jede Stelle weiter links um den Faktor 2 größer.

Bei Bits wird meist mit der Zählung bei 0 begonnen.

Um Ihnen das Leben als Einsteiger nicht noch komplizierter zu machen, als ich es Ihnen mit dieser Beschreibung schon mache, werde ich zur Erleichterung im folgenden Teil bei Bits mit der Zählung bei 1 beginnen.

Der Umstand, dass jede Stelle einer Binärzahl nur zwei Zustände haben kann, und die Tatsache, dass das Hexadezimalsystem pro Stelle 16 mögliche Zustände haben kann, sorgen dafür, dass diese beiden Zahlenformate wie geschaffen füreinander sind. Zum einen ist 16 ein Vielfaches von 2, zum anderen kommt man auf die 16, indem man die Zahl 2 mehrfach mit sich selbst multipliziert. Um genau zu sein, exakt vier Mal, denn $2 \times 2 \times 2 \times 2$ ist 16. Daraus ergibt sich, dass in einer Stelle einer hexadezimalen Zahl vier Stellen einer Dualzahl untergebracht werden können.

Wäre bei einer Variablen, die ein Byte an Speicher belegt, nur das unterste Bit gesetzt, würde diese in binärer Schreibweise als 00000001 dargestellt und zum Beispiel als 0x01in hexadezimaler Schreibweise. Wäre stattdessen nur das vierte Bit gesetzt, würde die Variable als 00001000 in binärer Schreibweise und als 0x08 in hexadezimaler Schreibweise dargestellt werden. Sind bei der Variablen nun das zweite und das vierte Bit gesetzt, also 00001010, würde die Variable in hexadezimaler Schreibweise mit 0x0A dargestellt werden. Warum hier jetzt der Wert A bei der niederwertigsten Stelle herauskommt, werde ich kurz erläutern.

Bei Dezimalzahlen hat jede Stelle eine bestimmte Stellenwertigkeit. Die Stelle ganz rechts beinhaltet die Einer, die Stelle links daneben die Zehner, danach die Hunderter und so weiter. Bei der Binärdarstellung übertragen auf dezimale Werte beinhaltet die ganz rechte Stelle die 1, die Stelle links daneben die 2, die dann daneben liegende die 4, dann die 8. Bei der Binärzahl 00001010 sind ja, wie erwähnt, das zweite und das vierte Bit gesetzt. Das zweite Bit hat die Stellenwertigkeit 2 und das Vierte die Stellenwertigkeit 8, das ergibt 10 in dezimaler Schreibweise.

Beim Hexadezimalsystem hat jede Stelle 16 mögliche Zustände, zunächst von 0 bis 9, wie beim Dezimalsystem auch, und dann geht es weiter mit A bis F.

Zäumen wir das Pferd einmal von hinten auf – oder von vorn, je nach Sichtweise. Nehmen wir zunächst die 8. Sie wird als Hexadezimalzahl auch als 8 dargestellt, nun zählen wir die 2 in einzelnen Schritten dazu.

8 + 1 auch in hexadezimaler Schreibweise ergibt dies 9.

9 + 1 hier landen Sie nun beim ersten Buchstaben, dem A.

Ich möchte nach diesem Exkurs in die verschiedenen Zahlensysteme noch einmal auf das Beispiel mit der Binärzahl 1001001011000010 eingehen.

Wie erwähnt, kann eine Stelle einer hexadezimalen Zahl vier Stellen einer binären Zahl darstellen. Dem Umstand Rechnung tragend schreibe ich das Beispiel im Folgenden etwas anders.

1001 0010 1100 0010

Was die Sache schon etwas lesbarer macht, noch besser wird es aber als hexadezimale Zahl. Als solche sieht das Beispiel nun wie folgt aus.

9 2 C 2

Da das Thema leider nicht so einfach zu verdauen ist, werde ich kurz erläutern, wie ich auf diese Zahl komme. Dabei gehe ich von rechts nach links vor.

Bei der Stellenwertigkeit wird bei jeder Vierergruppe der Binärzahl wieder erneut von vorne mit der Stellenwertigkeit – also der 1 – begonnen.

Die unterste Stelle der hexadezimalen Zahl ist die zwei. Wie sie zustande kommt, ist, denke ich, einmal relativ einfach. Der zu dieser Stelle gehörende Teil der Binärzahl lautet 0010. Es ist nur das zweite Bit gesetzt, das die Stellenwertigkeit 2 hat.

Der zur nächsten Stelle der hexadezimalen Zahl gehörende Teil der Binärzahl lautet 1100. Hier sind das dritte und das vierte Bit gesetzt. Das dritte Bit hat die Stellenwertigkeit 4 und das vierte die Stellenwertigkeit 8. Dies ergibt, wieder einzeln weiter gezählt, 9, A, B, C.

Die dritte Stelle der hexadezimalen Zahl bedarf keiner Erklärung, hier gilt das zur ersten Stelle geschriebene.

Bei der vierten Stelle ist der zugehörige Teil der Binärzahl 1001. Es sind das erste Bit mit der Stellenwertigkeit 1 und das vierte Bit mit der Stellenwertigkeit 8 gesetzt, macht zusammen 9.

Ich habe volles Verständnis dafür, falls Ihnen eventuell Zweifel kommen, dass diese Darstellung besser lesbar sein soll. Hier kann ich Sie nur bitten, meiner Erfahrung zu vertrauen und mir zu glauben, dass dem tatsächlich so ist und Sie mit fortschreitendem Wissen und wachsender Erfahrung an den Punkt kommen werden, an dem Sie das genauso sehen.

Noch ein Nachschlag zum Thema hexadezimale Zahlen.

Zahlen in hexadezimaler Schreibweise werden immer einzeln Ziffer für Ziffer von links nach rechts genannt ohne eine Stellenwertigkeit. Beim dezimalen Zahlenformat würde man die Zahl 123 als Wort »Einhundertdreiundzwanzig« schreiben, als hexadezimalzahl nur als »Eins zwei drei«.

Bevor ich nun endlich den Nachweis antrete, dass die Darstellung unter anderem von Bytes und WORDs als Dezimalzahl nicht sinnvoll ist, muss ich etwas gestehen. Ich habe Ihnen bei der Auflistung der Stellenwertigkeit etwas unterschlagen. Bei der Stellenwertigkeit habe ich nur die Wertigkeit von vier Bits erwähnt, dies ist im Zusammenhang mit hexadezimalen Zahlen auch voll und ganz sinnvoll, da ja vier Bits immer eine Stelle einer hexadezimalen Zahl abbilden und das Ganze bei der nächsten Stelle wieder von vorne losgeht.

Das Dezimalsystem hat als Basis die 10, die wie beim hexadezimalen System auch ein Vielfaches der Basis des Binärsystems, der Zahl 2, ist. Der Unterschied: Auf die 10 kommt man nicht durch mehrfaches Multiplizieren mit 2. Beim hexadezimalen System ist der höchste Wert einer Stelle F. Das bedeutet, dass alle vier Bits der zugehörigen Binärzahl gesetzt sind. Die Hexadezimalzahl 0x0F wäre als Binärzahl dargestellt also 0000 1111. Addiert man zu der Hexadezimalzahl nun 1 hinzu, kommt es zu einem Überlauf, da die erste Stelle ja schon ihren höchstmöglichen Wert erreicht hatte. Die Zahl in hexadezimaler Schreibweise wäre nun 0x10 und als Binärzahl ist es jetzt 0001 0000. Womit das Spiel der Stellenwertigkeit beim nächsten Viererblock wieder vorne beginnt.

Der höchstmögliche Wert einer Stelle einer Dezimalzahl beträgt 9.

Betrachtet man nun die Zahl 9, so würde diese als Binärzahl mit 8 Bits als 0000 1001 dargestellt werden. Addiert man zu dieser nun die Zahl 1 dazu, kommt es auch bei der Dezimalzahl zu einem Überlauf, und man erhält die Zahl 10 oder in Worten Zehn. Diese Zahl in binärer Schreibweise mit 8 Bits wäre dann 0000 1010. Sie sehen sicher sofort den Unterschied zum Überlauf bei einer hexadezimalen Zahl. Bei einer dezimalen Zahl kommt es sozusagen zu keiner Wiederholung bei der Stellenwertigkeit, was bedeutet, dass die obige Auflistung nach der 8 noch weitergeführt werden muss. Bei 8 Bit lautet die Stellenwertigkeit von rechts nach links betrachtet zunächst, 1, 2, 4, 8, so weit ist noch alles gleich. Die Stellenwertigkeit steigt bei einer Binärzahl mit jeder Stelle um den Faktor 2, somit geht es also mit 16 weiter und dann mit 32, 64 und schließlich 128. Bei einem WORD geht es dann bis zum sechzehnten Bit weiter, das die Stellenwertigkeit 32768 hat.

Mit diesen Kenntnissen ermitteln wir jetzt einmal zusammen, wie eine Zahl in binärer Schreibweise in dezimaler Schreibweise aussieht.

Die binäre Zahl 1001 0010 1100 0010 soll als dezimale Zahl dargestellt werden. Angesichts der Tatsache, dass eine Stelle der Dezimalzahl nicht einer bestimmten Anzahl von Stellen einer Binärzahl entspricht, könnten Sie die Binärzahl auch 1001001011000010 schreiben, aber gruppiert wird die Sache doch übersichtlicher.

Das erste gesetzte Bit ist das zweite Bit mit einer Stellenwertigkeit von 2. Dann folgt das siebte mit 64, das achte mit 128, das zehnte mit 512, das dreizehnte mit 4096 und zum Schluss das sechzehnte mit 32768.

So, rechnen Sie mal zusammen, 2 + 64 + 128 + 512 + 4096 + 32768 ergibt 37570.

Sie werden mir sicher zustimmen, dass im Vergleich zur binären oder hexadezimalen Schreibweise hier nun wirklich nicht zu erkennen ist, welches Bit gesetzt ist, vom untersten Bit einmal abgesehen.

Setzt man bei dem Beispiel das zwölfte Bit auf 1, käme als Dezimalzahl 39618 heraus. Hier müssten Sie jetzt erst groß nachrechnen, um herauszufinden, welches Bit geändert wurde. In hexadezimaler Schreibweise würde aus der 0x92C2 nun eine 0x9AC2 werden, und Sie könnten leicht sehen, dass sich an der dritten Stelle des Werts in hexadezimaler Schreibweise etwas geändert hat. Vorher hatte die Stelle den Wert 2 und jetzt den Wert A. Wenn Sie etwas mehr Erfahrung im Umgang mit hexadezimalen Zahlen haben, werden Sie auch sehen, dass die Änderung 8 beträgt (A – 2 = 8 oder in dezimaler Schreibweise 10 – 2 = 8).

Ich hoffe, ich habe Sie mit dieser Ausführung nicht zu sehr verwirrt und Ihnen qualmt der Kopf, wenn überhaupt, nur leicht. Aber dieses Thema ist sehr wichtig.

Einen Technikertipp habe ich zum Thema Zahlensysteme jedoch noch.

Im Internet kommt es immer wieder zu einer bestimmten Frage. Jemand hat eine Variable, deren Wert als Dezimalzahl 18_{Dez} angezeigt wird, und sucht nun einen Konvertierungsbefehl, um diese in die hexadezimale Zahl 12_{Hex} umzuwandeln.

Suchen Sie einmal in der Hilfe Ihrer Entwicklungsumgebung nach einem solchen Konvertierungsbefehl. Was sagen Sie? Sie haben nichts oder aber nicht ganz das Richtige gefunden? Das ist auch völlig in Ordnung, denn einen solchen Befehl gibt es nicht. Aber warum ist das so?

Die Antwort ist relativ einfach, in der Abschnittsüberschrift schon enthalten und lässt sich auch aufgrund bestimmter von mir in diesem Abschnitt verwendeter Wörter ermitteln.

In der Abschnittsüberschrift ist von *Darstellungsarten* die Rede, und genau das sind hexadezimale oder dezimale Zahlen. Es sind verschiedene Schreibweisen desselben Betrags. Mal angenommen, an dem Speicherplatz einer 8 Bit großen Variablen in Ihrer SPS steht das Folgende:

0001 0010

Kommt Ihnen das bekannt vor? Richtig, das entspricht der Zahl 18, aber auch der Zahl 12. »Ja, was denn jetzt?«, werden Sie sich vielleicht fragen. Na ja, beides eben. Der Zahl 18 in dezimaler Schreibweise und der Zahl 12 in hexadezimaler Schreibweise.

Haben Sie schon mal ein Ziffernblatt einer Uhr gesehen, auf der so komische Sachen wie ein X oder V standen? Das sind auch Zahlen, hier in lateinischer Schreibweise. In dieser Schreibweise würde die obige Zahl als XVIII dargestellt werden.

Es handelt sich dabei um eine andere Art, wie eine Zahl dargestellt wird. Aus diesem Grunde kann es auch keinen Konvertierungsbefehl geben, sondern lediglich eine Einstellung in Ihrer Entwicklungsumgebung, wie Zahlen dargestellt werden sollen.

Mancher von Ihnen hat vielleicht dennoch Konvertierungsbefehle gefunden und fragt sich jetzt womöglich, was der Autor hier für einen Blödsinn erzählt, wo es doch Konvertierungsbefehle gibt. Derartige Befehle gibt es tatsächlich, aber ist das kein Widerspruch. Und zwar deshalb …

Es kann vorkommen, dass Zahlenwerte einer Variablen in Textform dargestellt werden sollen und das in unterschiedlichen Schreibweisen. Und dafür gibt es dann entsprechende Befehle.

Wie kann man gerade und ungerade Werte erkennen

Quizfrage: Ist der Wert der Binärzahl 0001 0010, die schon weiter oben verwendet wurde, gerade oder ungerade?

Richtig, der Wert der Binärzahl ist gerade, weil das unterste oder äußerst rechte Bit nicht gesetzt ist und es das einzige mit einer ungeraden Stellenwertigkeit ist, bei allen anderen Bits ist die Stellenwertigkeit gerade.

Dieses Bit wird übrigens auch als das niedrigstwertige Bit bezeichnet oder in Englisch als *least significant bit*. Abgekürzt wird die englische Bezeichnung mit LSB. Dagegen wird das äußerst linke Bit als das höchstwertige Bit bezeichnet, *most significant bit* wäre in diesem Fall die englische Bezeichnung und MSB die entsprechende Abkürzung.

Wie werden Variablen im Speicher abgelegt?

Ich hatte eben vom Abschluss des Themas geschrieben. Dies stimmt leider nur für den Teil mit den Zahlenformaten. Beim Thema Variablen gibt es noch einen wichtigen und leider auch nicht ganz so leicht verdaulichen Teil zu der Art, wie Variablen im Speicher abgelegt werden.

Im Laufe der letzten Jahre haben sich Computer und damit auch SPSen stetig weiterentwickelt. Die Prozessoren wurden schneller, der Arbeitsspeicher und der Massenspeicher wurden größer und der Funktionsumfang stieg. Eine Eigenschaft hat sich im Laufe der letzten

Jahre allerdings nicht geändert. Eine Speicherzelle, sei es nun der Schreib-/Lesespeicher, also das RAM, oder der Nur-Lese-Speicher, also das ROM, besteht auch heute noch aus acht einzelnen Bits und hat damit die Größe eines Bytes. Um ein WORD zu speichern, was 16 Bits benötigt, müssen also zwei Speicherzellen, sprich zwei Bytes, verwendet werden.

Bleiben wir erneut bei obigem Beispiel und dem WORD mit dem Wert 0x92C2. Wenn ich Ihnen nun sage, dass »0x92, 0xC2« und »0xC2, 0x92« ein und dasselbe sind, werden Sie sich eventuell fragen, was ich geraucht habe … aber das ist tatsächlich so.

Es gibt verschiedene Hersteller von Prozessoren. Zwei führende Vertreter aus diesem Bereich sind, beziehungsweise waren, die Firmen Intel und Motorola. »Waren« bezieht sich dabei auf die Firma Motorola, da diese den Bereich der Prozessorentwicklung und -fertigung in eine eigenständige Firma ausgegliedert hatte, die es zum Zeitpunkt der Entstehung dieses Buchs so allerdings nicht mehr gibt. Die hier genutzten Bezeichnungen stammen aber aus Zeiten, als die Prozessorsparte noch direkt zu Motorola gehörte.

Neben dem WORD benötigt auch eine Variable vom Typ Integer zwei Bytes als Speicher. Aber welches der beiden Bytes kommt im Speicher wo hin, welcher Teil wird als Erstes abgelegt? Man könnte jetzt vermuten, dass dies immer gleich ist. Dann wäre dieser Abschnitt hinfällig. Aber daran, dass der Abschnitt existiert, sehen Sie: Da gibt es Erklärungsbedarf.

Es geht hier darum, in welcher Reihenfolge die einzelnen Bytes im Speicher abgelegt werden. Darum spricht man auch von Bytereihenfolge oder – neudeutsch ausgedrückt – von der Byteorder.

Es gibt zwei Varianten, wie ein WORD oder auch ein Integer abgelegt werden kann. Dazu in den folgenden Abschnitten mehr.

Das Big-Endian-Verfahren, auch »Motorola-Format« genannt

Die erste Variante orientiert sich daran, wie wir normalerweise Zahlen schreiben. Wir fangen auf der linken Seite mit der höchstwertigen Ziffer an und bewegen uns dann nach rechts bis zur niederwertigsten Ziffer vorwärts oder rückwärts, wie Sie es sehen wollen. Übertragen auf den Speicher, bedeutet dies, dass an der ersten oder, was die Adresse der Speicherzelle angeht, niedrigsten Adresse der höchstwertige Teil der Zahl steht. Allerdings mit dem kleinen Unterschied, dass wir nicht einzelne Ziffern abspeichern, sondern pro Speicherplatz acht Bits, was zwei Ziffern in hexadezimaler Schreibweise entspricht. Dieses Verfahren wird Big-Endian genannt oder, da es von diesem Hersteller verwendet wurde, auch Motorola-Format.

