# Einführung in SCL

## Step 7

**V1.0**


**Autor:**
**JSEngineering**
Jens Schürmann
Dipl.-Ing. (FH)
Veilchenweg 4
D-26209 Hatten

Kontakt:
www.jsengineering.de
info@jsengineering.de
+49 4481 9059863

# Inhalt

# 1 Vor- und Nachteile von SCL

## 1.1 Vorteile

- Strukturierte Programmierung ähnlich einer Hochsprache
- Programmierer ist sehr frei in der Gestaltung des Programms/Quelltextes
- Je nach Anwendung und Programmierer übersichtlicher als große Symbole in FUP
- Kann ausgiebig kommentiert werden
- Durchgehendes Programm ohne Einteilung in Netzwerke
- Anstatt bildschirmfüllender Symbole in FUP/KOP/CFC kann viel Quelltext strukturiert auf einen Blick erfaßt werden.
- Bausteine können mit weiterreichenden Attributen belegt werden, als in FUP/KOP/AWL, z.B. KnowHow-Schutz.
- Sehr gut geeignet für Datenhandling, Schleifen, mathematische Funktionen
- Es gibt Entwicklungstools, die logische Programmabläufe als Diagramm darstellen, um Programmierfehler zu minimieren. Ausgabe als SCL.

## 1.2 Nachteile

- Teilweise Probleme mit den Referenzdaten
- Keine Verlinkung aufgerufener Bausteine
- Kein Drag&Drop für Bausteine
- Teilweise Probleme bei der Beobachtung
- Bei unerfahrenen Programmierern kann es schnell zu unübersichtlichem Spaghetti-Code kommen.

## 1.3 Resumé

In der Praxis hat sich gezeigt, daß aufgrund der Nachteile – die hauptsächlich in der Entwicklungsumgebung Simatic S7 liegen – es nicht sinnvoll ist, komplette Programme in SCL zu schreiben.

Die Benutzung von SCL sollte auf Grundfunktionen beschränkt werden, die wenig geändert, aber häufig aufgerufen werden. Das Hauptprogramm mit allen Aufrufen sollte in FUP/KOP/CFC geschrieben sein.

Für die Entwicklung von Grundbausteinen, besonders im datenverarbeitenden Bereich ist SCL sehr gut geeignet.

# 2 Der Editor

Der Editor wird aufgerufen, sobald Sie eine vorhandene SCL-Quelle öffnen.

## 2.1 Wichtige Einstellungen

Bevor Sie anfangen, den Editor das erste Mal zu benutzen, sollten Sie unter Extras → Einstellungen... die nachfolgend dargestellten Optionen auswählen.

