Autrex Programmierhandbuch

Befehlsreferenz für die Programmierung in Autrex

Stand dieses Dokuments: März 2003 – Status: Draft

Autrex Programmierhandbuch

Befehlsreferenz für die Programmierung in Autrex

Waldweg 55, 88690 Uhldingen, Deutschland

Telefon +49-7556-9218-0, Telefax +49-7556-9218-50

E-Mail: technik@microdesign.de

http://www.microdesign.de

We like to move it!™

Inhaltsverzeichnis

n Raum für Ihre Notizen

Kapitel 1 Einführung

Autrex ist die nächste Generation der SPS-Entwicklung für CPU80 Steuerungssysteme. In einer konsequenten Weiterentwicklung des MC90 Programmierkonzepts bietet Autrex eine strukturierte Programmierung mit Möglichkeiten, wie sie aus den grossen Standard-Programmiersprachen bekannt sind:

- Strukturierte Programmierung
- Objektorientierte Behandlung der Systemressourcen
- Symbolische Definitionen
- Funktionsdefinitionen mit Parametern und Rückgabewerten
- Multi-Task Engine mit bis zu 16 Tasks
- Achspositionierung mit an DIN angelehnten Programmierkonventionen
- 64 Systemtimer mit Zugriff auf den Timerwert zur Zeitmessung
- Programmierbares Interrupt-System
- Abwärtskompatibilität zur MC90 Programmiersprache

Gleichzeitig bleiben die Vorzüge der MC90-Programmierung, wie sequentielle Programmabarbeitung und schnelle Signalverarbeitung, selbstverständlich in vollem Umfang vorhanden.

Kompatibilität hat den Namen Autrex

Intern arbeiten alle Systeme, die in Autrex programmiert werden, weiterhin mit der bewährten MC90 Programmiersprache. Die Autrex-Programme werden vom Compiler auf dem PC in ein kompatibles MC90 Programm umgesetzt.

Dies bedeutet, dass Sie auch weiterhin Funktionen oder Programmteile in der MC90 Sprache programmieren können. So kann z.B. beim Umstieg von MC90 auf Autrex ein Programm schrittweise umgearbeitet werden, während bestehende, funktionierende Programmteile in MC90 Sprache nahezu unverändert ubernommen werden können.

Grundlage dieser Beschreibung

Die Beschreibung des Funktionsumfangs, der Wirkungsweise der SPS-Befehle, der erhältlichen Erweiterungsmodule, der PC-Software sowie des eMC200 Systems als Ganzem beruht auf dem Entwicklungsstand März 2003.

Wenn Sie mit älteren Versionen der Produkte arbeiten ist es möglich, daß einzelne Funktionen nicht so arbeiten wie in dieser Dokumentation beschrieben. Bitte aktualisieren Sie in diesem Fall den Versionsstand der Einzelkomponenten. Aktualisierte Versionen der VMC Workbench und der eMC200 Betriebssysteme erhalten Sie stets im Internet unter http://www.microdesign.de.

1.1 Über diese Dokumentation

Die hier vorliegende Dokumentation ist logisch in folgende Kapitel gegliedert:

Kapitel 1 - Einführung (ab Seite 7)

Dieses Kapitel lesen Sie gerade. Es gibt Ihnen einen kurzen Überblick über die Autrex Sprache und den Aufbach dieser Dokumentation. Außerdem finden Sie hier wichtige Hinweise wie Sie am besten mit dem Handbuch umgehen, damit Sie die gewünschten Information auch schnell auffinden können.

Kapitel 2 - Programmieren (ab Seite 11)

Im zweiten Kapitel wenden wir uns der grundsätzlichen Idee der Autrex Programmierung zu. Dieses Kapitel ist insbesondere dann für Sie interessant, wenn Sie bislang noch keine Programme in Autrex entwickelt haben. Hier erfahren Sie viel über die Struktur der Programme, die grundlegende Abarbeitung von Programmen, die gültigen Datentypen und vieles mehr.

Kapitel 3 - Befehlsübersicht (Seite 23)

Das wohl wichtigste Kapitel dieser Dokumentation ist die Befehlsreferenz. Hier finden Sie alle innerhalb der Autrex Sprache verfügbaren Befehle alphabetisch sortiert. Zu jedem Befehl wird auch ein Beispiel für dessen Anwendung gegeben.

Kapitel 4 - Systemobjekte (ab Seite 49)

Über Systemobjekte bietet Autrex Zugriff auf alle physikalischen und logischen Ressourcen der Steuerung. Eine Übersicht und Erläuterung aller Systemobjekte finden Sie in diesem Kapitel.

Anhänge (ab Seite 73)

In den Anhängen finden Sie eine Anzahl von Übersichten, wie z.B.