Das Little-Endian-Verfahren, auch »Intel-Format« genannt

Kurze Zwischenfrage: Wie addieren Sie ohne Taschenrechner (ja, das geht!) auf dem Papier zwei mehrstellige Zahlen miteinander? Genau, Sie beginnen bei der Einerstelle und arbeiten sich dann stellenweise bis zur höchsten Stelle vor. Sie werden es kaum glauben, aber Computer und SPSen machen es auch so. Die ersten Computer mit Mikroprozessoren waren zwar deutlich schneller als Computer auf Röhrenbasis, aber es wurde trotzdem versucht, die Geschwindigkeit mit weiteren Tricks noch zu erhöhen. Deswegen gibt es auch

eine zweite Art, Zahlen abzuspeichern, die mehr als ein Byte belegen. Wie erwähnt erfolgt das Speichern von Zahlen beim Motorola-Format beginnend beim höchstwertigen Teil. Und hier lauern das Problem und der Grund für das weitere Verfahren. Der Prozessor muss bei dieser Methode zunächst an die Adresse springen, an der die niederwertigste Stelle abgelegt ist, und das kostet Zeit. Das andere Verfahren heißt Little-Endian oder Intel-Format. Bei diesem wird die Zahl mit dem niederwertigsten Teil beginnend abgelegt, was den Vorteil mit sich bringt, dass zum Beispiel eine Addition ohne vorherige Sprünge begonnen werden kann und damit schneller beendet ist.

Hier noch ein Tipp, wie Sie die beiden Methoden unterscheiden können. Wenn Sie die eine Methode nicht Little-Endian nennen, sondern leicht verfälscht, »Little End In« (übersetzt: »kleines Ende rein«) aussprechen, können Sie es sich vielleicht etwas leichter merken. Entsprechend sagen Sie statt Big-Endian dann »Big End In«.

Alles, was Sie hier gelernt haben, gilt auch für Integerzahlen mit mehr als zwei Bytes, zum Beispiel für DINTS, oder neben dem Bit-Container WORD auch für ein DWORD.

Nachdem ich Ihnen erläutert habe, wie Daten im Speicher abgelegt werden, die mehr als ein Byte belegen, möchte ich hier noch erläutern, wie vorzeichenbehaftete Zahlen in binärer Schreibweise dargestellt werden.

Ablage von vorzeichenbehafteten Variablen im Speicher

Für die Erklärung, wie vorzeichenbehaftete Variablen im Speicher abgelegt werden, werde ich die Bytereihenfolge Big-Endian verwenden. Wie die dezimale Zahl 257 in binärer Schreibweise dargestellt wird, sollte nach der vorherigen Erklärung der Binärzahlen für Sie nicht mehr ganz so schwer sein. Sie wissen also längst: In binärer Schreibweise mit 16 Bits wird die Zahl als 0000 0001 0000 0001 dargestellt.

Aber wie kann jetzt zum Beispiel die Zahl –1 in binärer Schreibweise dargestellt werden? Wenn Sie darüber kurz nachdenken, kommen Sie vermutlich auf die Idee, einfach ein Bit für das Vorzeichen zu nehmen und den Rest, um die Zahl dazustellen. Am sinnvollsten wäre hier wohl das höchstwertige Bit, bei 16 Bit also das 16te. Der Wert –1 würde dann als 1000 0000 0000 0001 in binärer Schreibweise dargestellt werden. Sieht erst mal gar nicht schlecht aus, ist aber leider nicht optimal.

Ein »Problem« ist, dass es bei dieser Lösung zwei Werte für die Zahl 0 gibt. Einen positiven und einen negativen, was mathematisch nicht sinnvoll ist, da die 0 zwar eine gerade Zahl, aber weder positiv noch negativ ist. Außerdem bewirken die beiden Werte für die 0, dass sich die Programmierung aufwendiger gestaltet.

Was wird denn dann stattdessen verwendet? Das verwendete Verfahren nennt sich Zweierkomplement. Bei diesem Verfahren gibt es kein Bit, das dem Vorzeichen entspricht, und somit existiert auch das Problem nicht, dass es für die Zahl 0 zwei Werte gibt. Dennoch kann man mit einem Blick erkennen, ob eine Zahl positiv oder negativ ist.

Bleiben wir für die Erklärung einmal bei der Zahl –1 und gehen Schritt für Schritt durch, wie diese als Zweierkomplement in binärer Schreibweise dargestellt wird.

1. Vorzeichen ausblenden und dadurch aus der –1 eine 1 machen.
2. Diese Zahl mit 16 Bit als Binärzahl darstellen, also als 0000 0000 0000 0001.
3. Die Zahl umkehren, dies wird auch invertieren genannt. Hat eine Stelle den Wert 1 wird diese Stelle zu einer 0 und umgekehrt. Die daraus resultierende Binärzahl ist 1111 1111 1111 1110.
4. Zu dieser Zahl 1 hinzuaddieren, dadurch ergibt sich 1111 1111 1111 1111, was das Endergebnis ist.

Die Dezimalzahl –1 wird also in binärer Schreibweise als 1111 1111 1111 1111 dargestellt.

Versuchen Sie einmal die Zahl –32768 in binärer Schreibweise darzustellen. Wenn Sie 1000 0000 0000 0000 herausbekommen haben, liegen Sie richtig.

Damit hätten wir die größtmögliche und die kleinstmögliche negative Zahl, die eine Variable vom Typ Integer aufnehmen kann, abgehandelt, alle anderen negativen Zahlen bewegen sich dazwischen. Somit sehen Sie hoffentlich auch, woran Sie erkennen, ob eine Zahl negativ ist. Immer dann, wenn das oberste Bit gesetzt ist, ist die Zahl negativ, allerdings darf man hier nur bedingt von einem Vorzeichen-Bit sprechen.

Warum gibt es mehr negative als positive Zahlenwerte?

Als Letztes möchte ich noch kurz erklären, warum der Wertebereich bei den negativen Zahlen um 1 größer ist als bei den positiven Zahlen. Für die positiven Zahlen und auch für die Darstellung der 0 wird das oberste Bit nicht genutzt, bei den negativen schon, und damit ist deren Wertebereich um 1 größer.

Ablage von Fließkommazahlen im Speicher

Neben den Variablen für Ganzzahlen gibt es auch Variablen, die sogenannte Fließkommazahlen speichern können. Eine Fließkommazahl ist eine Zahl mit Nachkommastellen, bei der das Komma an einer (in gewissem Rahmen) beliebigen Stelle stehen kann. Die Position des Kommas ist also fließend, daher der Name. Dabei ist die Anzahl der darstellbaren Dezimalstellen begrenzt.

Das Verfahren, das hinter der Ablage einer Fließkommazahl steckt, ist relativ kompliziert. Soweit Sie Interesse an Details haben, suchen Sie im Internet einmal nach dem Begriff »Gleitkommazahl« oder der Norm »IEEE 754«.

Es gibt Fließkommazahlen mit verschiedenen Genauigkeiten und damit unterschiedlich großem Speicherbedarf. Bei SPSen sind zwei Sorten von Fließkommazahlen gebräuchlich. Nämlich Fließkommazahlen

1. mit einfacher Genauigkeit, die 4 Bytes belegen und 7 bis 8 Dezimalstellen darstellen können, und
2. solche mit doppelter Genauigkeit, die 8 Bytes belegen und 15 bis 16 Dezimalstellen genau darstellen können.

Im SPS-Umfeld heißen diese Variablen REAL und LREAL, die REAL-Variable ist in der Auflistung am Anfang dieses Kapitels auch aufgeführt.

Um Sie nicht völlig im Dunkeln zu lassen und Ihnen zugleich zu zeigen, dass die Umwandlung tatsächlich nicht so einfach ist und ihre Tücken hat, möchte ich hier kurz ein Beispiel anführen.

Eine Zahl wird in mehrere Teile gewandelt. Das erste Bit ist das Vorzeichenbit, die nächsten Bits enthalten den Exponenten und die restlichen die Mantisse. Soweit Ihnen die Begriffe *Exponent* und *Mantisse* geläufig sind, werden Sie erkennen, dass eine Fließkommazahl in Form einer Exponentialschreibweise abgelegt wird.

Wenn Sie die Begriffe noch nicht kennen, gar kein Problem! Hier eine kurze Aufklärung.

Neben den bereits erwähnten verschiedenen Zahlensystemen, in denen eine Zahl dargestellt werden kann, besteht auch noch eine weitere Möglichkeit, eine Zahl darzustellen – nämlich in der *Exponentialschreibweise*. Die Zahl 2500 könnten Sie somit auch als $2{,}5 \times 10^3$ schreiben, wobei 2,5 die Mantisse und 10^3 der Exponent ist. In eine »normale« Formel gebracht, sähe die Exponentialschreibweise dann so aus.

$$2{,}5 \times 10 \times 10 \times 10 = 2500$$

Bei obiger Schreibweise wurde als Basis die 10 genutzt. Bei der Ablage von Fließkommazahlen im Speicher einer SPS oder eines Computers wird als Basis die zum Binärsystem passende 2 verwendet.

Mögliche Probleme bei der Nutzung von Fließkommazahlen

Da sowohl die Mantisse als auch der Exponent nur eine bestimmte Anzahl Bits verwenden, ist die Anzahl der möglichen Kombinationen begrenzt. Dies hat zur Folge, dass nicht jede beliebige Fließkommazahl darstellbar ist, auch wenn die maximal mögliche Anzahl an Stellen nicht überschritten wurde. In einem solchen Fall wird auf die nächste darstellbare Fließkommazahl abgerundet.

Wenn Sie Fließkommazahlen auf einen bestimmten Wert abfragen wollen, müssen Sie immer auf einen Bereich mit einer für Ihre Anwendung ausreichenden Genauigkeit abfragen, aber nie auf eine genaue Zahl.

In Abbildung 1.5 sehen Sie zwei jeweils als Fließkommazahl deklarierte Variablen. Die Variable `rValue01` ist vom Typ REAL und die Variable `lrValue01` ist vom Typ LREAL. Ich habe der Variablen `rValue01` den Wert 2,34 zugewiesen. Dabei handelt es sich um einen Wert, der sich nicht genau als Fließkommazahl darstellen lässt.

Sie werden sich jetzt vielleicht fragen, wieso ich behaupte, dass sich dieser nicht darstellen lässt, wo bei der Variablen `rValue01` doch `2.34` steht. Um es mit den Worten des Fernsehdetektivs Magnum zu sagen: Sie haben recht, da steht tatsächlich `2.34`, aber das täuscht. Die Abweichung oder der Fehler, wie Sie es auch nennen möchten, liegt bei den letzten darstellbaren Stellen. Dadurch erfolgt eine Rundung auf den augenscheinlich richtigen Wert.

Über einen Befehl, den ich etwas später in diesem Kapitel noch erkläre, habe ich den Wert der REAL-Variablen einer LREAL-Variablen zugewiesen, und hier wird die Abweichung dann deutlich.

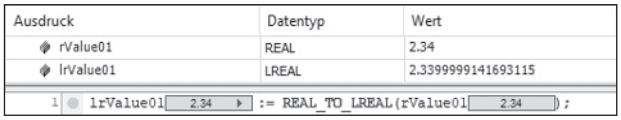


Abbildung 1.5: Rundungsfehler bei Fließkommazahlen

Für ein weiteres Beispiel dafür, dass nicht alle Zahlen als Fließkommazahlen darstellbar sind, möchte ich Ihnen zunächst eine weitere Quizfrage stellen.

Was ergibt 2,7773 multipliziert mit 10?

Was meinen Sie? Es ergibt 27,773? Da ist meine SPS aber anderer Meinung, was Sie in Abbildung 1.6 sehen können. Lassen Sie sich bitte nicht von den Zahlen im unteren Codeteil irritieren, diese werden wieder gerundet dargestellt, aber im oberen Deklarationsteil steht das Ergebnis in seiner ganzen falschen Schönheit, nämlich 27,7729988. Dies ist ein weiterer Nachweis für die Richtigkeit der Behauptung, dass man mit Fließkommazahlen nicht beliebige Zahlen darstellen kann.

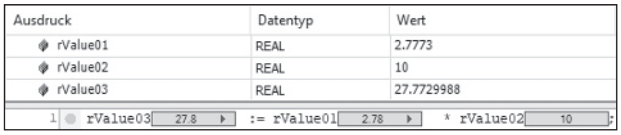


Abbildung 1.6: Fehler beim Rechnen mit Fließkommazahlen

Ablage von Festkommazahlen im Speicher

Eine weitere Variante zur Darstellung von Zahlen mit Nachkommastellen sind sogenannte Festkommazahlen. Bei diesen steht das Komma, wie der Name schon vermuten lässt, immer an einer festen Stelle.

Ein Beispiel für die Verwendung einer Festkommazahl im Bereich der SPS sind die Messergebnisse von Klemmen für Temperatursensoren. Diese werden als Integerzahl, also als Ganzzahlen übermittelt, allerdings nicht in °C, sondern zum Beispiel in 1/10°-Schritten. Der Wert 205 entspricht in diesem Fall 20,5 °C.

Ablage von Zeichen und Zeichenketten im Speicher

Neben den bisher erwähnten Variablen gibt es noch weitere Variablentypen. Zum Beispiel Variablen zum Speichern einzelner Buchstaben oder Texte. Diese Typen heißen CHAR für einzelne Buchstaben und STRING für Texte. Ein CHAR belegt dabei ein Byte, ein STRING

hat eine variable Länge, die beim Anlegen der Variablen angegeben werden muss, wobei pro Zeichen auch ein Byte belegt wird.

Genaugenommen wird in einem CHAR jedoch kein Buchstabe und in einem STRING nicht einer oder mehrere Buchstaben gespeichert, denn ein Computer und somit auch eine SPS wissen nicht, was ein Buchstabe ist. Für Buchstaben und andere Zeichen werden stattdessen Zahlen abgelegt, deren Wert einem Buchstaben entsprechen. Zum Beispiel entspricht die Zahl 65 dem Buchstaben »A«, die 72 dem »H«, die 97 dem »a« und die 104 dem »h«.

Unterschiede zwischen dem CODESYS-Universum und Siemens

Zwischen Steuerungen aus dem CODESYS-Universum und Siemens gibt es bei Strings grundlegende Unterschiede bei der Speicherung.

Bei Steuerungen, die auf CODESYS basieren, kann ein String auf die beiden folgenden Arten deklariert werden.

```
sText01    : STRING(10);
sText02    : STRING[10];
```

Bei diesen Beispielen wird eine Variable vom Typ STRING angelegt, die bis zu zehn Zeichen aufnehmen kann. Ein Zeichen belegt ja 1 Byte. Dennoch erhält man, wenn man mit einem Befehl im SPS-Programm die Größe dieser Variablen abfragt, als Größe 11 Bytes gemeldet. Das hängt damit zusammen, wie in CODESYS das Ende eines Strings gekennzeichnet wird. Die Variable kann zwar bis zu zehn Zeichen aufnehmen, aber was ist, wenn Sie einen kürzeren Text ablegen möchten? Sollte dies der Fall sein, wird von der SPS automatisch der Wert 0 hinter dem letzten Zeichen angefügt. Würden Sie zum Beispiel die Buchstaben »ABCDEFG« in dieser Variablen ablegen, würde im Speicher »65 66 67 68 69 70 71 0« stehen. Da die 0 auch angefügt wird, wenn die Variable komplett mit einem Text gefüllt ist, muss die Variable immer ein Byte mehr Speicher belegen, als für die maximale Textlänge tatsächlich benötigt werden würde. Im Falle von obigem Beispiel somit elf Bytes.

In der Abbildung 1.7 ist die Deklaration eines Strings mit maximal zehn Zeichen in TIA zu sehen.

Abbildung 1.7: Deklaration eines Strings bei Siemens

Bei Siemens nimmt diese Variable nun 12 Bytes ein. Aber warum das nun wieder? Bei Siemens sind im Speicher vor dem eigentlichen Text zwei Bytes vorhanden. Das erste Byte enthält die maximale Größe des Strings. In diesem Fall steht in diesem Byte die Zahl 10. Das zweite Byte gibt an, wie groß der aktuell gespeicherte Text ist, im Falle von obigem Beispiel würde hier eine 7 stehen.

Die hier vorgestellten Variablen zur Ablage von einzelnen oder mehreren Zeichen können, von wenigen Ausnahmen abgesehen, nur lateinische Zeichen und arabische Ziffern enthalten. Was ist jetzt aber, wenn Sie zum Beispiel griechische Buchstaben oder gar chinesische Zeichen ablegen möchten? Mit den vorgestellten Variablen geht dies leider nicht. Es setzen sich bei den verschiedenen Herstellern aber immer mehr neue Variablentypen durch, die Zeichen als sogenannten Unicode ablegen. Eine zum Zeitpunkt der Entstehung dieses Buchs gängige Codierungsart ist UTF-8. Dabei wird ein Zeichen mit bis zu vier Bytes beschrieben.

Pointer und Referenzen

Die bisher vorgestellten Variablen sind sogenannte *Wertevariablen*, das heißt, dass an der Speicherstelle, an der die Variable abgelegt ist, deren Wert liegt. Jede neu deklarierte Wertevariable belegt ihren eigenen Speicher und kann unabhängig von anderen Variablen desselben Typs geändert werden.

Eine weitere Gruppe von Variablen sind die sogenannten *referenziellen Variablen*. Bei diesen wird nicht der eigentliche Wert abgelegt, sondern eine Referenz, daher der Name. Eine Referenz ist eine Adresse, unter der dann der eigentliche Wert liegt. Referenzielle Variablen werden auch als *Zeiger* bezeichnet oder neudeutsch *Pointer*.