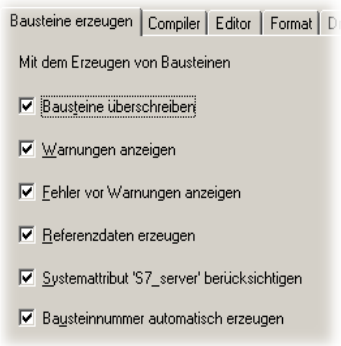



Abbildung 1: Optionen für die Generierung von Bausteinen einstellen

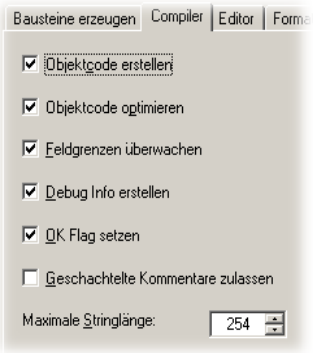



Abbildung 2: Optionen für die Übersetzung einstellen

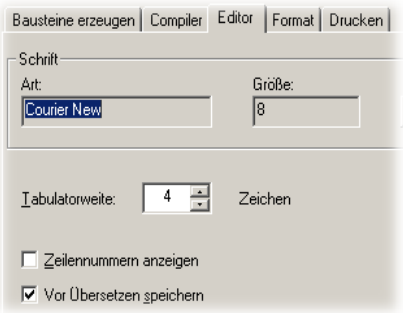



Abbildung 3: Wichtig: Speichern vor dem Übersetzen

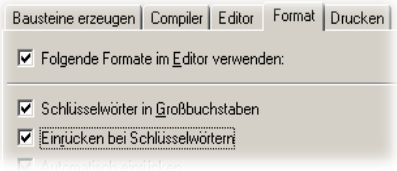



Abbildung 4: Darstellungsformat

## 2.2 Die Symbolleiste

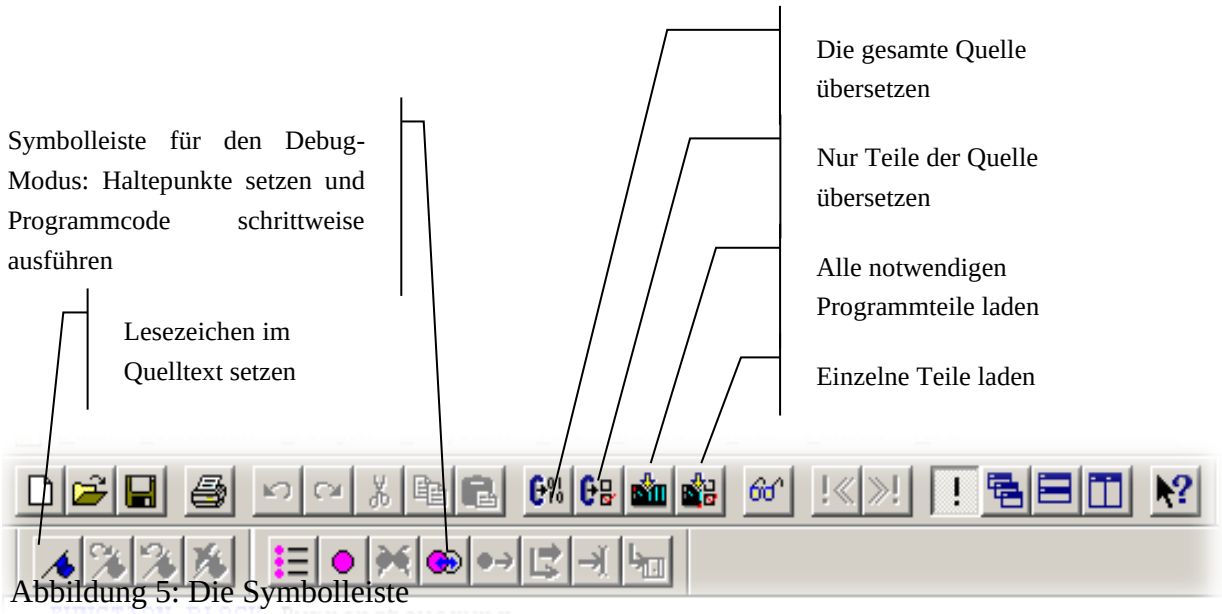



Abbildung 5: Die Symbolleiste

# 3 Ein neues Programm erstellen

Um ein Programm oder einen Baustein zu erstellen, muß mindestens eine Quelle erstellt werden.

Das gesamte Programm kann in *einer* Quelle geschrieben werden. Wenn aber nur einzelne Programmteile bearbeitet werden sollen oder wenn es ein größeres Programm ist, wird dieses unersichtlich.

Die Quelle kann beliebig benannt werden. Sucht man eine bestimmt Funktion in einem umfangreicheren Programm, kann dieses schwierig werden, wenn die Quellen nicht eindeutig benannt sind.

Daher folgende Regeln

- Für *jeden* Baustein, den man mit einer Quelle erzeugen möchte, eine *eigene* Quelle benutzen.
- Die Quelle so benennen, wie den Baustein, den man erzeugt.