- ⇒ eine Wahrheitstabelle für logische Verknüpfungen
- ⇒ ein Tabellenverzeichnis,
- ⇒ eine Abbildungsübersicht,
- ⇒ einen Index der benötigten PC-Software
- ⇒ und einiges mehr.

n Struktur und Nummerierung

Zur besseren Übersicht haben wir darauf verzichtet jede Überschrift mit einer Kapitelnummer zu versehen. Lediglich die wichtigsten Abschnitte sind mit einer Kapitelnummer gekennzeichnet. Falls Sie spezielle Informationen zu einem Thema suchen, verwenden Sie am besten das Inhaltsverzeichnis am Anfang dieser Dokumentation.

n Formatierung in dieser Dokumentation

Damit Sie sich in dieser Dokumentation schnell zurechtfinden können, werden spezielle Informationen stets durch eine besondere Formatierung gekennzeichnet. Wenn Sie sich mit dieser Formatierung vertraut machen, werden Sie sich wesentlich einfacher innerhalb dieser Dokumentation zurechtfinden können.

Wichtige Hinweise

Besonders wichtige Informationen, wie die grundsätzliche Syntax eines Autrex Befehls, werden stets durch **Fettdruck** und eine Kennzeichnung am linken Rand hervorgehoben, z.B.:

IF (Bedingung) ... [Anweisungen] ... ENDIF

Beispiele

Die in dieser Dokumentation häufig anzutreffenden Programmbeispiele sind durch eine `andere Schrift` und durch eine Wellenlinie am linken Rand gekennzeichnet, so z.B.:

```
IF (IN17 & !IN18)                // Wenn Eingang 17 und nicht Eingang 18
          OUT29 = TRUE           // Ausgang 29 einschalten
          OUT1 = FALSE           // Ausgang 31 ausschalten
ELSE                             // ansonsten – wenn Bedingung nicht erfüllt
          OUT29 = FALSE          // Ausgang 29 ausschalten
          OUT31 = TRUE           // Ausgang 31 einschalten
ENDIF                            // Ende der Bedingung
```

Tabellen

Wenn viele Informationen auf einmal dargestellt werden müssen verwenden wir in dieser Dokumentation Tabellen. Eine Übersicht aller Tabellen finden Sie im Anhang dieses Handbuchs.

n Dokumentation der PC-Software

Die vollständige Dokumentation der PC-Software VMC Workbench X2 finden Sie als Online-Dokumente bzw. als Windows-Hilfedateien auf der VMC Workbench X2 CD. Eine gedruckte Fassung der VMC Workbench X2 Dokumentation ist nicht verfügbar.

n Raum für Ihre Notizen

Kapitel 2 Programmieren in Autrex

Wer schon einmal auf dem PC programmiert hat, dem wird die Autrex Sprache auf Anhieb bekannt vorkommen: denn die grundsätzliche Struktur ähnelt sehr den verbreitesten Programmiersprachen, wie z.B. C/C++ oder Visual Basic.

Die Programmstruktur in Autrex wurde sehr stark an die Konventionen der Programmiersprache C angelehnt. Dies betrifft in erster Linie die Definition von Symbolen und Strukturen, die Logik von Abfragen und Bedingungen sowie die Programmierung von Schleifen. Zusätzlich haben wir einige Elemente aus der Programmiersprache Visual Basic übernommen, die in erster Linie der besseren Lesbarkeit des Programms dienen. So kennzeichnen wir z.B. in Autrex Anfang und Ende eines Blocks nicht mit geschweiften Klammern wie in C, sondern mit sprechenden Befehlen wie „ENDWHILE" oder „ENDIF" – ein klares Tribut an Basic als Programmiersprache.

Parallel hierzu haben wir für die Programmierung der Achsbewegungen Anleihen bei der bewährten und weit verbreiteten DIN-Programmierung genommen. So können Sie Achsbewegungen mit den bekannten Anweisungen PTP oder LIN programmieren, absolute Fahrbewegungen mit G90 einleiten und vieles mehr.

Autrex ist das Beste aus drei Welten: der PC-Programmieren, der NC-Programmierung nach DIN und der bewährten MC90 Programmiersprache.

n Alles in Einem

MC-1B integriert alle Funktionen die Sie im Umfeld einer MC200 Umgebung nutzen können, in eine einheitliche Programmiersprache. Insbesondere auf die einfache und effektive Programmierung der Positionierbewegungen wurde sehr großen Wert gelegt. Im Einzelnen können Sie folgende Geräte direkt und ohne Umwege aus der MC-1B Sprache ansprechen:

- Servomotorachsen
- Schrittmotorachsen
- Digitale Aus- und Eingänge
- Analoge Aus- und Eingänge
- Eine Auswahl von Displays, Handbedienteilen und Tastaturen
- Schnelle Zählermodule
- Protokolldrucker mit serieller Schnittstelle
- Andere serielle Gerät, wie z.B. Bilddatenerfassung, PCs, Barcode-Leser usw.