Jede Wertevariable hat ihren eigenen Speicherbereich. Wenn Sie also Daten von einer Variablen an eine andere desselben Typs übergeben, benötigen Sie schon den doppelten Speicherplatz, was speziell bei größeren Variablen, zum Beispiel den später vorgestellten *Arrays*, zu Problemen führen kann.

Mal angenommen, Sie möchten an einer anderen Stelle Ihres SPS-Programms auf Variablen zugreifen, auf die Sie direkt keinen Zugriff haben. In einem solchen Fall können Sie, statt die Variable in eine andere zu kopieren, eine Referenz auf die Variable in Form eines Pointers weitergeben. In diesem Fall enthält die neue Variable keine Daten, sondern die Adresse der anderen Variablen.

Bei Wertevariablen arbeiten Sie mit einer vollständigen Kopie der ursprünglichen Werte. Änderungen wirken sich nicht auf den Wert der ursprünglichen Variablen aus. Bei Pointern ist dies anders. Da hier nur ein Verweis verwendet wird, erfolgt die Änderung bei den »Originaldaten«.

Zunächst erkläre ich Pointer bei CODESYS-basierten Steuerungen und später bei Siemens.

Die Deklaration erfolgt ähnlich wie die Deklaration einer Wertevariablen. In Abbildung 1.8 ist die Deklaration zweier Wertevariablen vom Typ Integer und die Deklaration eines Pointers zu sehen, der auf eine Integervariable zeigt. Warum bei der Deklaration des Pointers die Angabe des Variablentyps wichtig ist, auf den dieser zeigt, werde ich noch erläutern.

```
FB_ST04
1  FUNCTION_BLOCK FB_ST04
2  VAR
3      iValue01    : INT := -30;
4      iValue02    : INT;
5      piValue     : POINTER TO INT;
6  END_VAR
```

Abbildung 1.8: Deklaration eines Pointers

Wenn bei der Deklaration keine Zuweisung beim Pointer erfolgte, enthält dieser noch keine gültige Adresse, auf die er verweist. Diese muss ihm dann im Programm erst zugewiesen werden. In Abbildung 1.9 sind wieder die schon in Abbildung 1.8 abgebildete Deklaration und ein kleines Programm zu sehen.

```
FB_ST04
1  FUNCTION_BLOCK FB_ST04
2  VAR
3      iValue01    : INT := -30;
4      iValue02    : INT;
5      piValue     : POINTER TO INT;
6  END_VAR
---
1  piValue     := ADR(iValue01);
2  iValue02    := piValue^;
```

Abbildung 1.9: Programm mit Pointer

In der ersten Zeile des Programms wird dem Pointer die Adresse der Variablen `iValue01` zugewiesen. Das geschieht mithilfe der Funktion `ADR`. Der Funktion wird als Parameter in Klammern die Variable übergeben, deren Adresse ermittelt werden soll.

In der zweiten Zeile wird der Variablen `iValue02` der Wert der Variablen zugewiesen, auf die der Pointer gerade zeigt. Dies erfolgt durch Nutzung des sogenannten *Inhaltsoperators* ^. Wenn Sie diesen Inhaltsoperator vergessen, würde der Variablen, soweit diese den passenden Typ hat, als Wert die Adresse zugewiesen werden, auf die der Pointer zeigt. Falls der Typ nicht passt, würde eine Warn- oder Fehlermeldung erfolgen. In Abbildung 1.10 ist das Ganze einmal in Aktion zu sehen.

FB_ST_04 [Online]

Kapitel01.Kapitel01.MAIN.fbST_04

Ausdruck	Datentyp	Wert
iValue01	INT	-30
iValue02	INT	-30
piValue	POINTER TO INT	16#FFFF8A0FA116E1B8
piValue^	POINTER TO INT	-30

```
1  piValue 16#FFFF8A0FA116E1B8  := ADR(iValue01 -30 );
2  iValue02 -30  := piValue^ -30 ; RETURN
```

Abbildung 1.10: Laufendes Programm mit Pointern

In der ersten Programmzeile ist die Adresse zu sehen, an der die Variable `iValue01` liegt, die dem Pointer `piValue` zugewiesen wurde. In der zweiten Zeile wird, wie schon beschrieben, der Variablen `iValue02` der Wert der Variablen zugewiesen, auf die der Pointer zeigt.

Mögliche Fehler bei der Nutzung von Pointern

Ich hatte bereits erwähnt, dass Sie bei der Deklaration eines Pointers nach Möglichkeit darauf achten sollten, dass der angegebene Typ zum Typ der Variablen passt, auf den der Pointer zeigen soll. Aber warum ist das so?

Den Grund dafür können Sie in Abbildung 1.11 sehen. In dieser Abbildung ist ein auf den ersten Blick seltsamer Effekt zu sehen, der jedoch logisch ist, wenn man genauer darüber nachdenkt.

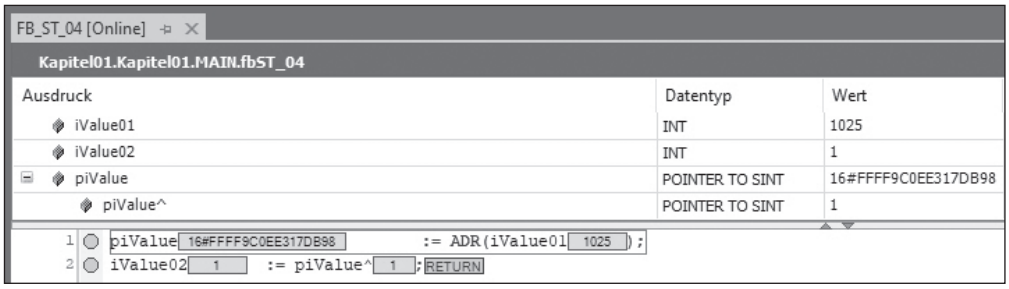


Abbildung 1.11: Die Folgen bei der Wahl des falschen Variablentyps bei Pointern

Anstatt INT bei der Deklaration des Pointers anzugeben, wurde in dem Beispiel der Typ SINT angegeben. Der Unterschied zwischen diesen beiden Typen liegt darin, dass ein INT zwei Bytes belegt, ein SINT aber nur ein Byte. Diese »falsche« Deklaration hat nun Auswirkungen auf den Inhaltsoperator. Die Funktion, die hinter diesem Operator steckt, orientiert sich an dem bei der Deklaration des Pointers angegebenen Typ, auf den dieser zeigen soll. In diesem Fall ein SINT.

Doch wie kommt nun der Wert 1 zustande? Dies wird hoffentlich klar, wenn ich das Bitmuster, das der Zahl 1025 entspricht, in hexadezimaler Schreibweise darstelle. In hexadezimaler Schreibweise wird die Zahl als 0x0401 dargestellt. Da ein SINT nur ein Byte im Speicher belegt, wird nur der untere Teil des ursprünglichen Werts zugewiesen, also 0x01, und damit landen Sie bei der 1.

Nun zu Pointern bei Siemens in TIA. Hier sieht das Ganze etwas anders aus, wie Sie in Abbildung 1.12 sehen können.

Statt einer Adresse wird eine Referenz übergeben. Sie werden sich jetzt vielleicht fragen, wo der Unterschied ist. Auch bei einer Referenz wird doch sicher eine Adresse übertragen. Das stimmt, aber die Funktion `REF` macht noch mehr. Bei Siemens erfolgt zusätzlich eine Typüberprüfung, also eine Überprüfung, ob die Variable, die mit REF referenziert werden soll, denselben Typ wie der Pointer hat. Ist dies nicht der Fall, erfolgt beim Übersetzen des Programms eine Fehlermeldung.

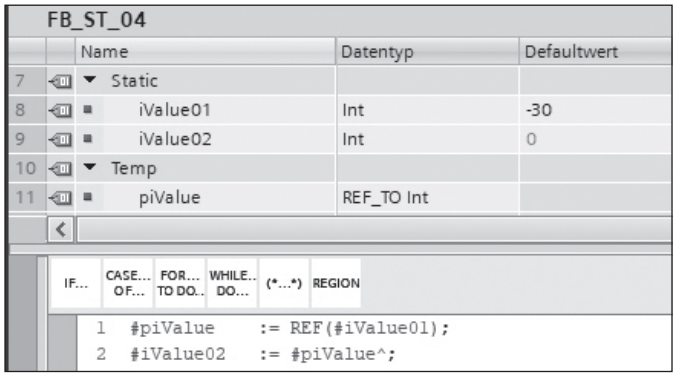


Abbildung 1.12: Pointer in TIA

Wissenswerts zu Arrays

Bisher habe ich Ihnen einzelne Variablen vorgestellt. Aber was machen Sie, wenn Sie mehrere Variablen gleichen Typs für ähnliche Dinge benötigen?

Mal angenommen, an Ihre Steuerung sind zehn Temperaturfühler angeschlossen, die Temperaturen von –100 °C bis 100 °C messen. Die Analogeingabebaugruppen, die die Messwerte der Fühler auswerten, geben die gemessene Temperatur als Festkommazahl an, also als Zahl mit einer festen Kommaposition, wobei im Speicher die Zahl ohne Nachkommastellen in 1/10°-Schritten abgelegt wird. Der Wert 205 entspricht dann einer Temperatur von 20,5 °C.

Sie können jetzt natürlich 10 einzelne Variablen vom Typ INTEGER deklarieren, wie im folgenden Beispiel in gekürzter Form zu sehen ist.

```
iTempVorlauf  : INT;
iTempRücklauf  : INT;
iTempRaum01  : INT;

i TempRaum08  : INT;
```

Das geht aber eventuell zulasten der Übersichtlichkeit, bläht das Programm auf und verhindert, dass man die Werte auf eine effektivere Art verarbeitet.

Deklaration von Arrays

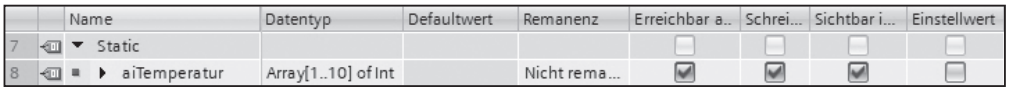
Die Lösung für diese Probleme heißt ARRAY. Ein ARRAY ist, wenn man so möchte, eine Liste mit Variablen eines Typs. Die Deklaration bei CODESYS-basierten Steuerungen sieht wie folgt aus.

```
aiTemperatur   : ARRAY[1..10] OF INT;
```

Die Deklaration sieht nicht wesentlich anders aus als bei den anderen Beispielen. Hier mal die Beschreibung der einzelnen Punkte der Deklaration.

1. Dem Array einen Namen geben.
2. Der Entwicklungsumgebung mitteilen, dass es sich nicht um eine einzelne Variable, sondern um eine Sammlung von Variablen eines bestimmten Typs handelt, nämlich um ein Array, und das ist der Unterschied zu einer einzelnen Variablen. Ein Array entsteht, indem Sie hinter dem Doppelpunkt das Schlüsselwort ARRAY hinzufügen.
3. Den Bereich des Arrays angeben. Ich komme später dazu, was das bedeutet. Fürs Erste reicht es, wenn Sie wissen, dass obiges Array aus zehn Elementen besteht.
4. Hinter der Definition des Bereichs des Arrays das Schlüsselwort `OF` anfügen. Hier endet der Unterschied zur Deklaration einer einzelnen Variablen.
5. Angabe des Variablentyps der einzelnen Elemente des Arrays. Wie bei der Deklaration der einzelnen Variablen zur Aufnahme der Temperatur ist der Typ auch bei obigem Beispiel wieder ein Integer.

In Abbildung 1.13 ist zu sehen, wie die Deklaration bei Siemens aussieht.



	Name	Datentyp	Defaultwert	Remanenz	Erreichbar a..	Schrei...	Sichtbar i...	Einstellwert
7	Static				☐	☐	☐	☐
8	aiTemperatur	Array[1..10] of Int		Nicht rema...	☑	☑	☑	☐

Abbildung 1.13: Deklaration eines Arrays in TIA

Auch hier sind die Unterschiede zu einer einzelnen Variablen überschaubar.

Wie werden Arrays genutzt?

Doch wie arbeitet man mit einem Array? Nun, ähnlich wie mit einer einzelnen Variablen auch, nur dass ein Array aus mehreren Elementen eines Variablentyps besteht und man das gewünschte Element auswählen muss. Dies geschieht über den Index, der hinter dem Array in Klammern steht.

Das Array aus obigem Beispiel besteht aus 10 Elementen. Die Anzahl der Elemente, aus denen ein Array besteht, ergibt sich aus dem Start- und Endindex, die bei der Deklaration angegeben wurden. Der Startindex muss nicht bei 1 beginnen, er kann auch 0 oder eine andere, sogar negative Zahl sein. Zu beachten dabei ist nur, dass der Endindex immer größer als der Startindex sein muss, ansonsten kommt es beim Übersetzen Ihres Programms zu einer Fehlermeldung.

Die Anzahl der Elemente eines Index erhalten Sie, indem Sie den Startindex vom Endindex abziehen. Da das Startelement auch mitgezählt werden muss, muss zum Ergebnis dann noch der Wert 1 addiert werden. Bezogen auf obiges Beispiel ergibt 10 – 1 den Wert 9, dies plus 1 ergibt 10 Elemente.

Varianten von Arrays

Das vorgestellte Array ist ein sogenanntes *eindimensionales Array*. Übertragen auf eine Tabellenkalkulation wäre das so, als ob alle Elemente des Arrays untereinander in einer Spalte stehen.

Mal angenommen, Sie möchten jede Stunde eine Temperatur messen und diese abspeichern. Das würde mit dem eindimensionalen Array funktionieren. Aber was, wenn Sie die Messwerte nicht nur von einem Tag erfassen möchten, sondern von einer ganzen Woche? Dann wäre ein eindimensionales Array nicht ausreichend.

Hier muss ein *mehrdimensionales Array* genutzt werden, genauer, ein *zweidimensionales Array*. Wieder übertragen auf eine Tabellenkalkulation würden auch dort die Messwerte eines Tages untereinander in einer Spalte erfasst werden, die Messwerte der folgenden Tage stehen dann in den folgenden Spalten.

Doch wie wird so ein Array erstellt? Ganz einfach: Fast so wie ein eindimensionales Array. Die folgende Deklaration erstellt ein Array

```
aiTemperatur   : ARRAY[1..7, 1..10] OF INT;
```

Die erste Dimension des Arrays geht von 1 bis 7 und enthält die Daten des jeweiligen Wochentages, die in der zweiten Dimension abgelegt sind, die von 1 bis 10 geht. Auf ein Arbeitsblatt einer Tabellenkalkulation übertragen würde jedes Element der ersten Dimension einer Spalte im Arbeitsblatt und somit einem Wochentag entsprechen, und in diesen Spalten würden dann die einzelnen Messwerte für jeden Tag untereinanderstehen.

Sie können bei der Deklaration auch weitere Dimensionen hinzufügen. Hier ein Beispiel für ein dreidimensionales Array, das die Temperaturen mehrerer Wochen beinhaltet.

```
aiTemperatur   : ARRAY[1..53, 1..7, 1..10] OF INT;
```

Hier erfolgt die Aufzeichnung von bis zu 10 Messwerten pro Tag über bis zu 53 Wochen, also über ein Jahr.

Bei Siemens sieht die Deklaration wieder ähnlich aus, erfolgt aber, je nach Einstellung in tabellarischer Form oder in der oben zu sehenden textlichen Form.

Wie werden Arrays genutzt (..., die Zweite)?

Wie greifen Sie nun auf die einzelnen Elemente eines Arrays zu? Wenn Sie auf einzelne Elemente eines Arrays zugreifen möchten, dann haben Sie zwei Möglichkeiten, die bei mehrdimensionalen Arrays auch kombiniert werden können. Zunächst geben Sie den Namen des Arrays an und anschließend in eckigen Klammern den Index des gewünschten Elements. Bei mehrdimensionalen Arrays müssen Sie entsprechend für jede Dimension den Index angeben.

Beispiel für ein-, zwei- und dreidimensionale Arrays. Bei diesen Beispielen wird der Wert eines Elements des Arrays einer anderen Variablen zugewiesen.

```
iTemperatur := aiTemperatur[5];
iTemperatur := aiTemperatur[3, 5];
iTemperatur := aiTemperatur[8, 3, 5];
```

Beim ersten Beispiel wird der fünfte an dem Tag gemessene Temperaturwert der Variablen `iTemperatur` zugewiesen. Beim zweiten der fünfte Messwert des dritten Tages und beim dritten wieder der fünfte Messwert ebenfalls des dritten Tages, aber in diesem Fall des dritten Tages der achten Woche.

Bei diesen Beispielen wurde direkt auf die einzelnen Elemente des Arrays durch Angabe des Indexes zugegriffen. Es gibt aber auch die Möglichkeit, den Zugriff flexibler zu gestalten.

Ich bin mir sicher: Hätte ich statt des Worts »flexibler« das Wort »variabler« genutzt, dann wären Sie sofort draufgekommen, mit welchem Element man dies erreichen kann. Möglich ist das, klar, mithilfe einer oder mehrerer Variablen vom Typ einer Ganzzahl, zum Beispiel einem INTEGER.

Beispiele, wie Sie auf einzelne Elemente eines Arrays mithilfe von Variablen zugreifen können:

```
iTemperatur   := aiTemperatur[iMesswert];
iTemperatur   := aiTemperatur[iTag, iMesswert];
iTemperatur   := aiTemperatur[8, iTag, iMesswert];
```

Bei den beiden ersten Beispielen werden die Indizes für den Zugriff auf ein Element des Arrays über Variablen vorgegeben. Beim dritten Beispiel wird der erste Index fest über die Angabe einer Zahl vorgegeben, die beiden weiteren Indizes wiederum über je eine Variable.