## 3.1 Eine Quelle erzeugen

Egal was man mit SCL erzeugen möchte: OB, FB, FC, DB, UDT – man benötigt eine neue Quelle:

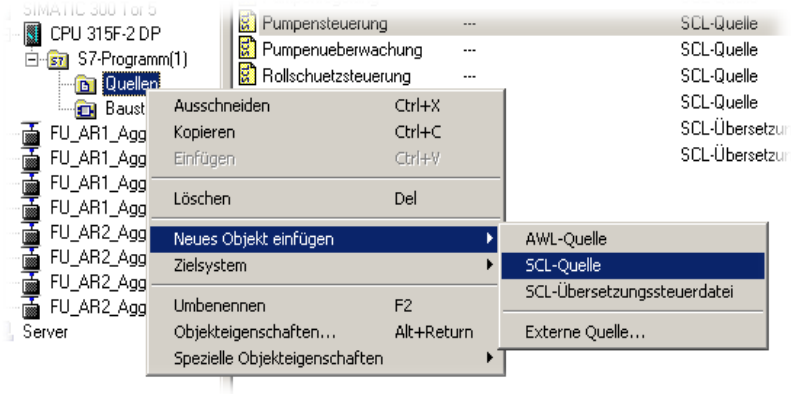



Abbildung 6: Eine neue SCL-Quelle erzeugen

## 3.2 Die Struktur einer Quelle

Wenn Sie die neu erzeugte Quelle öffnen, sehen Sie ein leeres Blatt im Editor. Editor deshalb, weil SCL eine komplett geschriebene Programmiersprache ist, ähnlich einer Hochsprache wie C, Pascal oder Basic.

Die effektivste Art, einen Baustein zu programmieren, ist, sich der von SCL bereitgestellten Vorlagen zu bedienen:

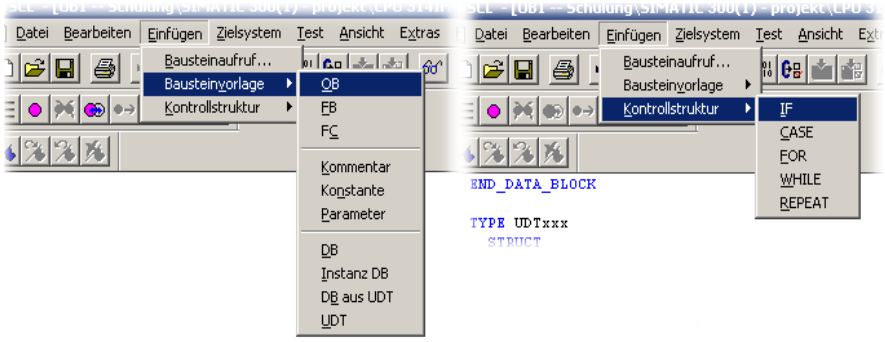



Abbildung 7: Vorlagen in SCL

## 3.3 Das Prinzip einer strukturierten Programmiersprache

Eine strukturierte Programmiersprache basiert auf einer schriftlich verfaßten Textlogik, die im Editor eingegeben wird. Das eingegebene Programm wird übersetzt. Erst jetzt ist daraus ein Baustein geworden, der in die SPS geladen werden kann.

Die Grundelemente einer Funktion sind

- Funktionsaufruf
- Funktionskopf
- Funktionsrumpf

Der Funktionsrumpf seinerseits besteht aus

- Kontrollstrukturen
- Weiteren Funktionsaufrufen
- Zuweisungen
- Abfragen

Damit beim Übersetzen der Interpreter weiß, welche Teile zusammengehören (beispielsweise, ob *eine* oder *mehrere* Anweisungen nach einer Bedingung ausgeführt werden), gibt es immer ein einleitendes Wort und ein beendendes Wort, meistens `END_xxxx`. Alle Anweisungen dazwischen werden als Anweisungsblock bezeichnet.

Alle Anweisungen enden mit einem Semikolon (`;`).

Da der Programmcode als Text eingegeben wird, ist eine Form nicht vorgeschrieben, bei SCL wird nicht zwischen Groß- und Kleinschreibung unterschieden. Rein theoretisch kann das gesamte Programm in einer langen Zeile hintereinander geschrieben werden. Leerzeichen zwischen Anweisungen oder Anweisungsteilen werden ignoriert.