Außerdem ist eine sehr einfache Integration in ein komplexes Feldbussystem wie den Profibus möglich. Noch einfacher wird die Handhabung einer Dezentralisierung mit unseren dezentralen c:Box I/O-Boxen: dann nämlich müssen Sie sich in der Programmierung um gar nichts kümmern, Sie können alle dezentralisierten Module direkt ansprechen.

2.1 Grundsätzliche Vereinbarungen

Zunächst wollen wir die Grundlagen der Autrex Sprache einmal definieren. Dies beinhaltet z.B. wie Befehle geschrieben werden müssen oder wie Kommentare zu deklarieren sind. Wenn Sie bislang noch nicht mit der Autrex Sprache gearbeitet haben, dann sollten Sie sich unbedingt mit diesem Abschnitt beschäftigen.

Bitte beachten Sie, dass wir in diesem Kapitel auf die grundsätzliche Struktur der Sprache eingehen. Für eine vollständige Referenz der einzelnen Sprachelemente beachten Sie bitte die Kapitel 3 und 4.

n Sprachdefinition

Grundsätzlich besteht die Autrex Sprache aus folgenden Elementen:

Strukturbefehle

Mit den vordefinierten Strukturbefehlen können Sie den Programmablauf und die Struktur des Programms beeinflussen. Zu dieser Kategorie gehören folgende Abfragen, Schleifen und Sprünge. Folgende Strukturbefehle stehen zur Verfügung:

- **n** FOR...NEXT
  Ausführung einer Schleife mit einer Zählvariable
- **n** IF...ELSE...ENDIF
  Bedingte Ausführung eines Programmblocks
- **n** WHILE...ENDWHILE
  Schleife mit Bedingungsprüfung am Anfang der Schleife
- **n** REPEAT...UNTIL
  Schleife mit Bedingungsprüfung am Ende der Schleife
- **n** SWITCH...CASE...ENDSWITCH
  Abfrage einer Bedingung mit mehreren Möglichkeiten
- **n** GOTO
  Verzweigung zu einem Label
- **n** FUNCTION
  Deklariert eine Funktion (ein Unterprogramm)
- **n** RETURN
  Ende einer Funktion

Diese Strukturbefehle erlauben die umfassende Programmierung aller denkbaren Bedingungen. Durch eine nahezu unbegrenzte Verschachtelungstiefe der Struktur können auch höchst komplexe Aufgaben einfach gelöst werden.

Systemobjekte

Die in Autrex vordefinierten Systemobjekte kapseln alle notwendigen Statusinformationen und Funktionen der Steuerung. Den Systemobjekten können auch Daten zugewiesen werden, um den Status des System zu ändern. Einige typische Beispiele für Systemobjekte sind:

- **n** Systemobjekt SYS
  Im Objekt SYS sind alle Funktionen und Statusinformationen des Systems als solches gespeichert. SYS verfügt über eine Anzahl Unterobjekte, auf die man ebenfalls direkt zugreifen kann, wie z.B.:
  - ⇒ SYS.RESET ermöglicht den Neustart der Steuerung über die Zuweisung SYS.RESET = TRUE
  - ⇒ SYS.AXIS ermittelt, wie viele Achscontroller angeschlossen sind.
  - ⇒ SYS.DATE gibt das aktuelle Datum der Echtzeituhr zurück.

- Achsenobjekte A1 bis A8
  In den Objekten A1 bis A8 sind die Zustände von bis zu 8 Achsen hinterlegt. Auch hier kann über die Unterobjekte direkt auf den Status zugegriffen werden.

  ⇒ A1.INPOS ermittelt, ob die Achse in Position ist.

  ⇒ Über A1.VEL kann die Sollgeschwindigkeit ermittelt oder auch gesetzt werden.

  ⇒ A1.START speichert die letzte Startposition der Achse 1, konsequenterweise findet sich in A1.END die letzte Zielposition für diese Achse.

- Digitale Eingänge IN1 bis IN512
  Die digitalen Eingänge speichern nicht nur den aktuellen Status eines digitalen Eingangs, sondern auch einen Simulationsstatus.

- Digitale Ausgänge OUT1 bis OUT512
  Die Systemobjekte OUT1 bis OUT512 speichern jeweils einen bool'schen Wert für einen digitalen Ausgang und können nur den Wert TRUE oder FALSE annehmen.