Bei mehrdimensionalen Arrays können Sie frei wählen, welchen Index Sie fest und welchen Sie variabel vorgeben möchten.

Mögliche Fehlerquellen bei der Nutzung von Arrays

Geben Sie beim Zugriff auf ein Element eines Arrays den oder bei mehrdimensionalen Arrays die Indizes direkt als Zahl an, wird deren Gültigkeit beim Übersetzen Ihres Programms überprüft. Liegen die Werte außerhalb der bei der Deklaration des Arrays definierten Grenzen, so wird eine Fehlermeldung erzeugt.

Wird beim Zugriff auf ein Arrayelement anstelle einer festen Zahl eine Variable verwendet, dann erfolgt keine Überprüfung bei der Übersetzung. Eine Prüfung ist auch nicht möglich, da eine Variable für eine Zahl während der Ausführung des Programms einen beliebigen Wert innerhalb ihres Wertebereichs annehmen kann.

Da hier der zulässige Bereich unter- oder überschritten werden kann, ist es möglich, auf Daten außerhalb des Arrays zuzugreifen, was Sie in Ihrem Programm verhindern müssen. Erfolgt ein Schreibzugriff außerhalb des zulässigen Bereichs,

werden Daten anderer Variablen überschrieben, und es kann auch zu einem Absturz der SPS kommen. Ein Absturz einer SPS führt dazu, dass sie in den Stopp geht und das Programm nicht mehr ausgeführt wird.

Bei Siemens erfolgt während der Ausführung des Programms, man spricht in diesem Fall auch von »zur Laufzeit«, eine Überprüfung. Erfolgt hier ein falscher Zugriff, geht die SPS in den Stopp. Hier spricht man dann aber nicht von Absturz, da in diesem Fall die SPS noch reagiert und man zum Beispiel durch Auslesen der Diagnose noch feststellen kann, was passiert ist.

Vieles zum Thema Strukturen und User Defined Types

Bisher habe ich Ihnen, hoffentlich erfolgreich, Wissen über die Arbeit mit einzelnen Variablen und mit Gruppen von Variablen eines Typs, sprich Arrays, vermittelt. Nun möchte ich noch auf die Möglichkeit eingehen, wie man Variablen unterschiedlicher Typen gruppieren kann.

Mal angenommen, Sie müssten die folgenden Daten für eine Achse erfassen.

- ✔ Name der Achse
- ✔ Start
- ✔ Stopp
- ✔ Geschwindigkeitssollwert
- ✔ Beschleunigungssollwert
- ✔ Verzögerungssollwert
- ✔ Zielposition

Sie können jetzt diese Variablen einzeln deklarieren und auf diese auch einzeln zugreifen. Wenn es aber nicht nur um eine Achse, sondern um mehrere geht, oder wenn noch weitere Variablen dazukommen, wird die Sache schnell unübersichtlich. Hier gibt es nun eine Möglichkeit, die Variablen zu gruppieren. Dazu in den folgenden Abschnitten mehr.

Die Deklaration von Strukturen und User Defined Type

Das Konstrukt, mit dem Sie Variablen gruppieren und so ordnen können, nennt sich *Struktur*.

Beispiel für die Deklaration einer Struktur basierend auf dem oben vorgestellten Beispiel einer Achse in Beckhoff TwinCAT:

```
TYPE ST_Axis :
STRUCT
```

```
        sName            : STRING[20];
        xStart           : BOOL;
        xStop            : BOOL;
        rVelocity        : REAL;
        rAcceleration    : REAL;
        rDeceleration    : REAL;
        rTargetPosition  : REAL;
    END_STRUCT
    END_TYPE
```

Die Deklaration von Strukturen erfolgt bei Beckhoff TwinCAT in einem eigenen Objekt. Dieses Objekt gehört bei Beckhoff zur Gruppe der *DUTs*, was in Langform für »**D**ata **U**nit **T**ype« steht.

Bei Siemens ist die Deklaration einer Struktur in tabellarischer Form in der Abbildung 1.14 zu sehen.

ST_Axis

	Name	Datentyp	Defaultwert	Erreichbar a..	Schrei...	Sichtbar i...	Einstellwert	Überwachung	Kommentar
1	sName	String[20]	''	☑	☑	☑	☐		
2	xStart	Bool	false	☑	☑	☑	☐		
3	xStop	Bool	false	☑	☑	☑	☐		
4	rVelocity	Real	0.0	☑	☑	☑	☐		
5	rAcceleration	Real	0.0	☑	☑	☑	☐		
6	rDeceleration	Real	0.0	☑	☑	☑	☐		
7	rTargetPosition	Real	0.0	☑	☑	☑	☐		

Abbildung 1.14: Deklaration einer Struktur in TIA

Bei Siemens heißt dieses mit den Strukturen von Beckhoff vergleichbare Objekt allerdings nicht Struktur, sondern Siemens nennt dieses Objekt in Kurzform *UDT*, was ausgeschrieben für »**U**ser **D**efined **T**ype« steht. Im Deutschen wird daraus »Benutzerdefinierter Typ« oder genauer »Benutzerdefinierter Datentyp«. Bei Siemens sind die UDTs im TIA-Portal im Ordner »PLC-Datentypen« zu finden.

Bei Siemens gibt es auch das Schlüsselwort `STRUCT`. Es kann beispielsweise verwendet werden, um eine Struktur direkt im Deklarationsteil zu erstellen, die nur an einer Stelle des Projekts verwendet wird. In Abbildung 1.15 sehen Sie ein Beispiel für die Deklaration einer solchen Struktur.

	Name	Datentyp	Defaultwert	Remanenz	Erreichbar a..	Schrei...	Sichtbar i...	Einstellwert
4	Static				☐	☐	☐	☐
5	stAxis01	Struct		Nicht rema...	☑	☑	☑	☐
6	sName	String[10]	'Master'	Nicht rema...	☑	☑	☑	☐
7	xStart	Bool	false	Nicht rema...	☑	☑	☑	☐
8	xStop	Bool	false	Nicht rema...	☑	☑	☑	☐
9	rVelocity	Real	100.0	Nicht rema...	☑	☑	☑	☐
10	rAcceleration	Real	300.0	Nicht rema...	☑	☑	☑	☐
11	rDeceleration	Real	250.0	Nicht rema...	☑	☑	☑	☐
12	rTargetPosition	Real	200.0	Nicht rema...	☑	☑	☑	☐

Abbildung 1.15: Deklaration einer Struktur in TIA zur einzelnen Nutzung

Im Vergleich zu einer einzelnen Variablen oder einem Array ist es mit dieser Deklaration nicht getan. Mit der bisherigen Deklaration wird nur der Aufbau der Struktur festgelegt. Zusätzlich muss auch eine Variable deklariert werden, die den Typ der Struktur hat.

Beispiel für die Deklaration einer Variablen vom Typ der Struktur ST_Axis bei einer CODESYS-basierten Steuerung.

```
stAchse01 := ST_Axis;
```

Wie können Sie Strukturen oder User Defined Types nutzen?

Doch wie erfolgt nun der Zugriff auf die einzelnen Elemente der Struktur? Auch das ist relativ einfach, über eine Kombination aus dem Namen, der eben deklarierten Variablen vom Typ der Struktur und dem gewünschten Element innerhalb der Struktur, auf das zugegriffen werden soll.

Angenommen, Sie möchten dem Element `sName` der Struktur den Text »Hauptantrieb« und dem Element `rVelocity` den Wert `100.0` vergeben, dann sähe das im Programm so aus.

```
stAchse01.sName       := 'Hauptantrieb';
stAchse01.rVelocity   := 100.0;
```

Was passiert nun, um beim Beispiel mit den Achsen zu bleiben, wenn Sie statt der einen Achse drei haben? Raten Sie mal. Wie bitte, was haben Sie da gesagt – Array? Absolut richtig, super!

Neben den Grundtypen wie Integer, WORD, String und so weiter kann ein Array auch aus einer bestimmten Anzahl von Strukturen bestehen. Die Deklaration ist dabei genau so wie bei den Grundtypen. Der folgende Codeschnipsel zeigt, wie ein Array von Strukturen deklariert wird.

```
astAchse   := ARRAY[1..4] OF ST_Axis;
```

Sie haben es sicher schon bemerkt: Dieses Thema ist sehr umfangreich.

Informationen zur Typkonvertierung

Ehe es mit anderen Programmierthemen weitergeht, möchte ich hier noch einen letzten Punkt zu Variablen anschneiden, der ebenfalls sehr wichtig ist.

Bei den bisherigen Erläuterungen wurde den Variablen entweder direkt ein zum Typ passender Wert zugewiesen oder der Wert einer anderen Variablen gleichen Typs. Aber was ist zu beachten, wenn dies einmal anders läuft? Angenommen, Sie müssen einer Variablen vom Typ INTEGER den Wert einer Variablen vom Typ DOUBLE INTEGER zuweisen.

Auf den ersten Blick scheint dies nicht zu gehen, da, bezogen auf den Speicherbedarf der beiden Variablentypen, ein DOUBLE INTEGER nicht in ein INTEGER passt. Ein DOUBLE INTEGER belegt 4 Bytes, während ein INTEGER 2 Bytes belegt. Sie bekommen ja, außer vielleicht im Märchen und selbst da nur mit Aufwand, einen Fuß nicht in einen viel zu kleinen Schuh.

Wie schon erwähnt, gibt es noch andere Möglichkeiten, einer Variablen einen Wert zuzuweisen. Verschiedene Programmteile, sogenannte *Funktionsbausteine* oder *Funktionen*, können auch Werte zurückliefern. Was Funktionsbausteine und Funktionen genau sind, das erkläre ich Ihnen etwas später in diesem Kapitel im Abschnitt »Weitere Objekte, beispielsweise Funktionen«. Im Moment reicht es für das weitere Verständnis, wenn Sie sich diese als Programmteile vorstellen, die Werte in Form sogenannter *Rückgabewerte* liefern können.

Es ist nicht möglich, bei SPS-Programmen mehrere solcher Programmteile mit demselben Namen, aber unterschiedlichen Rückgabewerten zu erstellen. Dies würde wohl auch der Übersichtlichkeit schaden. Die Programmteile liefern immer einen Wert mit einem bei der Erstellung des Programmteils definierten Variablentyp zurück.

Was ist also zu tun, wenn Sie etwas Großes in etwas viel Kleineres quetschen möchten oder müssen? Um beim bereits erwähnten Märchen zu bleiben: einfach den überschüssigen Teil abschneiden und den Rest in die kleinere Variable legen.

Umgekehrt geht es aber auch, etwas Kleineres kann in etwas Größeres abgelegt werden. Auf eine weitere Variante werde ich später noch eingehen.

Bei Siemens gibt es noch spezielle Konvertierungsfunktionen, die es im CODESYS-Universum, hier muss ich mal wertend werden, leider mit der Funktion nicht gibt. Das Verhalten bestimmter Konvertierungsfunktionen im CODESYS-Universum ist nur bedingt sinnvoll. Auf diese werde ich ebenfalls später noch im Abschnitt »Spezielle Konvertierungsfunktionen bei Siemens« eingehen.

Die Umwandlung, die man auch als *Konvertierung* bezeichnen kann, von einem Variablentyp in einen anderen, erfolgt mithilfe eines bestimmten Mechanismus.

Raten Sie bitte einmal, wie der Mechanismus heißen könnte, mit dem ein Variablentyp in einen anderen gewandelt werden kann.

Richtig, dieser Mechanismus nennt sich *Typumwandlung* oder *Typkonvertierung*.

Es gibt dabei zwei Arten von Typkonvertierungen:

1. die *implizite* und die

2. *explizite* Typkonvertierung.

Bei der impliziten Typkonvertierung erfolgt die Umwandlung in einen anderen Datentyp ohne die, jetzt wird es verwirrend, explizite Angabe eines Befehls. Die explizite Konvertierung müssen Sie aktiv durch – ich liebe diese Wortspiele – die implizite Nutzung eines entsprechenden Befehls ausführen.

Versuchen Sie, die implizite Typkonvertierung möglichst zu vermeiden, indem Sie immer einen Konvertierungsbefehl nutzen.

Der Grund für diese Warnung: Sie wissen bei einer impliziten Typkonvertierung nie genau, was die Entwicklungsumgebung beziehungsweise Ihre SPS da

eigentlich genau macht. Nur weil es bei mehreren Tests richtig zu funktionieren scheint, heißt das noch nicht, dass die Konvertierung so arbeitet, wie Sie es erwarten.

Auch wenn ich eben von der Nutzung der impliziten Typkonvertierung abgeraten habe, kann das für mich keine Ausrede sein, sie nicht zu erklären. Also los.

Eine implizite Typkonvertierung erfolgt, wie erwähnt, automatisch, ohne dass Sie extra einen Befehl dafür angeben müssen. Sie funktioniert aber nur bei bestimmten Bedingungen.

Die implizite Typkonvertierung funktioniert zum Beispiel dann problemlos, wenn Sie Daten einer Variablen einer kleineren Variante eines Typs einer größeren Variante zuweisen. Ich bitte diesen womöglich verwirrenden Satz zu entschuldigen, aber die folgende Erklärung sollte Klarheit bringen.

Angenommen, Sie möchten einer Variablen vom Typ UNSIGNED DOUBLE INTEGER oder kurz UDINT, den Wert einer Variablen vom Typ UNSIGNED INTEGER, kurz UINT, zuweisen, dann ginge dies problemlos mit der folgenden Zuweisung.

```
udiValue01    := uiValue01;
```

Eine Variable vom Typ UINT hat einen Wertebereich von 0 bis 65535, eine UDINT Variable einen von 0 bis 4.294.967.295, das passt also mehr als dicke.

Probleme, die bei der Typkonvertierung auftreten können

Etwas anders sieht es bei der Konvertierung einer INT-Variablen in eine UINT-Variable aus. In Abbildung 1.16 ist ein Programm für eine CODESYS-basierte Steuerung zu sehen, bei dem eine solche Konvertierung erfolgt. Abbildung 1.17 zeigt dasselbe mit Siemens TIA.

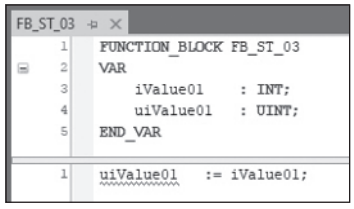


Abbildung 1.16: Implizite Typkonvertierung bei CODESYS-basierter Steuerung

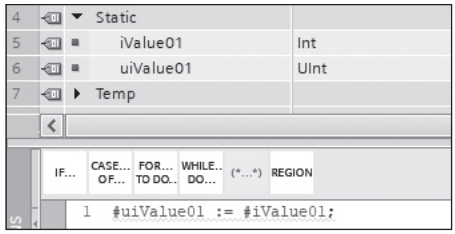


Abbildung 1.17: Implizite Typkonvertierung bei TIA

Bei beiden Abbildungen ist eine Wellenlinie unter dem Programmcode zu erkennen. Bei CODESYS-basierten Steuerungen ist diese blau, bei TIA dagegen gelb.

In beiden Fällen bedeutet die Wellenlinie dasselbe: Der markierte Code verursacht eine Warnung. Doch warum ist das so? Ein Blick auf die entsprechende Warnmeldung würde die Ursache offenbaren, aber diese möchte ich Ihnen im Moment noch ganz frech vorenthalten.

Die Frage kann auch leicht durch einen Blick auf die Wertebereiche der beiden Variablentypen beantwortet werden. Der Wertebereich einer Integervariablen geht bis 32767, der einer Unsigned-Integervariablen dagegen bis 65535. So weit ist noch alles gut und es besteht kein Grund für eine Warnung. Aber das angegebene Ende des Wertebereichs ist ja nur ein Ende, nämlich das obere Ende. Betrachten wir nun das untere Ende.

Das untere Ende des Wertebereichs eines INT ist der Wert –32768.

Bei welchem Wert ein UINT startet, ist auch bei der erwähnten Auflistung aufgeführt und wurde zusätzlich noch ein paar Zeilen weiter oben erwähnt.

Tun wir aber kurz mal so, als ob Sie dies alles nicht wüssten oder nachsehen könnten. Es gibt einen Weg, bei allen Unsigned-Variablen den kleinstmöglichen Wert herauszufinden. Der Schlüssel liegt im Wort »unsigned«, was übersetzt »vorzeichenlos« heißt oder auf Hochdeutsch »nicht negativ« bedeutet.

Welcher kleinste Wert, bei dem Sie starten könnten, fällt Ihnen auf Anhieb ein, der nicht negativ ist?

Ganz genau, die 0. Alle Unsigned-Variablen beginnen bei 0. Da die Zahl 0 weder positiv noch negativ ist, kann man hier die Bedingung »positiv« nicht alternativ nutzen.

Doch zurück zu den Warnmeldungen. Da die Zielvariable nur nicht negative Werte enthalten, die Quellvariable aber auch negative Werte beinhalten kann, kann es zu Problemen kommen. Und deshalb geben die Entwicklungsumgebungen eine Warnung aus.

Solange die Quellvariable keine negativen Werte enthält, ist alles gut. Bei negativen Werten treten auf den ersten Blick aber seltsame und nicht nachvollziehbare Effekte auf. Oder ist Ihnen klar, warum bei einem Wert von –32768 bei der Zielvariablen auf einmal eine 32768 steht oder aus einer –1 auf einmal eine 65535 geworden ist? Sollte Ihre Antwort »Ja klar!« lauten, dann ziehe ich den Hut vor Ihnen, denn dann haben Sie bei den vorherigen Erklärungen hervorragend aufgepasst. Aber Sie brauchen auch nicht enttäuscht zu sein, wenn Ihnen dieser Umstand noch nicht klar ist, es ist auch nicht so ganz einfach.

Die Ursache für diesen Effekt liegt in der Art, wie negative Zahlen abgelegt werden, und was die Typkonvertierung macht.