Da das extrem unübersichtlich ist, gibt es folgende Konventionen, die man – auch wenn es anfänglich umständlich erscheint – einhalten sollte. Sie werden sehen, das Programm wird sehr übersichtlich:

- Elemente, die innerhalb der Funktion eine Einheit bilden, direkt untereinander schreiben und mit Leerzeilen von anderen Einheiten trennen.
- Viele Kommentare in den Quellcode einfügen, die die einzelnen Schritte erläutern.
- Anweisungsblöcke mit Tabulator einrücken, um Ebenen zu kennzeichnen.

Der Editor hilft Ihnen dabei, indem er teilweise automatisch einrückt, Befehle/Konstanten/Schlüsselwörter/Kommentare andersfarbig darstellt.

## 3.4 Das Wichtigste zuerst: Kommentare

Kommentare sind für ein Programm unerläßlich, da ohne Kommentare das beste Programm schwer zu erfassen ist.

Es gibt zwei Arten von Kommentaren:

1. Zeilenkommentare
2. Block-Kommentare

Kommentare werden bei der Übersetzung ignoriert und beeinflussen somit den Programmablauf nicht.

Die Zeilenkommentare beziehen sich – wie der Name schon sagt – nur auf eine Zeile. Alles, was hinter einem doppelten Schrägstrich steht, wird als Kommentar angesehen:

```
Anweisung1;
//Kommentar, was Anweisung 2 macht
Anweisung2; //Kommentar zur Zeile
```

Ein Zeilenkommentar ist auch ein einfaches Werkzeug, einzelne Anweisungszeilen aus dem Programm herauszunehmen:

```
Anweisung1;
//Anweisung2;
Anweisung3;
```

Ein Kommentarblock umfaßt mehrere Zeilen Kommentar:

```
(* Klammer-Stern leitet einen mehrzeiligen Kommentar ein, Stern-Klammer
beendet den Kommentarblock. Es ist beliebig, wie viele Zeilen im Kommentar
stehen. *)
```

Mit dem Blockkommentar können leicht komplette Programmblöcke aus der Bearbeitung herausgenommen werden:

```
Anweisung1;
(*
Bedingung
      Anweisung2;
      Anweisung3;
```

```
      Anweisung4;
END_Bedingung;
*)
Anweisung5;
```

Jeder Funktionskopf sollte einen Kommentarblock beinhalten. Dieser wird im Simatic-Manager auch in den Objekteigenschaften des erzeugten Bausteins angezeigt.

Dieser Kommentarkopf sollte die Aufgabe des Objekts beschreiben, den Autor beinhalten und als Änderungsindex geführt werden:

```
//////////////////////////////////////////
// Autor:         Jens Schürmann
// Erstellt:      15.08.03
// Geändert von: Sc
// Geändert am:  15.07.04; xy geändert
//       Funktion:                                  Baustein       macht....
//////////////////////////////////////////
```

Hinter einer Variablendeklaration angegebene Kommentare werden als Tooltip im FUP/KOP eingeblendet:

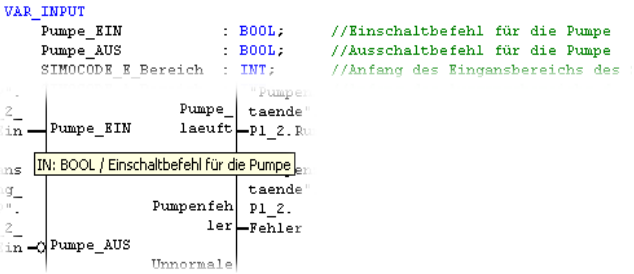



Abbildung 8: Kommentar als Tooltip

# 4 Programmerstellung

Als Komponenten eines Programms sollen in diesem Kapitel betrachtet werden

- Zuweisungen
- Funktionsaufrufe
- Kontrollstrukturen

## 4.1 Zuweisungen

Wenn einer Variable ein Wert einer Konstante oder einer anderen Variable zugewiesen werden soll, geschieht dieses mit Doppelpunkt-Gleichheitszeichen ( `:=` ):

```
Variable1 := 1;
Variable2 := Variable3;
```

Auf der linken Seite steht immer die Variable, *der* ein Wert zugewiesen wird. Auf der rechten Seite steht immer der *Wert*, der einer Variable zugewiesen wird.