Neben der hier aufgeführten Systemobjekten gibt es noch eine ganze Reihe weiterer für alle Funktionen der Steuerung. Der Zugriff auf die Funktionen erfolgt damit über jeweils entsprechende Systemobjekte. Eine vollständige Übersicht finden Sie im Kapitel 4 dieser Dokumentation.

Compilerbefehle

Ebenfalls sehr wichtig sind die Compilerbefehle und –Anweisungen. Hiermit können Symbole und Makros definiert werden oder auch – über symbolische Definitionen – bestimmte Programmteile von der Compilierung ausgeblendet werden.

Jeder Compilerbefehl wird stets eingeleitet durch das Gatterzeichen #. Die verfügbaren Compilerbefehle sind im Einzelnen:

- #DEFINE
  Definiert einen symbolischen Namen auf einen Wert oder ein Systemobjekt.

- #IF...#ELSE...#ENDIF
  Definiert einen Abschnitt für die bedingte Compilierung. Im Gegensatz zu der Konstruktion IF...ELSE...ENDIF, die eine Bedingung während des Programmablaufs prüft, können mit den Compileranweisungen #IF...#ELSE...#ENDIF ganze Programmblöcke von der Compilierung ausgeschlossen werden.

- #MACRO
  Definiert ein Macro, dem auch Parameter übergeben werden können.

n Operatoren

Mit Operatoren bezeichnet man in der Programmierung alles, was zwei oder mehr Werte auf irgendeine Weise miteinander verbindet. Dies kann eine einfache logische UND-Verknüpfung sein, oder auch ein komplexer mathematischer Ausdruck mit mehreren Klammerebenen.

Grundlegend unterscheidet man folgende Arten von Operatoren:

- Logische Operatoren
  Mit logischen Operatoren werden bool'sche Ausdrücke miteinander verknüpft. Zu dieser Gruppe gehören z.B. die bekannten UND- bzw. ODER-Verknüpfungen

- Vergleichsoperatoren
  Mit Vergleichsoperatoren werden zwei Werte beliebigen Typs miteinander verglichen. Eine Vergleichsoperation ergibt immer einen bool'schen Wert. Typische Operatoren hierfür sind „>" für „grösser als", „==" für „ist gleich" usw.

- Arithmetische Operatoren
  Wenn Zahlenwerte verändert werden, spricht man von arithmethischen Operatoren. Zu dieser Gruppe gehören natürlich auch alle Rechenfunktionen, wie Addition, Subtraktion oder Mulitplikation.

Logische Operatoren

Mit logischen Operatoren bool'sche Ausdrücke vom Wert TRUE/FALSE miteinander verknüpft werden. Autrex kennt die logischen Operatoren UND, ODER sowie NICHT in beliebiger Verknüpfung. Die Notation der Operatoren hält sich an den C/C++ Standard, allerdings ohne die dort übliche Zeichenverdoppelung:

- UND-Verknüpfung: Zeichen „&"
- ODER-Verknüpfung: Zeichen „|"
- NICHT-Verknüpfung: Zeichen „!"

Typische logische Verknüpfungen sehen damit wie folgt aus:

```
IF IN1 & IN2                      // Wenn Eingang 1 UND Eingang 2
IF IN1 & !IN2                     // Wenn Eingang 1 UND NICHT Eingang 2
IF IN1 | IN2 | IN3                // Wenn Eingang 1 ODER Eingang 2 ODER Eingang 3
IF (IN1 & !IN2) | (!IN1 & IN2)    // Wenn Eingang 1, aber NICHT Eingang 2 ODER
                                  // NICHT Eingang 1, aber Eingang 2
```

Vergleichsoperatoren

Vergleichsoperatoren kommen immer dann ins Spiel, wenn zwei Zahlenwerte verglichen werden sollen. Das Ergebnis eines Vergleichs ist immer ein bool'scher Wert, also TRUE oder FALSE. Autrex kennt folgende Vergleichsoperatoren:

- „Ist gleich": Zeichen „=="
- „Ist nicht gleich": Zeichen „!=" oder alternativ „<>"
- „Ist kleiner als": Zeichen „<"
- „Ist kleiner als oder gleich": Zeichen „<="
- „Ist grösser als": Zeichen „>"
- „Ist grösser als oder gleich": Zeichen „>="

Im Programm wird dies dann wie folgt formuliert:

```
IF (A1.START > A1.END)            // War die Startposition grösser als das Ziel?
IF (A1.VEL == 1000)               // Ist die Geschwindigkeit auf 1000 gesetzt?
```

Natürlich kann das Ergebnis einer Vergleichsoperation auch mit anderen bool'schen Werten kombiniert werden:

```
IF (A1.END == A2.END) && IN10     // Wenn die Zielpositionen der Achsen 1 und 2
                                  // gleich sind und der Eingang 10 ein ist
```

Arithmetische Operatoren

Unter arithmetischen Operatoren versteht man alle Verknüpfungen, die zwei Werte verknüpft und daraus einen neuen Wert bildet. Autrex kennt hier die üblichen Grundrechenarten (+, -, *, /) sowie die Zuweisung eines Wert (Zeichen =). Hierbei beachtet Autrex die üblichen arithmetischen Regeln (Punkt vor Strich, Klammer geht vor usw.).