Negative Zahlen werden als Zweierkomplement abgelegt.

Die Zahl –32768 sieht als Binärzahl mit 16 Bit so aus:1000 0000 0000 0000, und die Zahl –1 entsprechend 1111 1111 1111 1111. Die Typkonvertierung kopiert nun einfach den Inhalt des Speichers der Quellvariablen an die Stelle der Zielvariablen, womit das Chaos seinen Lauf nimmt. Im Falle der Zahl –1 steht jetzt im Speicher der nicht vorzeichenbehafteten Integervariablen der höchstmögliche Wert. Da hier jetzt alle Bits gesetzt sind, ist das eben die 65535.

Ich muss gestehen, dass auch mich dieses Verhalten bei der Erstellung und dem Test von Programmbeispielen für dieses Buch ein wenig überrascht hat. Ich hatte ehrlich gesagt erwartet, dass die Konvertierungsfunktion bei negativen Zahlen einfach eine 0 in die Zielvariable schreibt, aber so kann man sich irren.

Nun reiche ich noch die Warnmeldungen nach. Mit »Das Vorzeichen oder die Genauigkeit des Werts kann verloren gehen« macht Siemens auf diesen Umstand aufmerksam. Bei CODESYS-basierten Steuerungen lautet die Warnmeldung »Implizite Konvertierung von vorzeichenbehaftetem Datentyp 'INT' nach nicht vorzeichenbehaftetem Datentyp 'UINT': Möglicherweise Änderung des Vorzeichens«. Beide Warnungen beschreiben meiner Meinung nach allerdings nicht das Chaos, in das man geraten kann, wenn man bei der Konvertierung von vorzeichenbehafteten Ganzzahlen nicht aufpasst.

Neben der impliziten Konvertierung gibt es noch die explizite Konvertierung, die eben nicht automatisch ausgeführt wird, sondern von Ihnen angestoßen werden muss. Aber wie? Einfach ausgedrückt: über die entsprechenden Befehle.

Die Konvertierungsbefehle, die sowohl bei CODESYS-basierten Steuerungen als auch bei Siemens TIA existieren, sind gleich aufgebaut. Der Name der jeweiligen Konvertierungsfunktion setzt sich aus dem Kürzel des Quelltyps, dem Schlüsselwort TO und dem Kürzel des Zieltyps zusammen, verbunden durch Unterstriche.

Für die Konvertierung eines INTEGER in ein UNSIGNED INTEGER hieße die Funktion also `INT_TO_UINT`. Für die Konvertierung eines DOUBLE INTEGER in ein INTEGER lautet der Befehl `DINT_TO_INT`.

Auch bei der expliziten Konvertierung müssen Sie aus denselben Gründen wie bei der impliziten Konvertierung aufpassen.

Bei der Konvertierung einer Integervariablen, die den Wert –1 hat, in eine Variable vom Typ UNSIGNED INTEGER tritt derselbe »Fehler« wie bei der impliziten Konvertierung auf, und die Variable hat den Wert 65535.

Spezielle Konvertierungsfunktionen bei Siemens

Im Abschnitt »Informationen zur Typkonvertierung« weiter oben hatte ich schon kurz erwähnt, dass es bei Siemens in TIA Konvertierungsbefehle gibt, die gegenüber CODESYS-basierten Steuerungen eine andere, sinnvollere Funktionsweise haben. Bei diesen Konvertierungsbefehlen handelt es sich um Befehle, mit denen ein DWORD in ein REAL konvertiert werden kann.

Vielleicht werden in diesem Moment einige von Ihnen protestieren, nachdem Sie den Satz gelesen haben. Welchen Sinn hat ein Befehl, der einen Variablentyp, der nicht für die Aufnahme von Zahlen gedacht ist, in einen Variablentyp konvertiert, der Fließkommazahlen aufnimmt? Die Antwort lautet: sehr viel Sinn.

Eine Steuerung verarbeitet ja Daten aus verschiedenen Quellen. Da kann es zum Beispiel vorkommen, dass die SPS einen Wert vom Typ REAL eines externen Geräts, zum Beispiel über eine serielle Verbindung als DWORD, erhält, da eine solche Übertragung nur eine Aneinanderreihung von Bits ist. Von der Anzahl der Bytes passt das auch, denn ein REAL und ein DWORD belegen beide vier Bytes, aber die Darstellung passt halt leider nicht.

Sie könnten nun den Speicherbereich, in dem das DWORD liegt, mit einem Kopierbefehl an den Speicherplatz einer REAL-Variablen kopieren. Schöner wäre es jedoch, wenn es einen Konvertierungsbefehl gäbe, der das Bitmuster im DWORD direkt in eine REAL-Variable überträgt.

Einen solchen Befehl gibt es bei Siemens in TIA. Dieser Befehl lautet `DWORD_TO_REAL`. Wenn in Ihrem DWORD in hexadezimaler Schreibweise zum Beispiel 0x42 0x00 0x00 0x00 stehen würde, steht in TIA nach Anwendung des Konvertierungsbefehls die Zahl 32, was dem Bitmuster als Fließkommazahl dargestellt entspricht.

Wenn Sie jedoch bei einer CODESYS-basierten Steuerung den dort auch verfügbaren und gleichlautenden Befehl ausführen, dann erhielten Sie in der REAL-Variablen den Wert 1,107296.26E+09. CODESYS-basierte Steuerungen rechnen hier anhand der gesetzten Bits und deren Stellenwertigkeit eine Zahl aus und legen diese dann als REAL ab, was nur bedingt sinnvoll ist.

Zum Themengebiet der Konvertierungen gehört auch der richtige Umgang mit direkt angegebenen Zahlen. Wenn Sie zum Beispiel eine Variable mit 3 multiplizieren und anschließend einer Variablen zuweisen möchten, dann könnten Sie dazu den folgenden Code nutzen.«

```
iWert02   := iWert01 * 3;
```

Quizfrage: Sind die Zahlen 3 und 3.0 dasselbe?

Sie werden sich sicher schon denken: Eine solche Frage, bei der die Antwort auf den ersten Blick eigentlich Ja lautet, muss vermutlich mit Nein beantwortet werden. Und Sie haben recht. Die Zahl 3 wird von der Entwicklungsumgebung sowohl bei Siemens als auch bei CODESYS-basierten Steuerungen als Ganzzahl interpretiert, also als Zahl ohne Nachkommastellen. Die Zahl 3.0 wird bei beiden Systemen als Fließkommazahl interpretiert.

Sie sollten immer auf die richtige Verwendung von Variablentypen beziehungsweise Zahlentypen und deren Kombination achten.

Warum das so ist beziehungsweise was passiert, wenn Sie diesen Tipp nicht beachten, erörtere ich näher in Kapitel 10 »Alles IO!«.

Wichtige Programmkonstrukte

Als Nächstes möchte ich Ihnen verschiedene wichtige Konstrukte erläutern. Wie am Anfang dieses Kapitels im Abschnitt »Einführung« schon erwähnt, nutze ich für die Einführung in die Programmierung hier lediglich eine Programmiersprache. Das Folgende kann auch in anderen Programmiersprachen realisiert werden. Teilweise ist dies allerdings mit einem erheblichen Aufwand verbunden, und die Sprachen sollten dafür nicht genutzt werden.

Bedingte Anweisungen, Verzweigungen und Mehrfachverzweigungen

Der erste Punkt, auf den ich eingehe, sind *bedingte Anweisungen*, *Verzweigungen* und *Mehrfachverzweigungen*. Es wäre natürlich schön, wenn ein Programm immer Zeile für Zeile ausgeführt werden würde, aber dieser Wunsch ist illusorisch. Es kommt immer wieder vor, dass ein Codeteil nur unter bestimmten Bedingungen ausgeführt werden soll, was eine bedingte Anweisung darstellt. Eine Verzweigung liegt vor, wenn bei einer Bedingung entweder ein Teil eines Programms ausgeführt wird und ansonsten ein anderer. Von einer Mehrfachverzweigung spricht man wiederum, wenn mehr als zwei Verzweigungen existieren.

Sowohl bei der bedingten Anweisung als auch bei der Verzweigung wird ein *boolescher Ausdruck* verwendet. Ein boolescher Ausdruck ist ein Ausdruck, dessen Ergebnis *entweder wahr*, also erfüllt, *oder falsch*, also nicht erfüllt, ist.

Ist die Bedingung wahr, wird im Falle der bedingten Anweisung die Anweisung oder der Anweisungsteil ausgeführt. Bei einer Verzweigung wird bei einem erfüllten Ausdruck der obere Anweisungsteil ausgeführt und bei einem nicht erfüllten Ausdruck der untere Anweisungsteil.

Sowohl eine bedingte Anweisung als auch eine Verzweigung verwenden hierfür eine `IF`-Anweisung. Der folgende Codeausschnitt soll Ihnen beispielhaft den Aufbau einer IF-Anweisung für eine bedingte Anweisung zeigen.

```
IF Bedingung THEN
   Anweisungsteil;
END_IF
```

Eine bedingte Anweisung wird mit dem Schlüsselwort `IF` eingeleitet, dann folgt die Bedingung und danach das Schlüsselwort `THEN`, nach diesem der Anweisungsteil, der mit dem Schlüsselwort `END_IF` endet. Der Anweisungsteil wird nur ausgeführt, wenn die Bedingung erfüllt ist.

Der nun folgende Codeausschnitt zeigt beispielhaft, wie die `IF`-Anweisung bei einer Verzweigung genutzt wird.

```
IF Bedingung THEN
   Anweisungsteil;
ELSE
   Anweisungsteil;
END_IF
```

Auch hier werden wieder die Schlüsselwörter IF, THEN und END_IF verwendet.

Im Vergleich zu einer bedingten Anweisung hat eine Verzweigung zusätzlich noch einen Alternativzweig, der immer ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Dieser Alternativzweig wird durch das Schlüsselwort ELSE eingeleitet.

Hier einmal ein paar Beispiele für Bedingungen mit booleschen Variablen.


```
IF xBoolVar THEN
IF xBoolVar = TRUE THEN
IF NOT xBoolVar THEN
IF xBoolVar = FALSE THEN
```

Beim ersten und zweiten Beispiel wäre die Bedingung erfüllt, wenn die boolesche Variable gesetzt, also TRUE ist. Das erste Beispiel ist eine verkürzte Form des zweiten Beispiels. Um eine boolesche Variable abzufragen, reicht es, sie allein als Bedingung zu schreiben, da sie ja selbst entweder schon TRUE oder FALSE ist, was eine boolesche Bedingung ja ausmacht. Beim dritten und vierten Beispiel ist die Bedingung dann erfüllt, wenn die Variable FALSE ist. Hier muss bei der Variante ohne Gleichheitszeichen das Schlüsselwort NOT der Variablen vorangestellt werden.

Es können auch mehrere einzelne Variablen oder Bedingungen zu einer »großen« Bedingung kombiniert werden. Die Kombination der Variablen oder Bedingungen erfolgt über bestimmte Schlüsselwörter, nämlich AND, OR und XOR.

Bei einzelnen Bedingungen, die mit AND verknüpft wurden, müssen alle Bedingungen erfüllt sein, damit die »übergeordnete«, also große Bedingung, erfüllt ist. Bei Variablen müssen diese, je nachdem, ob sie auf TRUE oder FALSE abgefragt werden, entsprechend TRUE oder FALSE sein. Hier ein paar Beispiele für mit AND kombinierte einzelne Bedingungen.


```
xBoolVar01 AND xBoolVar02 AND xBoolVar03
xBoolVar01 AND xBoolVar02 AND NOT xBoolVar03
xBoolVar01 = TRUE AND xBoolVar02 = TRUE AND xBoolVar03 = FALSE
```

Beim ersten Beispiel ist die Bedingung erfüllt, wenn alle drei Variablen TRUE sind. Beim zweiten Beispiel ist die Bedingung erfüllt, wenn die ersten beiden Variablen TRUE sind und die dritte FALSE ist. Das dritte Beispiel entspricht dem zweiten, allerdings in einer alternativen Schreibweise. Auch hier ist die Bedingung erfüllt, wenn die ersten beiden Variablen TRUE sind und die dritte FALSE ist.

Nun zwei Beispiele für mit OR verknüpfte Variablen. Bei mit OR verknüpften Variablen muss immer mindestens eine Variable TRUE sein beziehungsweise bei Nutzung des Schlüsselworts NOT entsprechend FALSE.


```
xBoolVar01 OR xBoolVar02 OR xBoolVar03
xBoolVar01 OR NOT xBoolVar02 OR xBoolVar03
```

Beim ersten Beispiel muss mindestens eine der drei Variablen TRUE sein, damit die Bedingung erfüllt ist. Beim zweiten Beispiel muss mindestens die zweite Variable FALSE sein oder die erste oder dritte TRUE.

Als Letztes stelle ich Ihnen ein paar Beispiele mit XOR verknüpften Variablen vor. XOR steht dabei für *Exklusiv Oder*. Bezüglich der XOR-Verknüpfung muss ich hier zunächst einen Hinweis loswerden.

Die XOR-Verknüpfung ist für die *Verknüpfung von zwei Variablen* gedacht. Bitte vermeiden Sie die Nutzung der XOR-Verknüpfung mit mehr als zwei Variablen. Es kann sein, dass sich die Verknüpfung anders verhält, als Sie es erwarten. Außerdem kann das Verhalten herstellerabhängig sein.

Bei mit XOR verknüpften Bedingungen oder Variablen ist die XOR-Bedingung nur erfüllt, wenn **genau eine** der beiden Bedingungen erfüllt oder die Abfrage einer Variablen auf TRUE oder FALSE erfüllt ist. Auch hier folgen nun ein paar Beispiele.


```
xBoolVar01 XOR xBoolVar02
xBoolVar01 XOR NOT xBoolVar02
xBoolVar01 XOR xBoolVar02 = FALSE
```

Beim ersten Beispiel ist die Bedingung erfüllt, wenn die Variable xBoolVar01 TRUE ist und die Variable xBoolVar02 FALSE oder wenn die Variable xBoolVar01 FALSE und die Variable xBoolVar02 TRUE ist. Beim zweiten und dritten Beispiel ist die Bedingung wiederum erfüllt, wenn nur die erste Variable TRUE ist oder nur die andere FALSE ist.

Ich weiß nicht, ob es Ihnen aufgefallen ist. Aber ich habe Ihnen bis jetzt etwas vorenthalten. Ich habe ja geschrieben, dass entweder boolesche Variablen oder Bedingungen verknüpft werden können. Aber in den Beispielen wurden bis jetzt nur boolesche Variablen verwendet. Das möchte ich jetzt ändern.

Mal angenommen, ein Programmteil soll nur ausgeführt werden, wenn der Wert einer Variablen für eine Ganzzahl einen bestimmten Wert überschritten hat. Dazu fragen Sie keine boolesche Variable darauf ab, ob sie TRUE oder FALSE ist, sondern Sie müssen in Verbindung mit dieser Variablen eine Bedingung erstellen, die in diesem Fall erfüllt sein muss, wenn die Variable größer als der gewünschte Wert ist. Dasselbe geht natürlich auch in die umgekehrte Richtung, also wenn die Variable einen bestimmten Wert unterschreitet. Hier ein paar Beispiele dazu.


```
iIntegerVar01 > 1500
iIntegerVar01 >= 1500
iIntegerVar01 < 1700
iIntegerVar01 <= 1700
iIntegerVar01 <> 1500
iIntegerVar01 > 1500 AND iIntegerVar01 < 1700
```

Beim ersten Beispiel ist die Bedingung erfüllt, wenn die Variable iIntegerVar01 größer als 1500 ist. Dann wird der entsprechende Programmteil ausgeführt. Beim zweiten Beispiel ist es ähnlich, aber hier ist die Bedingung schon erfüllt, wenn die Variable den Wert 1500 hat, und auch weiterhin, wenn sie größer als 1500 ist.

Die Beispiele drei und vier sind mit den ersten beiden vergleichbar, allerdings gehen sie in die entgegengesetzte Richtung, sprich die Bedingung ist erfüllt, wenn die Variable kleiner 1700 beziehungsweise gleich oder kleiner 1700 ist.

Beim fünften Beispiel ist die Bedingung erfüllt, wenn die Variable jeden Wert außer 1500 hat.

Natürlich können auch diese Bedingungen wieder kombiniert werden, und zwar mit den Ihnen schon bekannten Schlüsselwörtern. Das sechste Beispiel demonstriert dies. Bei diesem Beispiel würde der Programmteil ausgeführt, wenn der Wert der Variablen größer 1500, aber kleiner 1700 ist.

Bis jetzt habe ich Ihnen die bedingte Anweisung und die Verzweigung (hoffentlich) nähergebracht. Es fehlt noch die Mehrfachverzweigung. Wie der Name schon erahnen lässt, ist diese verwandt mit der Verzweigung, mit dem Unterschied, dass es hier mehr als zwei Verzweigungen mit einer zugehörigen Bedingung gibt.

Die **Mehrfachverzweigung** verwendet, wie die bedingte Anweisung und die Verzweigung auch, die `IF`-Anweisung, allerdings auch ein weiteres Konstrukt, auf das ich im Abschnitt »Besondere Form der Mehrfachanweisung (CASE-Anweisung)« weiter unten eingehen werde.