## 4.2 Funktionsaufrufe

Eine Funktion, die in FUP/KOP/CFC als Block in das Programm gezogen wird, wird in SCL mit ihrem Namen aufgerufen. Ein Funktionsaufruf besteht immer aus dem Namen der Funktion und einer runden Klammer, die die Eingangsparameter der Funktion aufnimmt:

```
Funktion1(Input1 := Wert1
          ,Output := Aufnehmende_Var);
```

Die Reihenfolge der Parameter ist dabei unerheblich. Werden keine Parameter übergeben oder einem Parameter kein Wert zugewiesen, so werden die entsprechenden Parameter ausgelassen:

```
Funktion1(Input := Wert1);
Funktion2();
```

Wird ein FC programmiert, so hat dieser nicht nur Ausgangsvariablen, sondern selbst auch einen Rückgabewert. Dieser *muß* zugewiesen werden:

```
Ret_Val := FC_Funktion(Input := 1.0);
```

Wird ein FB aufgerufen, so benötigt dieser einen entsprechenden Instanz-DB. Dieser wird mit Punkt hinter dem Funktionsnamen geschrieben:

```
FB_Funktion.IDBxx();
```

Die Rückgabewerte werden dann direkt an diesem IDB abgefragt:

```
FB_Funktion1.IDBxx();
Variable1 := IDBxx.Ret_Val1;
Variable2 := IDBxx.Ret_Val2;
```

Wird der FB als Multiinstanz aufgerufen, wird er im Funktionskopf als Variable deklariert, dieser Name wird dann als Synonym für den Funktionsaufruf genutzt:

```
Var
      Multiinstanz : FB_Funktion;
END_Var

Multiinstanz(Input1 := 1.0);
Variable1 := Multiinstanz.Ret_Val1;
```

## 4.3 Kontrollstrukturen

Es gibt verschiedene Möglichkeiten, den Programmablauf zu kontrollieren. Die bekannteste ist die „Wenn/Dann“-Abfrage, im englischen „If/Then“. Daneben gibt es aber auch noch Schleifen, Abbrüche, Sprünge und Auswahl-Strukturen. Dieses Kapitel beschäftigt sich mit den entsprechenden Möglichkeiten. Als erstes werden die Möglichkeiten des Wertevergleichens vorgestellt.

### 4.3.1 Werte vergleichen

Werte können auf verschiedenste Weise miteinander verglichen werden. Verschiedene Bedingungen können auch logisch verknüpft werden:

Gleichheit prüfen: `=`

Ungleichheit prüfen: `<>`

„Ist größer“ prüfen: `>`

„Ist kleiner“ prüfen: `<`

Bedingungen logisch verknüpfen:

Und `AND`

Oder `OR`

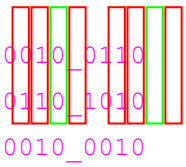
Oder, aber nur einer `XOR`

Negieren `NOT`

Daneben gibt es ein sog. unäres Und: `&`

Hierbei werden ganze Bitmuster ver-und-et. Rückgabewert ist ein Bitmuster, das das Ergebnis jedes Bitvergleichs darstellt:

```
0010_0110 & 0110_1010 = 0010_0010
```




Die logischen Verknüpfungen können mit runden Klammern auch geschachtelt werden. Das kann einerseits für differenziertere Abfragen notwendig sein, andererseits aber auch dem Verständnis dienen.

```
(Vergleich1 OR Vergleich2) AND Vergleich3
```

Sollte bei obigem Beispiel die Klammer weggelassen werden, würde das UND vor dem ODER ausgewertet werden (höhere Priorität), ähnlich wie bei Punkt-/Strichrechnung:

```
Vergleich1 OR Vergleich2 AND Vergleich3
```

ist gleichbedeutend mit:

```
Vergleich1 OR (Vergleich2 AND Vergleich3)
```

Das Ergebnis eines Vergleichs ist immer binär (boolsch):

1 oder 0 bzw. true oder false.