Einige Beispiele:

```
A1 = A1.Act + 100                 // Aktuelle Position Achse 1 + 100
IF A1.END * 100 < A2.END / 100    // Wenn die Zielposition Achse 1 * 100 kleiner
                                  // ist als die Zielposition der Achse 2 / 100
```


n Kommentare

Kommentare oder Hinweise zum Ablauf des Programms erleichtern die Lesbarkeit des Quelltextes. Deshalb sollten Sie mit entsprechenden Erläuterungen nicht zu sparsam umgehen. In den bisherigen Beispielen haben Sie bereits gesehen, daß wir hinter jedem Befehl entsprechende Kommentare eingefügt haben und zwar jeweils mit der Zeichenkette "//" beginnend. Dies stellt eine der Möglichkeiten dar Kommentare in Ihren Quelltext einzuflechten.

Rest der aktuellen Zeile als Kommentar markieren

Wenn Sie die Zeichenkette "//" in Ihrem Quelltext verwenden, wird der Rest der jeweiligen Zeile als Kommentar markiert und beim Programmablauf nicht berücksichtigt.

Befehl //Komentar

Dies ist die übliche Form einen Quelltext zu kommentieren: hinter jeden SPS-Befehl eine Erläuterung zu setzen, die erklärt, was an dieser Stelle gemacht werden soll.

Längere Kommentare

Wenn Sie einen längeren Kommentar einfügen möchten, z.B. um grundsätzliche Funktionen zu erklären, oder Aufgaben zu markieren die noch durchgeführt werden müssen, können Sie einen Block von mehreren Zeilen als Kommentar kennzeichnen. Hierzu markieren Sie den Anfang des Kommentars mit der Zeichenkette "/*". Alles was nach dieser Zeichenkette kommt, wird vom Compiler als Kommentar behandelt und beim Programmablauf nicht berücksichtigt. Um den Kommentar zu beenden verwenden Sie die Zeichenkette "*/". Danach wird der Quelltext vom Compiler wieder berücksichtigt.

/* Kommentar */

Beispiel für die Verwendung von Kommentaren

```
/* Hier beginnt unser Kommentar. Wir können nun mehrere Zeilen Erläuterung
zu dem Programm schreiben. Alles, was innerhalb des Kommentars steht, wird vom
Compiler ignoriert. */

Start:                          // Label (Sprungmarke)
IF IN10 | IN20                  /* Wenn Eingang 10 oder Eingang 20 */
```

In dem oben gezeigten Beispiel sehen Sie alle Möglichkeiten, Kommentare in Ihren Quelltext einzuflechten, auf einen Blick. Bitte beachten Sie, daß Sie jeden Kommentar den Sie mit "/*" beginnen, auch mit "*/" abschließen müssen!

Kommentare werden bei der Compilierung Ihres SPS-Programms nicht berücksichtigt und auch nicht in die Steuerung übertragen. Deshalb belegen Kommentare auch keinerlei Speicherplatz innerhalb der Steuerung.

n Labels (Sprungmarken)

In dieser Dokumentation sprechen wir grundsätzlich von Labels, meinen damit aber natürlich genauso Sprungmarken. Labels ist lediglich der englische Begriff für die gleiche Sache. Da sich die Bezeichnung "Label" jedoch im Allgemeinen technischen Sprachgebrauch eingebürgert hat, bleiben wir künftig auch bei diesem Begriff.

Mit einem Label definieren Sie eine bestimmte Stelle innerhalb Ihres SPS-Programms, zu der Sie von einer anderen Stelle des Programms aus verzweigen möchten. Dies kann z.B. der Fall sein, wenn Sie

- eine Schleife programmieren möchten,
- den Anfang eines Unterprogramms deklarieren möchten oder
- je nach Ergebnis einer Abfrage unterschiedliche Abläufe ausführen möchten.

Labels geben also einer bestimmten Stelle im Programm einen Namen, den Sie in Ihrem SPS-Programm jederzeit verwenden können.