Bei der »einfachen« Verzweigung wird ja ein `IF`-Zweig verwendet, der ausgeführt wird, wenn die Bedingung erfüllt ist, und ein Alternativzweig, der mit dem Schlüsselwort `ELSE` eingeleitet wird, der in allen anderen Fällen ausgeführt wird. Für die weiteren Verzweigungen kommt bei der Mehrfachverzweigung eine Kombination von `IF` und `ELSE` zum Einsatz, das entsprechende Schlüsselwort lautet `ELSIF`. Der erste Teil des Schüsselworts kommt vom `ELSE`-Zweig und kennzeichnet diesen Teil als Alternativzweig, der zweite Teil, der vom `IF`-Zweig kommt, sorgt dafür, dass dieser Zweig nicht wie ein normaler Alternativzweig bedingungslos ausgeführt wird, sondern nur, wenn die angehängte Bedingung erfüllt ist. Hier ein paar Beispiele für Mehrfachverzweigungen.


```
IF xBoolVar01 AND iIntegerVar01 > 1500 THEN
   Anweisungsteil;
ELIF NOT xBoolVar01 AND iIntegerVar01 < 1200 THEN
   Anweisungsteil;
ELSIF xBoolVar01 AND iIntegerVar01 > 1200 AND iIntegerVar01 < 1500
THEN
   Anweisungsteil;
END_IF
IF xBoolVar01 OR xBoolVar02 THEN
   Anweisungsteil;
ELSIF xBoolVar03 XOR xBoolVar04 THEN
   Anweisungsteil;
ELSE
   Anweisungsteil;
END_IF
```

Beim ersten Beispiel wird der erste Anweisungsteil ausgeführt, wenn die boolesche Variable `xBoolVar01 TRUE` ist und die Variable `iIntegerVar01` einen Wert von größer 1500 hat. Der zweite Anweisungsteil wird ausgeführt, wenn die Variable `xBoolVar01 FALSE` ist und die Variable `iIntegerVar01` einen Wert von kleiner 1200 hat. Der dritte Teil wird wiederum ausgeführt, wenn die Variable `xBoolVar01 TRUE` ist und die Variable `iIntegerVar01` einen Wert von größer 1200 und kleiner 1500 hat.

Beim zweiten Beispiel erfolgt die Ausführung des ersten Anweisungsteils, wenn entweder die Variable `xBoolVar01` oder die Variable `xBoolVar02` oder beide `TRUE` sind. Der zweite Anweisungsteil wird ausgeführt, wenn entweder die Variable `xBoolVar03` oder die Variable `xBoolVar04 TRUE`, aber nicht, wenn beide `TRUE` sind. Ist keine der vorhergehenden Bedingungen erfüllt, wird der dritte Anweisungsteil ausgeführt.

Wie gesagt ist das Schlüsselwort `ELSIF` eine Kombination aus `IF` und `ELSE`. Man kann das Ganze allerdings auch ohne `ELSIF` realisieren, was ich Ihnen im Folgenden gerne anhand des zweiten Beispiels von eben demonstrieren möchte. Der folgende Code bewirkt dasselbe wie das erste Beispiel.


```
IF xBoolVar01 OR xBoolVar02 THEN
   Anweisungsteil;
ELSE
   IF xBoolVar03 XOR xBoolVar04 THEN
      Anweisungsteil;
   ELSE
      Anweisungsteil;
   END_IF
END_IF
```

Es wird hier aber immer nur die erste erfüllte Bedingung ausgeführt. Sind die Bedingungen mehrerer Verzweigungen erfüllt, gewinnt die erste erfüllte Bedingung. Die anderen Verzweigungen werden nicht ausgeführt.

Besondere Form der Mehrfachanweisung (CASE-Anweisung)

Neben der `IF`-Anweisung kann bei einer Mehrfachverzweigung auch eine andere Anweisung verwendet werden, und zwar die `CASE`-Anweisung. Bei der `IF`-Anweisung können mehrere Variablen und Bedingungen verwendet werden, bei der `CASE`-Anweisung kann nur eine Variable ausgewertet werden, und es muss sich dabei um eine Variable für Ganzzahlen handeln, zum Beispiel eine Integervariable. Mit der `CASE`-Anweisung kann beispielsweise eine Schrittkette realisiert werden, wobei die von der `CASE`-Anweisung ausgewertete Variable als Schrittzähler fungiert. Das folgende Beispiel zeigt die Nutzung der `CASE`-Anweisung für eine Schrittkette.


```
CASE iIntegerVarStep01 OF
   0:
      Anweisungsteil;
      iIntegerVarStep01 := iIntegerVarStep01 + 1;
   1:
      Anweisungsteil;
      iIntegerVarStep01 := iIntegerVarStep01 + 1;
   3:
      Anweisungsteil;
      iIntegerVarStep01 := iIntegerVarStep01 + 1;
```

```
    4:
       Anweisungsteil;
       iIntegerVarStep01 := iIntegerVarStep01 + 1;
    5:
       Anweisungsteil;
       iIntegerVarStep01 := 0;
END_CASE
```

Bei diesem Beispiel erfolgt die Weiterschaltung zum nächsten Schritt bedingungslos. Dies kommt bei einer »echten« Schrittkette eher selten vor. In den meisten Fällen wird innerhalb eines Schritts die Schrittnummer erst erhöht, wenn ein bestimmtes Ereignis eingetreten ist. Die Abfrage, ob das betreffende Ereignis eingetreten ist, sowie die Erhöhung des Schrittzählers können zum Beispiel über eine bedingte Anweisung erfolgen.

Die Sprünge müssen auch nicht wie im Beispiel mit der Schrittweite eins (`+ 1`) erfolgen, und es kann in einem Schritt auch anstatt einer Berechnung der Schrittnummer eine Schrittnummer direkt angegeben werden. Das kann zum Beispiel erforderlich sein, wenn das Programm im Fehlerfall in einen bestimmten Schritt springen soll.

Um Änderungen, speziell das Einfügen von Schritten, zu erleichtern, sollten als Schritte nicht direkt Zahlen verwendet werden. Bei CODESYS-basierten Steuerungen können Sie sogenannte ENUMs nutzen. ENUMs sind Aufzählungselemente, die einen frei wählbaren Namen haben und als Platzhalter für einen Zahlenwert fungieren. Am Ende dieses Kapitels gehe ich näher auf ENUMs ein. Bei Siemens gibt es ENUMs nicht, hier müssen Sie Konstanten definieren. Konstanten sind Variablen, deren Werte sich nicht ändern lassen.

Was ist nun, wenn mehrere Wege nach Rom führen, sprich: wenn ein Anweisungsteil bei mehreren Zahlenwerten ausgeführt werden soll? Dies ist möglich, was ich mit dem folgenden Beispiel demonstrieren möchte.


```
CASE iIntegerVarStep01 OF
   0,20:
      Anweisungsteil;
   1..16:
      Anweisungsteil;
   17:
      Anweisungsteil;
   18:
      Anweisungsteil;
   19:
      Anweisungsteil;
   ELSE
      Anweisungsteil;
END_CASE
```

In diesem Beispiel wird der erste Anweisungsteil ausgeführt, wenn die Variable `iInteger VarStep01` den Wert 0 oder 20 hat. Bei einem Wert von 1 bis 16 wird der zweite Anweisungsteil ausgeführt. Der dritte Teil wird ausgeführt, wenn die Variable den Wert 17 hat, der vierte bei einem Wert von 18 und der fünfte bei einem Wert von 19.

Wie Sie sicher bemerkt haben, kann auch bei der CASE-Anweisung das Schlüsselwort ELSE genutzt werden. Der Anweisungsteil im ELSE-Zweig wird dann ausgeführt, wenn die Variable einen nicht bei den anderen CASE-Schritten verwendeten Wert hat. Im Falle des obigen Beispiels also kleiner 0 oder größer 20 ist.

Quizfrage: Schauen Sie sich das folgende Beispiel einmal genau an und entscheiden Sie dann, ob es sich hierbei um eine Verzweigung oder eine Mehrfachverzweigung handelt.

```
IF xBoolVar01 OR xBoolVar02 THEN
   Anweisungsteil;
ELSIF xBoolVar03 XOR xBoolVar04 THEN
   Anweisungsteil;
END_IF
```

Sollte Ihre Antwort Verzweigung gelautet haben, so muss ich Sie leider enttäuschen, denn das ist falsch. Eine Verzweigung hat nur einen Zweig, der bei einer erfüllten Bedingung ausgeführt wird, und einen Alternativzweig, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Bei obigem Beispiel gibt es aber zwei Zweige mit einer Bedingung, daher handelt es sich auch hier um eine Mehrfachverzweigung.

Vieles zum Thema Schleifen

Als Letztes in dem Abschnitt zum Thema »Wichtige Programmkonstrukte« möchte ich Ihnen noch Schleifen vorstellen. Es gibt bei den in diesem Buch vorgestellten SPS-Systemen jeweils drei verschiedene Schleifentypen.

Ehe ich mit den Schleifen richtig beginne, muss ich eine Warnung loswerden.

Die Nutzung von Schleifen kann zu Problemen führen, und Sie müssen sich gut überlegen, ob bei der von Ihnen angedachten Nutzung einer Schleife die entsprechenden problematischen Fälle auftreten können. Welche Probleme bei der Nutzung von Schleifen konkret auftreten können, erfahren Sie in Kapitel 2 »Immer schön im Kreis, ohne Pause und gleichmäßig«.

Deshalb hier mein Tipp für Sie: Solange Sie noch nicht so erfahren in der Programmierung von SPSen sind, empfehle ich Ihnen, Schleifen eher zu meiden.

Die FOR-Schleife

Der erste Schleifentyp, den ich Ihnen vorstellen möchte, ist die FOR-Schleife. Zunächst einmal der allgemeine Aufbau einer FOR-Schleife.

```
FOR iIntegerVar01 := Startwert TO Endwert BY Schrittweite DO
   Anweisungsteil;
END_FOR;
```

Nach dem Schlüsselwort FOR muss eine Variable angegeben werden. Die Variable muss den Typ einer Ganzzahlvariablen haben, zum Beispiel INT. Als Nächstes wird der Startwert der Variablen angegeben. Er muss nicht zwingend 0 sein. Danach erfolgt die Angabe des

Endwerts. Die Angabe des Schlüsselworts BY und die der Schrittweite sind optional und nur erforderlich, wenn die Variable bei jedem Durchlauf um mehr als 1 erhöht werden soll. Außerdem müssen das Schlüsselwort BY und die Schrittweite zwingend angegeben werden, wenn der Startwert größer als der Endwert ist und die Variable in negative Richtung zählen soll. Am Ende folgt noch das Schlüsselwort DO.

Hier nun zwei konkrete Beispiele für FOR-Schleifen.


```
FOR iIntegerVar01 := 0 TO 99 DO
   Anweisungsteil;
END_FOR;
FOR iIntegerVar01 := 99 TO 0 BY -1 DO
   Anweisungsteil;
END_FOR;
```

Bei beiden Beispielen wird die Schleife insgesamt 100-mal durchlaufen. Beim ersten Beispiel zählt die Variable dabei von 0 bis 99 hoch und beim zweiten zählt die Variable von 99 bis 0 runter.

Sowohl der Startwert als auch der Endwert müssen nicht als Zahl fest vorgegeben werden, sondern sie können durch Variablen ersetzt werden.

Beim nächsten Beispiel zählt die Variable iIntegerVar01 von 0 bis zum Wert der Variable iIntegerVar02 hoch.


```
FOR iIntegerVar01 := 0 TO iIntegerVar02 DO
   Anweisungsteil;
END_FOR;
```

Bei einer FOR-Schleife steht die Anzahl der Durchläufe beim Start der Schleife aufgrund der Angabe des Start- und des Endwerts fest. Bei den zwei folgenden Schleifentypen ist dies nicht unbedingt der Fall, was ihre Nutzung innerhalb einer SPS problematischer machen kann.

Die WHILE-Schleife

Fangen wir mit der WHILE-Schleife an. Wie schon bei der FOR-Schleife hier zunächst der grundsätzliche Aufbau einer WHILE-Schleife.

```
WHILE Bedingung DO
   Anweisungsteil;
END_WHILE;
```

Gegenüber der FOR-Schleife ist Ihnen sicherlich gleich der grundlegende Unterschied aufgefallen. Anstatt einer Variablen, die hoch- oder heruntergezählt wird, kommt bei der WHILE-Schleife eine Bedingung zum Einsatz. Solange diese Bedingung erfüllt ist, wird die Schleife ausgeführt. Ist die Bedingung beim ersten Start schon nicht erfüllt, wird der Anweisungsteil nicht ausgeführt.

Schleifen, bei denen eine Prüfung vor der Ausführung des Anweisungsteils erfolgt, nennt man auch *kopfgesteuerte Schleifen*, weil die Prüfung im oberen Teil, also dem Kopf, erfolgt. Auch hierzu wieder ein paar Beispiele.


```
WHILE xBoolVar01 DO
   Anweisungsteil;
END_WHILE;
WHILE xBoolVar01 AND iIntegerVar01 < 50 DO
   Anweisungsteil;
END_WHILE;
WHILE xBoolVar01 = FALSE DO
   Anweisungsliste;
END_WHILE;
```

Beim ersten Beispiel wird die Schleife so lange ausgeführt, wie die Variable `xBoolVar01 TRUE` ist. Beim zweiten Beispiel erfolgt die Ausführung, solange die Variable `xBoolVar01 TRUE` und der Wert der Variablen `iIntegerVar01` kleiner 50 ist. Beim dritten Beispiel wiederum wird die Schleife so lange ausgeführt, wie die Variable `xBoolVar01 FALSE` ist.

Die REPEAT-Schleife

Der letzte Schleifentyp ist die `REPEAT`-Schleife. Auch hier zunächst der allgemeine Aufbau der Schleife. Vergleichen Sie diesen bitte einmal mit dem Aufbau der `WHILE`-Schleife.

```
REPEAT
   Anweisungsteil;
UNTIL
   Bedingung;
END_REPEAT;
```

Wenn Sie den Aufbau dieses Schleifentyps mit der `WHILE`-Schleife vergleichen, wird Ihnen auffallen, dass bei der `REPEAT`-Schleife die Bedingung am unteren Ende steht. Aus diesem Grund wird die `REPEAT`-Schleife als *fußgesteuerte Schleife* bezeichnet.

Dieser auf den ersten Blick nur optische Unterschied führt zu einem grundlegenden Unterschied beim Verhalten der Schleifen. Ist die Bedingung bei der `WHILE`-Schleife beim Start nicht erfüllt, dann wird der Anweisungsteil nie ausgeführt. Bei der `REPEAT`-Schleife erfolgt die Prüfung erst nach dem Anweisungsteil, was zur Folge hat, dass dieser mindestens einmal ausgeführt wird.

Hier ein paar Beispiele für `REPEAT`-Schleifen, die denen der `WHILE`-Schleifen entsprechen.


```
REPEAT
   Anweisungsteil;
UNTIL
   xBoolVar01
END_REPEAT;
REPEAT
   Anweisungsteil;
UNTIL
   xBoolVar01 AND iIntegerVar01 < 50
END_REPEAT;
REPEAT
   Anweisungsteil;
```

```
UNTIL
   xBoolVar01 = FALSE
END_REPEAT;
```

Der Anweisungsteil wird beim ersten Beispiel nach der ersten Ausführung so lange wiederholt ausgeführt, wie die Variable `xBoolVar01 TRUE` ist. Beim zweiten Beispiel erfolgt die weitere Ausführung, solange die Variable `xBoolVar01 TRUE` und der Wert der Variable `iIntegerVar01` kleiner 50 ist. Beim dritten Beispiel wiederum erfolgt die weitere Ausführung, solange die Variable `xBoolVar01 FALSE` ist.

Welcher Schleifentyp der jeweils geeignete ist, hängt von der Anwendung ab, für die der jeweilige Typ eingesetzt werden soll.

Was sind ENUMs?

Hier nun noch die versprochene Einführung in die Funktionsweise von ENUMs.

Weiter oben im Abschnitt »Besondere Form der Mehrfachanweisung (CASE-Anweisung)« habe ich Ihnen die Funktionsweise der `CASE`-Anweisung erklärt und dort beiläufig erwähnt, dass Sie anstelle von Zahlen für die einzelnen `CASE`-Anweisungen oder -Schritte ENUMs nehmen können. Hier erkläre ich Ihnen nun genauer, was ENUMs sind und wie man sie verwendet.

Wie schon erwähnt sind ENUMs Aufzählungselemente, also eine Liste mit mehreren Elementen. Jedes Element hat dabei einen Namen und einen Wert.

In Abbildung 1.18 sehen Sie ein definiertes ENUM in der Entwicklungsumgebung.

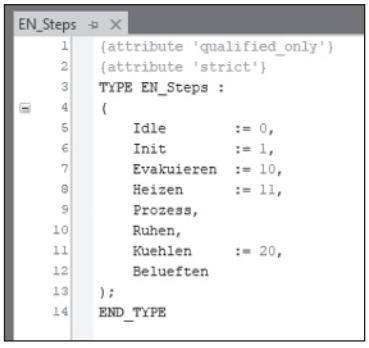


Abbildung 1.18: Definition eines ENUMs

Das ENUM hat acht Einträge, und allen Einträgen wurde auf unterschiedliche Weise jeweils ein Wert zugewiesen, auch wenn das für Sie auf den ersten Blick nicht so aussieht.

Dem Eintrag `Idle` ist der Wert 0 zugewiesen worden, dem Eintrag `Init` die 1, dem Eintrag `Evakuieren` der Wert 10 und dem Eintrag `Heizen` die 11.

Den nun folgenden Einträgen `Prozess` und `Ruhen` scheint kein Wert zugewiesen worden zu sein, denn da steht ja nur der Eintrag selbst ohne Zuweisung. Man kann hier von einer *unsichtbaren Zuweisung* sprechen, denn auch in diesen Fällen findet eine Zuweisung statt. Gibt man explizit keinen Wert vor, erhöht die Entwicklungsumgebung den Wert automatisch um 1. Im Falle dieses Beispiels bekommt also der Eintrag `Prozess` den Wert 12 und der Eintrag `Ruhen` den Wert 13 zugewiesen.

Als Nächstes folgt der Eintrag `Kuehlen` mit dem nun wieder aktiv zugewiesenen Wert 20.