### 4.3.2 IF

Die einfachste Abfrage ist „Wenn/Dann/Ansonsten“, im Englischen „If/Then/Else“.

```
IF a = b THEN
  // Anweisungsteil wenn a=b
  ;
ELSIF a = c THEN
  // Anweisungsteil wenn a=c
  ;
ELSE
  // Anweisungsteil wenn a<>c und a<>b
  ;
END_IF;
```

Die Teile ELSIF und ELSE können je nach Bedarf entfallen. Obiges Beispiel ist gleichbedeutend mit folgendem:

```
IF a = b THEN
  // Anweisungsteil wenn a=b
  ;
ELSE
       IF a = c THEN
        // Anweisungsteil wenn a=c
        ;
      ELSE
        // Anweisungsteil wenn a<>c und
        // a<>b
        ;
      END_IF;
END_IF;
```

ELSIF ist also gleichbedeutend mit einer IF-Abfrage nach einem ELSE.

### 4.3.3 CASE

Mit CASE wird eine Variable auf mehrere mögliche Zustände überprüft:

```
CASE Wert OF
       0..3 :
             // Anweisungen wenn Wert
               // zwischen 0 und 3 liegt
             ;
          8 :
             // Anweisungen wenn Wert=8
             ;
        ELSE:
             // Anweisungen wenn Wert
             // keinen der oben aufge-
             // führten Werte hat
             ;
END_CASE;
```

### 4.3.4 Schleifen

Eine Schleife ist ein Block von Anweisungen, die wiederholt abgearbeitet werden, bis eine Endbedingung erreicht ist. Es gibt Schleifen, in denen grundsätzlich Zähler mitlaufen, dazu werden in der Regel FOR-Schleifen benutzt. Weiterhin gibt es den Unterschied, ob die Abbruchbedingung am Beginn oder am Ende der Schleife überprüft wird. Das nennt man Kopf- oder Fußgesteuert. Hierzu werden WHILE und REPEAT genutzt.

**VORSICHT**

Wenn die Schleife keine geeignete Abbruchbedingung hat, läuft sie endlos!

Je nachdem, wie lange die Bearbeitung der Schleife dauert, kann die Zykluszeit überschritten werden und die SPS geht in STOP! Bei Schleifenprogrammierung immer die zyklische Abarbeitung des Programms beachten!

#### 4.3.4.1 FOR

Die Laufvariable wird beim Eintritt in die Schleife auf einen Startwert gesetzt und bei jedem Durchlauf um die Schrittweite hochgezählt, bis die Laufvariable den Endwert erreicht hat.

```
FOR Laufvariable := Anfang TO Ende BY Schrittweite DO
    // Anweisungsteil
    ;
END_FOR;
```

#### 4.3.4.2 WHILE

WHILE ist eine kopfgesteuerte Schleife. Bevor überhaupt die Schleife betreten wird, wird schon die Abbruchbedingung auf WAHR geprüft. Beispielsweise kann eine Schleife ausgeführt werden, so lange ein Eingang auf 1 ist.

```
WHILE a = b DO
    // Anweisungsteil
    ;
END_WHILE;
```

#### 4.3.4.3 REPEAT

REPEAT ist eine fußgesteuerte Schleife, das heißt, sie wird mindestens einmal durchlaufen, bevor die Abbruchbedingung überprüft wird.

```
REPEAT
    // Anweisungsteil
    ;
UNTIL a = b
END_REPEAT;
```

# 5 Beispiel

Im Folgenden Beispiel soll ein kleines Programm erstellt werden, mit dem Eingänge, die in einem DB vorgegeben werden, abgefragt werden. Die Namen der Eingänge werden verglichen und dementsprechend werden Ausgänge gesetzt.

Was lernen Sie in diesem Beispiel kennen?