Pro und Contra Labels

Labels sind eigentlich ein Relikt aus den „alten Zeiten" der Programmierung, bevor es Konstruktionen wie SWITCH...CASE oder WHILE...ENDWHILE gab. In modernen Programmiersprachen spielen Labels nur noch eine Aussenseiterrolle und werden höchst selten verwendet.

Es hat viele Vorteile, ein Programm ohne die Verwendung von Labels zu formulieren. So können z.B. Unterroutinen oder ganze Programmteile kopiert werden, ohne dass man ein Label umbenennen muss. Auch braucht man, wenn in einen Programmteile neue Abfragen eingefügt werden sollen, keine zusätzlichen Labelnamen vergeben.

Dennoch gibt es ein paar Situationen, in denen es bequem sein kann, mit Labels zu programmieren, z.B. wenn man über eine komplizierte Schleife oder Abfrage einfach hinweg springen möchte, ohne in jedem Fall alle Bedingungen abprüfen zu müssen.

Letzten Endes bleibt es Ihren überlassen, ob und in welchem Ausmass Sie Labels in Ihrem Programm verwenden. Autrex bietet Ihnen alle Möglichkeiten.

Einschränkungen für Labelnamen

Bitte beachten Sie folgende Einschränkungen für Labelnamen:

- Der Labelname darf nicht länger als 63 Zeichen sein.
- Der Name darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Labelnamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Labelnamen ignoriert.
- Der Labelname muss in der Deklaration stets mit einem Doppelpunkt abgeschlossen werden, beim Aufruf eines Labels ist dies nicht notwendig.

2.2 Datentypen

In Autrex gibt es vier unterschiedliche Datentypen:

- Variablen, angesprochen über den Buchstaben „R" (z.B. R20)
- Merker, angesprochen über den Buchstaben „M" (z.B. M100)
- Konstanten
- Systemobjekte

Im Folgenden werden diese Datentypen erläutert:

n Variablen (Register)

SPS-Variablen sind in Autrex stets 32 Bit breit und können mit bis zu Nachkommastellen angegeben werden. Die Anzahl der gewünschten Nachkommastellen wird zu Anfang des Autrex-Programms mit der Compiler-Direktive #DECIMAL bestimmt:

```
#DECIMAL 2                          // Zwei Nachkommastellen verwenden
```

Wird die Direktive #DECIMAL nicht angegeben, arbeitet Autrex mit einer Nachkommastelle (Genauigkeit 1/10). Intern werden in Autrex Variablen stets ganze Zahlen gespeichert. In der Fachsprache nennt man dies "Integer-Zahlen". Nachkommastellen werden vom Compiler automatisch in das benötigte Festkomma-Format umgesetzt.

Variablen können mit Autrex universell eingesetzt werden. Es gibt grundsätzlich keine Limitierung dafür was tatsächlich in einer Variable enthalten sein muß. So können Variablen einen normalen Wert enthalten, einen Text repräsentieren oder aber ein Zeiger auf anderen Daten sein.

n Bits (Merker)

SPS-Merker können nur 1 Bit Informationen speichern. Merker können also nur den Zustand "ein" oder "aus" bzw. TRUE/FALSE annehmen. Der Einsatz von Merkern empfiehlt sich immer dann, wenn nur eine einzelne Status-Information gespeichert oder an einen anderen Programmteil weitergegeben werden muß.

n Konstanten

Eine Konstante definiert einen symbolischen Namen als eine Zahl. Im SPS-Programm kann dieser symbolische Name dann verwendet werden, um bei Befehlen die konstante Zahlenwerte erwarten, statt der Zahl einen Klartextnamen anzugeben.

Besonders sinnvoll ist der Einsatz von Konstanten dann, wenn Sie den gleichen Wert an mehreren Stellen im Programm verwenden, z.B. um die Geschwindigkeit einer Positionierbewegung anzugeben. Statt im Falle einer Änderung dann an mehreren Stellen den Zahlenwert auszutauschen, müssen Sie dann nur noch den Wert bei der Definition der Konstante verändern.

n Systemobjekte

Wie bereits vorher beschrieben, werden in den Systemobjekten alle Funktionen der Steuerung eingebunden. Jedes einzelne Element eines Systemobjekts reduziert sich jedoch dann wieder auf einen der bereits oben beschriebenen Datentypen: Variablen, Merker oder Konstanten.