Quizfrage: Welchen Wert bekommt der Eintrag `Belueften` zugewiesen?

Genau, gut aufgepasst, der Eintrag erhält von der Entwicklungsumgebung automatisch den Wert 21 zugewiesen.

Schauen Sie sich bitte einmal die Abbildung 1.19 genauer an.

EN_Steps

```
1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE EN_Steps :
4  (
5      Idle        := 0,
6      Init        := 1,
7      Evakuieren  := 10,
8      Heizen      := 11,
9      Prozess,
10     Ruhen,
11     Kuehlen     := 20,
12     Belueften
13 ) INT;
14 END_TYPE
```

Abbildung 1.19: Definition eines ENUMs mit zusätzlichem Variablentyp

Auf den ersten Blick sieht die Definition dieses ENUMs genauso aus wie die in Abbildung 1.18 gezeigte Definition des ENUMs. Aber auf den zweiten Blick erkennen Sie sicher den Unterschied. Am Ende der Definition steht hinter der schließenden Klammer das Wort `INT`. Aber was hat es da verloren und was bewirkt es?

Ein ENUM ist, wie eine Variable vom Typ Integer auch, ein Datentyp, allerdings im Gegensatz zum Integer kein Standarddatentyp, sondern ein *benutzerdefinierter Datentyp*. Er kann nicht durch eine Typkonvertierung, deren Funktion ich bereits im Abschnitt »Informationen zur Typkonvertierung« erklärt habe, mit anderen Datentypen genutzt werden. In den meisten Fällen ist dies nicht weiter störend. Aber es kann sein, dass die Nutzung der Werte mit anderen Variablen erforderlich ist, wie zum Beispiel bei der Übertragung von Daten über das Netz.

Zu diesem Zweck kann man den Werten der Einträge, die normalerweise nur vom Typ des ENUMs sind, einen zusätzlichen Typ zuweisen. Im Falle des Beispiels in Abbildung 1.19 ist dies der Typ Integer.

Weitere Objekte, beispielsweise Funktionen

Als Letztes möchte ich Ihnen neben Variablen noch weitere Objekte vorstellen, die Ihnen im Bereich der SPS immer wieder begegnen werden, und Ihnen deren Funktionsweise und Besonderheiten erklären.

Hier zunächst eine Liste der Objekte, die Sie im Folgenden kennenlernen werden:

- ✔ Programm
- ✔ Funktionsbaustein
- ✔ Funktion
- ✔ Datenbaustein (Erklärung erfolgt im Abschnitt über die Funktionsbausteine)

Wissenswerts zum Objekt »Programm«

Das Objekt »Programm« gibt es bei CODESYS-basierten Steuerungen, aber nicht bei Siemens, das Objekt »Datenbaustein« wiederum bei Siemens und nicht bei Codesys basierten Steuerungen.

Ein Programm ist bei CODESYS-basierten Steuerungen das oberste ausführbare Objekt. Ein Programm existiert nur einmal und muss nicht instanziiert werden. Was diese beiden Aussagen genau bedeuten, werde ich bei der Erklärung der Funktionsbausteine erläutern.

In Abbildung 1.20 ist ein Beispiel für ein Programm zu sehen.

PRG_Zaehlen

```
1  PROGRAM PRG_Zaehlen
2  VAR_INPUT
3      xCount      : BOOL;
4  END_VAR
5
6  VAR_OUTPUT
7      uiValue     : UINT;
8  END_VAR
9
10 VAR
11     fbRT_Count  : R_TRIG;
12 END_VAR
```

```
1  fbRT_Count(CLK := xCount);
2
3  IF fbRT_Count.Q THEN
4      uiValue := uiValue + 1;
5  END_IF
```

Abbildung 1.20: Das Objekt »Programm«

Dieses Programm erhöht die Variable `uiValue` um den Wert 1, wenn an der Variablen `xCount` eine steigende Flanke, also ein Übergang von FALSE nach TRUE erkannt wurde.

An Programme können Daten, zum Beispiel über einzelne Variablen, übergeben werden, und Programme können auch Daten zurückgeben. Beides ist aber nur sinnvoll, wenn dieses Programm von einem anderen Programm, einem Funktionsbaustein oder einer Funktion aufgerufen wird.

Sollen an ein Programm Daten über einzelne Variablen übergeben werden, müssen diese unter `VAR_INPUT` deklariert werden und heißen *Eingangsvariablen*. Variablen, die Daten von einem Programm zurückgegeben sollen, müssen unter `VAR_OUTPUT` deklariert werden und heißen *Ausgangsvariablen*.

Sollen Daten an ein Programm übergeben, von diesem bearbeitet und wieder zurückgegeben werden, wären zum Datenaustausch zwei Variablen nötig, wenn es nur diese beiden Variablengruppen gäbe. Und Sie erraten es: Es gibt aber noch eine weitere Gruppe, mit der Daten an ein Programm übergeben, vom Programm bearbeitet und wieder zurückgegeben werden können. Diese Gruppe wird mit `VAR_IN_OUT` eingeleitet, und Variablen in dieser Gruppe werden *Ein-/Ausgangsvariablen* genannt.

In Abbildung 1.21 ist ein Programm zu sehen, das die beschriebenen Variablengruppen nutzt. Zusätzlich gibt es noch eine Gruppe für sogenannte lokale Variablen, also Variablen, die nur innerhalb des Programms genutzt werden.

Bei diesem Programm wird zu dem übergebenen Wert der Variablen `udiSum` der Wert der übergebenen Variablen `uiValue` hinzuaddiert, wenn die Variable `xAdd` TRUE ist. Das Ergebnis wird im selben Schritt der lokalen Variablen `udiResult` übergeben. Ist `xAdd` FALSE, dann wird die Variable `udiSum` direkt der Variablen `udiResult` übergeben. Anschließend wird der Variablen `udiSum` der Wert der Variablen `udiResult` zugewiesen. Dieser Wert wird am Ende des Programms an das Objekt zurückgemeldet, das das Programm aufgerufen hat. Am Schluss wird festgestellt, ob der Wert der Variablen `udiSum` gerade oder ungerade ist. Falls er gerade ist, wird die Ausgangsvariable `xEven` auf TRUE gesetzt, ansonsten auf FALSE.

PRG_Rechnen

```
PROGRAM PRG_Rechnen
VAR_INPUT
    xAdd        : BOOL;
    uiValue     : UINT := 2;
END_VAR

VAR_OUTPUT
    xEven       : BOOL;
END_VAR

VAR_IN_OUT
    udiSum      : UDINT;
END_VAR

VAR
    udiResult   : UDINT;
END_VAR
```

```
IF xAdd THEN
    udiResult   := udiSum + UINT_TO_UDINT(uiValue);
ELSE
    udiResult   := udiSum;
END_IF

udiSum  := udiResult;

xEven   := (udiSum MOD 2 = 0);
```

Abbildung 1.21: Programm mit verschiedenen Variablengruppen

Beim Aufruf eines Programms müssen nicht zwingend alle Eingangsvariablen angegeben werden. Auch müssen nicht alle Ausgangsvariablen verwendet werden. Weder die Eingangs- noch die Ausgangsvariablen müssen direkt beim Aufruf des Programms angegeben werden. Was dies genau bedeutet, werde ich Ihnen gleich anhand eines Beispiels für den Aufruf eines Programms zeigen.

Die Ein-/Ausgangsvariablen bilden hier eine Ausnahme. Sie müssen zwingend beim Aufruf des Programms angegeben werden. Außerdem dürfen keine festen Werte an sie übergeben werden, sondern nur Variablen. Der Grund leuchtet ein: Auch wenn der Wert im Programm nicht geändert wird, wird dieser beim Beenden des Programms wieder geschrieben.

In Abbildung 1.22 sind drei Aufrufe des in Abbildung 1.21 vorgestellten Programms zu sehen.

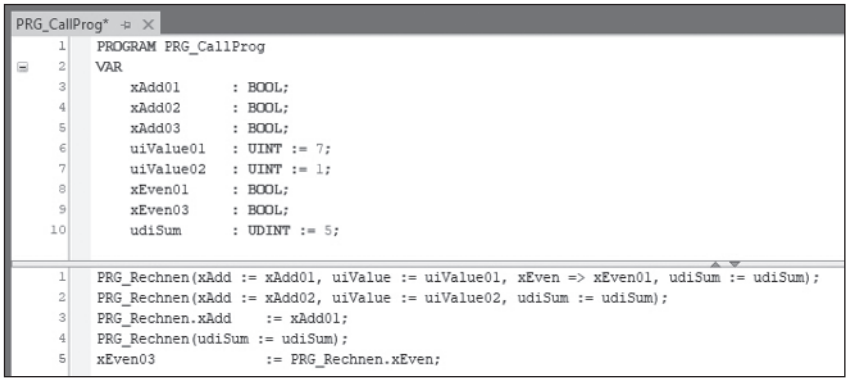


Abbildung 1.22: Aufruf eines Programms

Der erste Aufruf des Programms erfolgt in Zeile 1, wobei bei diesem Beispiel die Eingangs- und Ausgangsvariablen und auch Ein-/Ausgangsvariablen direkt beim Aufruf übergeben werden.

Die Eingangsvariablen bei diesem Beispiel sind die Variablen `xAdd` und `uiValue`, die Ausgangsvariable ist `xEven` und die Ein-/Ausgangsvariable ist die `udiSum`.

Die Übergabe von Werten an Eingangsvariablen erfolgt wie bei der Zuweisung von Werten an einzelne Variablen mit dem Zuweisungsoperator `:=`. Die Werte von Ausgangsvariablen werden beim Aufruf des Programms mit dem Operator `=>` an Variablen übergeben. Die Übergabe von Ein-/Ausgangsvariablen, die ja zwingend direkt beim Aufruf angegeben werden müssen, erfolgt über denselben Operator wie bei Eingangsvariablen.

In Zeile 2 erfolgt ein weiterer Aufruf des Programms, wobei hier die Ausgangsvariable `xEven` nicht genutzt wird.

Das letzte Beispiel des Aufrufs eines Programms erstreckt sich über die Zeilen 3 bis 5. Damit soll gezeigt werden, was mit der obigen Aussage gemeint ist: dass nämlich Ein- und Ausgangsvariablen nicht direkt beim Aufruf angegeben werden müssen.

Der Aufruf des Programms erfolgt bei diesem Beispiel in Zeile 4. Davor, also in Zeile 3, wird der Eingangsvariablen `xAdd` des Programms der Wert der lokalen Variablen `xAdd03` des Objekts übergeben, das das Programm aufgerufen hat.

Dann erfolgt in Zeile 4, wie erwähnt, der Aufruf des Programms.

In Zeile 5 wird der Wert der Ausgangsvariablen `xEven` des Programms der lokalen Variablen `xEven03` des Objekts zugewiesen, das das Programm aufgerufen hat. Sicher ist Ihnen bei dieser Zuweisung gleich etwas aufgefallen. Beim direkten Aufruf des Programms steht die Ausgangsvariable auf der linken Seite und die Variable, der der Wert der Ausgangsvariablen zugewiesen werden soll, auf der rechten Seite. Wenn Sie die Ausgangsvariable außerhalb des Programmaufrufs nutzen, ist die Reihenfolge andersherum. Erst steht die Variable, der der Wert zugewiesen werden soll, dann folgt der Zuweisungsoperator »:=« und schließlich die Ausgangsvariable.

Bei Programmen behalten alle Variablen ihren Wert auch nach Ausführung des Programms bei. Daher kann die Zuweisung der Ausgangsvariablen auch außerhalb der Ausführung des Programms einer Variablen zugewiesen werden.

Neben der Zuweisung der Ausgangsvariablen können sie auch direkt zum Beispiel bei einer bedingten Anweisung genutzt werden.

Sie haben sicher gemerkt, dass beim dritten Beispiel die Eingangsvariable `uiValue` nicht verwendet wurde. Aber welche Konsequenzen hat dies? Wird, selbst wenn die Variable `xAdd` TRUE ist, nichts addiert, oder wird 2 addiert, weil im Programm der Variablen `uiValue` dieser Wert bei der Deklaration als »Startwert« angegeben wurde? Die Antwort lautet weder noch.

Bei Programmen behalten die Variablen ihren Wert auch nach der Ausführung noch bei. Beim zweiten Aufruf des Programms wurde der Eingangsvariablen `uiValue` der Wert der Variablen `uiValue02` zugewiesen. Sie hatte zu dem Zeitpunkt den Wert 1. Wie gesagt behalten die Variablen ihren Wert. Und so hat die Variable `uiValue`, auch wenn sie nicht verwendet wurde, noch den Wert 1. Er würde beim dritten Aufruf zur Variablen `udiSum` addiert werden, wenn die Eingangsvariable `xAdd` TRUE wäre.

Funktionsbausteine verstehen

In diesem Abschnitt teile ich Ihnen Wissenswerts zum Thema Funktionsbausteine mit. Dabei behandele ich Steuerungen, die auf CODESYS basieren und Steuerungen von Siemens mit TIA separat.

Funktionsbausteine im CODESYS-Universum

Ich beginne auch hier wieder mit der Erklärung für CODESYS-basierte Steuerrungen. In Abbildung 1.23 ist ein Funktionsbaustein zu sehen.

FB_Rechnen

```
FUNCTION_BLOCK FB_Rechnen
VAR_INPUT
    rZahl01 : REAL;
    rZahl02 : REAL;
    rZahl03 : REAL := 5.0;
    rZahl04 : REAL;
END_VAR

VAR_OUTPUT
    rErgebnis   : REAL;
END_VAR

VAR
    rZwischenergebnis   : REAL;
END_VAR
```

```
rZwischenergebnis   := rZahl01 + rZahl02;
rErgebnis           := (rZwischenergebnis + rZahl03) * rZahl04;
```

Abbildung 1.23: Funktionsbaustein bei CODESYS-basierter Steuerung

Bei diesem Beispiel werden die Eingangsvariablen `rZahl01` und `rZahl02` addiert und das Ergebnis der lokalen Variablen `rZwischenergebnis` zugewiesen. Anschließend wird zum Wert der lokalen Variablen `rZwischenergebnis` der Wert der Eingangsvariablen `rZahl03` addiert. Das Ergebnis wird mit dem Wert der Eingangsvariablen `rZahl04` multipliziert und dann der Ausgangsvariablen `rErgebnis` zugewiesen.

Sie werden mich jetzt vielleicht fragen wollen, wo denn da, vom Schlüsselwort `FUNCTION_BLOCK` mal abgesehen, der Unterschied zum Programm ist? Und Sie haben recht, es gibt fast keinen Unterschied zu Programmen. Auch Funktionsbausteine haben dieselben Variablengruppen wie Programme, also Eingangs- und Ausgangsvariablen, Ein-/Ausgangsvariablen sowie lokale Variablen. Außerdem behalten auch bei Funktionsbausteinen die Variablen ihren Wert nach der Ausführung bei.

Der Unterschied zu Programmen besteht bei Funktionsbausteinen darin, dass ein Funktionsbaustein mehrfach existieren kann. Ein Programm kann mehrfach aufgerufen werden, aber dabei handelt es sich immer um dasselbe Programm mit demselben Namen sowie mit Variablen mit denselben Werten. Bei Funktionsbausteinen kann es von einem Funktionsbaustein mehrere Kopien geben. Aber wie?

Denken Sie bitte einmal an meine Erklärungen zum Thema Variablen zurück. Es gibt verschiedene Variablentypen, zum Beispiel die Integervariable. Von jedem Typ kann es jedoch nicht nur eine Variable geben, sondern es können mehrere Variablen eines Typs angelegt werden. Man spricht hier von der Deklaration von Variablen.

Instanziierung von Funktionsbausteinen

Bei Funktionsbausteinen ist der Vorgang ähnlich wie bei Variablen, nur dass man in diesem Fall nicht von Deklaration, sondern von Instanziierung spricht. Es wird eine Instanz eines Funktionsbausteins angelegt. Konkret heißt dies, dass jede Instanz eines Funktionsbausteins einen eigenen Speicherbereich für die verwendeten Variablen zugewiesen bekommt. Der eigentliche Programmcode existiert aber unabhängig von der Anzahl der Instanzen nur einmal. Ihm wird bei der Ausführung mitgeteilt, welchen Speicherbereich er für die jeweilige Instanz nutzen soll.

In Abbildung 1.24 ist ein Beispiel für zwei Instanzen des Funktionsbausteins (Abkürzung: *FB*) `FB_Rechnen` zu sehen. Den Instanzen wurden die Namen `fbRechnen01` und `fbRechnen02` vergeben.

PRG_CallFB

```
1  PROGRAM PRG_CallFB
2  VAR
3      fbRechnen01 : FB_Rechnen;
4      fbRechnen02 : FB_Rechnen;
5      rErgebnis01 : REAL;
6      rErgebnis02 : REAL;
7  END_VAR
```

```
1  fbRechnen01(rZahl01 := 2.0, rZahl02 := 7.0, rZahl03 := 1.0, rZahl04 := 10.0, rErgebnis => rErgebnis01);
2  fbRechnen02.rZahl01 := 2.0;
3  fbRechnen02.rZahl02 := 7.0;
4  fbRechnen02.rZahl04 := 10.0;
5  fbRechnen02();
6  rErgebnis02 := fbRechnen02.rErgebnis;
```

Abbildung 1.24: Nutzung von FBs bei CODESYS-basierter Steuerung

Ausführung von Funktionsbausteinen

In Zeile 1 wird die erste Instanz des FBs ausgeführt. Dabei werden sämtlichen Eingangsvariablen feste Werte zugewiesen und der Wert der Ausgangsvariablen der Variable `rErgebnis01` zugewiesen.

Der Aufruf der zweiten Instanz erstreckt sich über die Zeilen 2 bis 6, wobei der eigentliche Aufruf des Funktionsbausteins in Zeile 5 stattfindet. Wie schon bei der Erklärung des Objekts »Programm« erfolgt bei diesem Beispiel die Übergabe der Variablen nicht beim Aufruf.