- Wie Sie einen Baustein erstellen: OB/FC/UDT/DB
- Wie ein SCL-Programm aufgebaut ist
- Integer
- Boolesche Werte
- Strings
- Direkte und Indirekte Adressierung
- Symbolische und absolute Zugriffe
- Rückgabewerte von Funktionen
- Konstanten
- Eingangsparameter
- IF / FOR / CASE / = / := / > / + / *

Es wird ein UDT geschrieben der als Vorlage für die Array-Elemente eines DBs fungiert.

Der UDT enthält zwei Integer: Byte und Bit des Eingangs, einen booleschen Wert: ob der Eingang abgefragt wird, oder nicht; und einen String mit 10 Zeichen, der den Namen des Eingangs enthält.

Der DB besteht aus einem Array von 8 UDT-Elementen. Des Weiteren gibt es einen String, in dem der Name des ersten aktiven Eingangs eingetragen wird und einen Integer als Kontrollzähler, der anzeigt, wie oft die Abfrage der Eingänge stattfindet.

Es gibt einen FC für die Abfrage der Eingänge, dieser läuft mit einer Schleife durch das Array und fragt die dort eingetragenen Eingänge ab, sofern sie aktivgesetzt sind. Vom ersten Eingang, der aktiv ist, wird der Name in den entsprechenden String im DB geschrieben. Wird kein aktiver Eingang gefunden, wird der String gelöscht und der Rückgabewert der Funktion auf 0 gesetzt.

Der zweite FC nimmt den im DB gespeicherten String, vergleicht ihn mit eigenen Strings und setzt einen Integer. Dieser setzt über eine CASE-Anweisung einen zugewiesenen Ausgang.

Im OB1 werden beide Bausteine aufgerufen, letzterer nur, wenn ein aktiver Eingang gefunden wurde. Wird kein aktiver Eingang gefunden, werden alle Ausgänge gelöscht.

# 6 Diagnose

Über die Brille im SCL-Editor können die Bausteine beobachtet werden.

Es wird nur eine begrenzte Anzahl Zeilen beobachtet. Erkennbar an einem grauen Balken auf der linken Fensterseite, der die Zeilen markiert, die beobachtet werden. Anfang des beobachtbaren Bereichs ist immer die Cursorposition.

Auf der rechten Seite werden alle Variablen angezeigt.

Werden Variablen nicht angezeigt, liegen sie entweder außerhalb des beobachteten Bereichs oder liegen in einem Block (If/Schleife), der nicht bearbeitet wird.

**Vorsicht**

Wenn Variablen, die innerhalb einer Schleife oder einer IF-Bedingung liegen, einmalig durchlaufen wurden und dann nicht mehr bearbeitet werden, so werden die Werte angezeigt, die bei dem einmaligen Durchlaufen des Blocks ermittelt wurden. Es werden also nicht die Aktualwerte angezeigt!

Um den Beobachten-Modus zu verlassen, muß man im Menü „Test“ auf „Test beenden“ gehen. Wird die Brille betätigt, werden zwar die Variablen nicht mehr beobachtet, der Testmodus ist aber nicht abgeschaltet!

Über die Funktionsleiste können auch Haltepunkte in den einzelnen Zeilen gesetzt werden. Beobachtet man das Programm und hat einen Haltepunkt gesetzt, so hält das Programm an diesem Punkt an, sobald es ihn erreicht (beispielsweise ist ein Haltepunkt innerhalb einer IF-Bdeingung gesetzt, so hält das Programm an, sobald die Bedingung erfüllt ist).

**Vorsicht**

Sobald Sie Haltepunkte einsetzen, darf dadurch nicht die Anlage gefährdet werden! Das Programm *steht* in dem Moment! Es werden keine Ausgänge verändert und keine Eingänge gelesen! Bei jeder Anlage muß abgewogen werden, ob es vertretbar ist, das Programm anzuhalten!

Besitzt das Programm einen F-Teil, so müssen alle Not-Aus-Elemente als „Außer Betrieb“ gekennzeichnet werden und es ist sicherzustellen, daß niemand gefährdete Bereiche/Teile betritt oder bedient!

Dann kann man mit entsprechenden Tasten in der Symbolleiste schrittweise durch das Programm gehen oder es fortsetzen.