2.3 Adressierung von Daten

Grundsätzlich gibt es in Autrex drei verschiedene Möglichkeiten, Systemdaten – hierzu gehören Variablen und Merker, aber auch Eingänge, Ausgänge und Achsen – anzusprechen:

n Ansprechen über absolute Adresse

Jede der vordefinierten Ressourcen kann über ihre absolute Adresse direkt angesprochen werden. Mit dieser maschinennahen Art der Programmierung haben Sie direkten Zugriff auf die vorhandenen Ressourcen:

```
R1        entspricht  Variable 1
R100      entspricht  Variable 100
M20       entspricht Merker 20
OUT1      entspricht Ausgang 1
IN17      entspricht Eingang 17
A4        entspricht Achse 4
```

n Ansprechen über Index

Um die Ressourcen indirekt – z.B. über eine Zeigervariable – anzusprechen, kann statt der direkten Angabe der Nummer auch ein Index verwendet werden.

```
R[1]      entspricht  Variable 1
R[R20]    verwendet die Variable, auf den die Variable 20 (R20) zeigt
M[R300]   verwendet den Merker, auf den die Variable 300 (R300) zeigt
OUT[10]   entspricht Ausgang 10
A[Achse]  verwendet die Achse, auf die die Variable Achse zeigt
```

n Ansprechen über Symbolnamen

Die wohl komfortabelste Variante ist die Programmierung über symbolische Namen. Hierzu müssen die Symbole zunächst definiert werden. Dies kann z.B. so erfolgen:

```
#DEFINE   Achse     R20          // Definiert "Achse" als Variable 20 (R20)
#DEFINE   Zähler    VAR          // Definiert „Zähler“ als beliebige Variable
#DEFINE   Fehler    M100         // Definiert „Fehler“ als Merker 100
#DEFINE   Stopp     BIT          // Definiert „Stopp“ als beliebiges Bit
#DEFINE   A_Lampe   OUT7         // Definiert "A_Lampe" als Ausgang 7
```

Im Programm kann dann einfach der definierte Name verwendet werden. Autrex übersetzt beim Compilieren diesen Namen in die entsprechend defnierte Ressource:

```
Achse = 3                        // Weist R20 den Wert 3 zu
Fehler = FALSE                   // Weist dem Merker 100 den Wert FALSE zu
```

Selbstverständlich können Sie die verschiedenen Adressierungsformen frei mischen und so z.B. einen Variablenbereich als Datenbereich ohne symbolischen Namen verwenden, während andere Ressourcen über den Namen angesprochen werden.

2.4 Funktionen

Eine der mächtigsten Möglichkeiten innerhalb der Autrex Programmiersprache ist die freie Deklaration von Funktionen. Funktionen kann man im einfachsten Sinne als Unterprogramme verstehen, die aus dem Programmablauf heraus einen Parameter erhalten können und auch ein Ergebnis zurück geben können.

n Wie werden Funktionen deklariert?

Die Deklaration einer Funktion beginnt stets mit dem Schlüsselwort „Function":

```
FUNCTION ClearScreen()
     [Programmcode der Funktion]
ENDFUNC
```

Neben dem Schlüsselwort wird der Name dieser Funktion angegeben. Obiges Beispiel deklariert die Funktion „ClearScreen", die keine Parameter erwartet und auch kein Ergebnis zurückgibt. Damit entspricht diese Deklaration einem einfachen Unterprogramm, wie es auch in der MC90 Sprache bekannt war.

Diese Deklaration lässt sich jedoch erweitern um einen Parameter: wir übergeben jetzt der Funktion ClearScreen eine zusätzliche Information, nämlich welches der beiden angeschlossenen Displays gelöscht werden soll:

```
FUNCTION ClearScreen(VAR DisplayNummer)
     SWITCH (DisplayNummer)
          CASE 1:
          [Programmcode zum Löschen von Display 1]
          CASE 2:
          [Programmcode zum Löschen von Display 2]
     ENDCASE
ENDFUNC
```

In obigem Beispiel deklarieren wir nicht nur ein Unterprogramm, sondern übergeben auch noch eine zusätzliche Information an dieses Unterprogramm. Dadurch kann das Unterprogramm selbst entscheiden, wie es sich verhält. Aus dem Hauptprogramm wird das Unterprogramm dann ganz einfach mit:

```
ClearScreen(1)
```

aufgerufen, um das erste angeschlossene Display zu löschen. Statt der Konstante „1" kann natürlich auch ein Rechenergebnis oder eine Variable übergeben werden.

Funktionen mit Rückgabewert

Noch komfortabler wird eine Funktion jedoch durch den Rückgabewert. Wir wollen z.B. eine Funktion programmieren, die als Parameter eine Achsnummer erhält und als Ergebnis zurückliefert, ob der Benutzer die Teachfunktion ausgeführt oder abgebrochen hat. Dafür deklarieren wir eine Funktion:

```
FUNCTION TEACH(VAR Achse): BOOL
```

Mit dieser Funktionsdeklaration wird die Funktion "Teach" definiert, die als Parameter die zu teachende Achse erhält und als Ergebnis zurückgibt, ob das Teachen erfolgreich abgeschlossen wurde. Im Programm kann diese Funktion dann wie folgt aufgerufen werden:

```
IF Teach(1)                          // Wenn Teachen für Achse 1 erfolgreich...
```

2.5 Achsbewegungen

Zur besseren Verständlichkeit können alle Achsbewegungen in Autrex auch in DIN programmiert werden. Hierzu wurde ein Teil der in Siemens® S7® verwendeten Achsbefehle übernommen, wie z.B.