In Zeile 2 bis 4 werden den Eingangsvariablen der Instanz des Funktionsbausteins Werte übergeben. Dann wird diese Instanz des FBs in Zeile 5 aufgerufen und anschließend in Zeile 6 der Wert der Ausgangsvariablen des FBs der Variablen `rErgebnis02` zugewiesen.

Sie haben sicher bemerkt, dass bei `fbRechnen02` der Eingangsvariablen `rZahl03` kein Wert zugewiesen wurde. Welchen Wert wird diese Variable in diesem Fall bekommen?

Sollte Ihre Antwort 1.0 lauten, da in Zeile 1 der Variablen dieser Wert ja zugewiesen wurde, dann liegen Sie leider falsch. Im Gegensatz zum Objekt Programm, das nur einmal existiert, können von FBs mehrere sogenannte Instanzen existieren, was hier der Fall ist. Jede Instanz besitzt ihren eigenen Speicherbereich, und somit wirkt sich die Zuweisung des Werts 1.0 der Variablen `rZahl03` der Instanz `fbRechnen01` nicht auf die Instanz `fbRechnen02` aus.

Die richtige Antwort lautet hier 5. Die Variable wurde bisher nicht genutzt, daher hat sie noch ihren Initialwert, der eben 5 oder ganz genau 5.0 ist.

So weit die Einführung in Funktionsbausteine bei CODESYS-basierten Steuerungen.

Funktionsbausteine im Siemens-Universum

Wenden wir uns nun der Nutzung von Funktionsbausteinen in Siemens TIA zu. Bei Siemens TIA erfolgt die Implementation ähnlich wie bei CODESYS-basierten Steuerungen. In Abbildung 1.25 ist eine Implementierung eines FBs zu sehen, der dieselbe Funktionalität hat wie das Beispiel in Abbildung 1.23 für CODESYS-basierte Steuerungen.

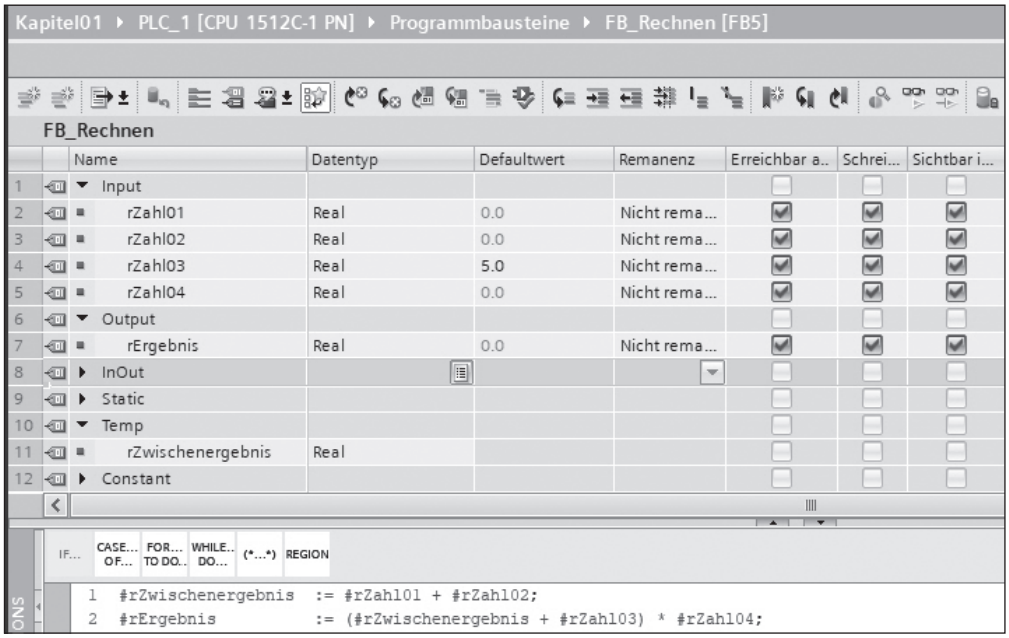


Abbildung 1.25: Funktionsbaustein bei TIA

So weit sind sich CODESYS-basierte Steuerungen und Siemens TIA ähnlich, aber jetzt kommt ein Unterschied.

Instanziierung von Funktionsbausteinen

Bei CODESYS-basierten Steuerungen erstellen Sie vom FB eine Instanz in dem Objekt, in dem Sie die Instanz nutzen möchten, oder in einer globalen Variablenliste. Dann können Sie von mehreren Stellen auf den FB zugreifen. Alle Instanzen nutzen denselben Code, aber jeweils ihren eigenen Speicherbereich.

Bei Siemens TIA werden die Daten einer Instanz eines FBs in einem sogenannten Datenbaustein abgebildet, der angelegt werden muss. Es gibt dabei zwei Möglichkeiten:

1. den *Instanz-DB* und
2. den *Multiinstanz-DB*.

Der Instanz-DB ist ein extra Baustein, ähnlich wie der Funktionsbaustein selbst. Der DB enthält alle im FB deklarierten Variablen. In Abbildung 1.26 ist ein Instanz-DB für den Funktionsbaustein `FB_Rechnen` abgebildet.

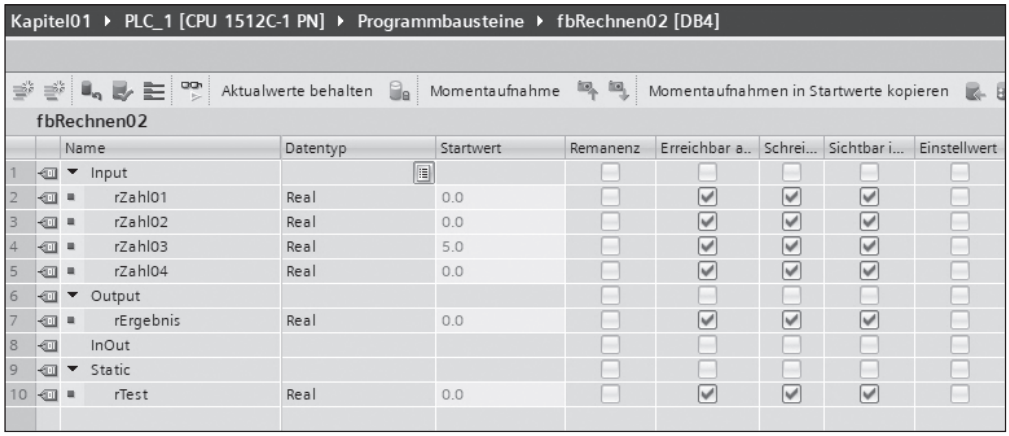


Abbildung 1.26: Instanz-DB

Der Vorteil eines Instanz-DBs liegt darin, dass auf diesen DB und somit auch auf den dazugehörigen FB von mehreren Stellen zugegriffen werden kann, ähnlich wie auf globale Variablen. Es gilt allerdings auch hier wieder die Warnung, die ich bei der Erklärung der globalen Variablen ausgesprochen habe: »So viel wie nötig, so wenig wie möglich«.

Eine andere Möglichkeit, eine Instanz eines FBs unter TIA zu erzeugen, ist, wie schon erwähnt, der sogenannte Multiinstanz-DB. Ein Multiinstanz-DB wird ähnlich angelegt wie eine Instanz eines FBs bei CODESYS-basierten Steuerungen.

In Abbildung 1.27 ist die Deklaration eines solchen Multiinstanz-DBs zu sehen, der in diesem Beispiel den Instanznamen `fbRechnen01` zugewiesen bekommen hat.

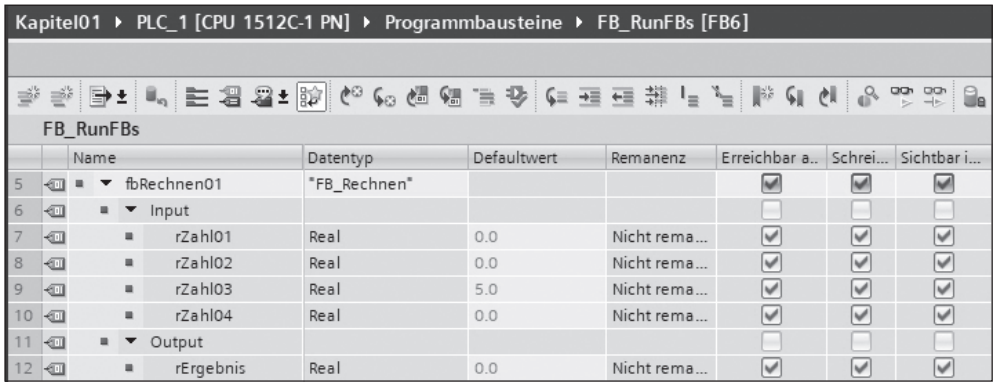


Abbildung 1.27: Multiinstanz-DB

In der Abbildung 1.28 ist ein Beispiel für die Verwendung eines Multiinstanz-DBs und eines Instanz-DBs zu sehen.

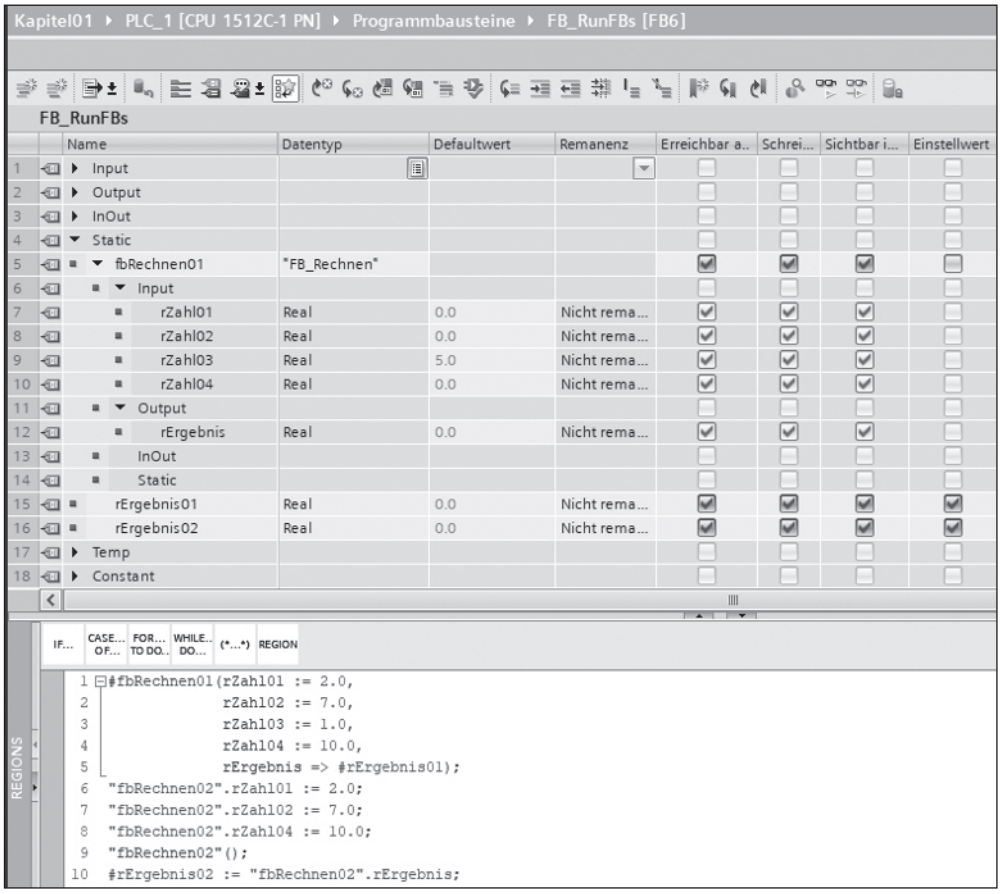


Abbildung 1.28: Nutzung von FBs bei Siemens TIA

Vieles zum Thema Funktionen

Nun folgt die Erklärung des Objekts »Funktion«. In Abbildung 1.29 ist ein Beispiel einer Funktion bei einer CODESYS-basierten Steuerung abgebildet.

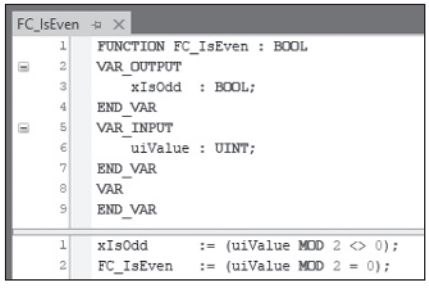


Abbildung 1.29: Funktion bei CODESYS-basierter Steuerung

Der Aufbau ist ähnlich wie bei den Objekten »Programm« und »Funktionsbaustein«. Ich bin mir aber sicher, dass Ihnen zwei Unterschiede sofort aufgefallen sind.

Der erste Unterschied ist das Schlüsselwort am Anfang des Deklarationsteils. Dieses lautet bei Funktionen `FUNCTION`. Der nächste Unterschied besteht darin, dass hinter dem Funktionsnamen ein Variablentyp angegeben ist. In diesem Beispiel `BOOL`.

Bei einem Funktionsbaustein werden Variablen über Ausgangsvariablen übergeben. Bei einer Funktion besteht diese Möglichkeit auch, wie Sie am Beispiel in Abbildung 1.29 sehen. Bei einer Funktion gibt es aber noch eine weitere Möglichkeit, wofür die Angabe des Variablentyps erforderlich ist. Wie diese andere Möglichkeit genau funktioniert, werde ich Ihnen im Abschnitt »Ausführung von Funktionen« erklären, in dem ich auf die Verwendung von Funktionen eingehe.

Gegenüber Funktionsbausteinen und Programmen gibt es bei Funktionen zwei grundlegende Unterschiede, die Sie in diesem Abschnitt kennenlernen:

1. Von einem Funktionsbaustein kann es mehrere unabhängige Instanzen mit eigenem Speicherbereich geben. Eine Funktion existiert, ähnlich wie ein Programm, nur einmal und muss dadurch auch nicht instanziiert werden. Der Aufruf einer Funktion erfolgt, wie der Aufruf eines Programms auch, über den bei der Implementierung vergebenen Namen der Funktion.

2. Bei einem Programm und bei einem Funktionsbaustein bleibt der Speicher auch nach der Ausführung des jeweiligen Objekts bestehen und somit auch die darin enthaltenen Variablen, auf die, auch nach Beendigung des Programms oder des Funktionsbausteins, immer noch zugegriffen werden kann. Bei einer Funktion ist dies anders. Hier gibt es verschiedene Varianten. Bei allen Varianten erhält eine Funktion bei jedem Aufruf einen Speicherbereich zugewiesen, wobei dieser meist an einer anderen Stelle liegt als beim vorhergehenden Aufruf. Bei einer Variante werden die in der Funktion deklarierten Variablen bei jedem Aufruf initialisiert, also auf einen definierten Startwert gesetzt. Bei einer anderen Variante enthält der Speicherbereich noch die Daten aus einer vorhergehenden Nutzung durch andere Objekte und damit »erhalten« die in der Funktion deklarierten Variablen diese Werte. Man kann somit sagen, dass die Variablen Zufallswerte erhalten.

Nach der Ausführung wird der Speicher wieder freigegeben. Dieser Umstand hat zur Folge, dass eine Ausgangsvariable der Funktion nur beim Aufruf der Funktion einer Variablen zugewiesen werden kann, aber nicht, wie bei einem Funktionsbaustein, auch danach noch.

Ausführung von Funktionen

In Abbildung 1.30 ist der Aufruf einer Funktion zu sehen.

PRG_CallFC

```
1  PROGRAM PRG_CallFC
2  VAR
3      xIsEven01   : BOOL;
4      xIsOdd01    : BOOL;
5      xIsOdd02    : BOOL;
6  END_VAR
```

```
1  xIsEven01   := FC_IsEven(uiValue := 4, xIsOdd => xIsOdd01);
2  FC_IsEven(uiValue := 3, xIsOdd => xIsOdd02);
```

Abbildung 1.30: Aufruf einer Funktion bei CODESYS-basierter Steuerung

Bei diesem Beispiel wird die Funktion `FC_IsEven` von Abbildung 1.29 zweimal ausgeführt.

Bei der ersten Ausführung wird der Wert der Ausgabevariablen der Funktion der lokalen Variablen `xIsEven01` zugewiesen und der Wert der Ausgangsvariablen `xIsOdd` der lokalen Variablen `xIsOdd01`. Bei der zweiten Ausführung wird die Ausgabevariable nicht verwendet, und die Ausgangsvariable wird hier der lokalen Variablen `xIsOdd02` zugewiesen.

Bei Funktionen werden bei einer Variante alle Variablen bei jeder Ausführung der Funktion neu initialisiert. Werden in einer Funktion Funktionsbausteine instanziiert und verwendet, gilt für diese dasselbe. Diese haben bei jedem Aufruf der Funktion den Zustand, als wären sie noch nie aufgerufen worden.

Soweit ein Funktionsbaustein innerhalb eines Zyklus abgearbeitet wird, stellt dies kein Problem dar, aber es gibt Funktionsbausteine, die mehrmals über mehrere Zyklen aufgerufen werden müssen, zum Beispiel Funktionsbausteine für das Senden und Empfangen von Daten. Derartige Bausteine dürfen nicht in Funktionen deklariert werden, auch wenn die Entwicklungsumgebung in einem solchen Fall beim Übersetzen nicht meckert.

In Kapitel 2 »Immer schön im Kreis, ohne Pause und gleichmäßig« erfahren Sie, was Zyklen sind.

So, das hier (hoffentlich erfolgreich) vermittelte Wissen sollte ausreichen, dass Sie bezüglich der Programmierung inhaltlich fit genug sind, um den Ausführungen in den folgenden Kapiteln leicht folgen zu können.