- G90 A1=Pos A2=Pos A3=Pos
  Startet eine absolute Positionierung für die Achsen 1, 2 und 3, wartet jedoch nicht, bis die Zielposition erreicht wurde.
- G91 A1=Strecke A2=Strecke
  Startet eine absolute Positionierung für die Achsen 1 und 2, wartet jedoch nicht, bis die Zielposition erreicht ist.
- PTP G90 A1=Pos A2=Pos
  Startet eine „Point-To-Point" Positionierung für die Achsen 1 und 2. Die Programmausführung wartet, bis die Positionierung beendet ist.
- G0/G1
  Schaltet um zwischen Eilgang und parametrierter Geschwindigkeit.
- G641/G64
  Schaltet die Strombegrenzung ein oder aus.

Eine vollständige Übersicht der implementierten Achsbefehle nach DIN finden Sie im Kapitel 3.

n Flexibilität in der Achsprogrammierung

Intern werden die Achsbefehle nach DIN immer zunächst in „normale" Autrex-Sprache umgesetzt. So führt die Programmzeile

```
PTP G90 A1=100.00 A2=200.00
```

zu folgendem Autrex Programmcode:

```
While (!A1.Inpos & !A2.Inpos & !A1.Error & !A2.Error)
Endwhile

A1.End = 100.00
A2.End = 200.00
A1.Run = TRUE
A1.Run = TRUE

While (!A1.Inpos & !A2.Inpos & !A1.Error & !A2.Error)
Endwhile
```

n Eigener Programmcode während der Positionierung

Manchmal möchte man, während man auf eine Achse wartet, zusätzliche Aktionen ausführen. In diesem Fall ist es wenig sinnvoll, die Funktion PTP zu verwenden, da hier die Programmausführung erst nach erfolgter Positionierung fortgesetzt wird. Schreiben Sie jedoch den DIN-Befehl G90 bzw. G91 ohne den Zusatz PTP, dann wird die Programmausführung direkt nach dem Start der Achsen fortgesetzt.

In folgendem Beispiel schalten wir den Ausgang 7 ein, sobald die Achse 2 nur noch 10mm von ihrer Zielposition entfernt ist:

```
G90 A1=100.00 A2=200.00
While (!A1.Inpos & !A2.Inpos & !A1.Error & !A2.Error)
     If (A2.Remain <= 10.00) & !Out7
          Out7 = TRUE;
     Endif
Endwhile
```


2.6 Abläufe und Tasks

Autrex ist von Haus aus eine Mehrtask-Maschine, d.h. es können mehrere Abläufe voneinander unabhängig bearbeitet werden. Tasks werden hierbei ganz normal als Programmteile geschrieben, die dynamisch als Tasks aufgerufen werden können. Noch wichtiger ist hingegen zu verstehen, daß Autrex nicht als eine Zyklus-Maschine im eigentlichen Sinn arbeitet. Im Gegensatz zu einer Zyklusmaschine werden bei Autrex sämtliche Betriebszustände – wie z.B. der Zustand von Ein- und Ausgängen – werden fortlaufend aktualisert, auch die Programmausführung ist nicht an einen festen Zyklus gebunden.

n Wie läuft ein Programm in Autrex ab?

Grundsätzlich besteht ein Programm in Autrex zunächst einmal aus dem Initialisierungscode, in dem Sie für Ihr Programme notwendige Grundeinstellungen vornehmen – wie z.B. das Zurücksetzen von Statusinformationen, das Einschalten der Achsen, die Anzeige einer Einschaltmeldung oder das Starten zusätzlicher Tasks.

Im Normalfall folgt der Initialisierung dann die Hauptprogrammschleife, die tatsächlich als Schleife programmiert ist. Folgende Darstellung mag dies deutlich machen:

```
// Initialisierung

ModemPasswort()                  // Funktion Modempasswort überprüft Code
ClearScreen(1)                   // Bildschirm 1 löschen
Copyright()                      // Copyright Meldung anzeigen
Leistungsteile(TRUE)             // Leistungsteile einschalten

// Weitere Initialisierungen

// Hauptprogrammschleife

while (TRUE)                     // Schleife unendlich ausführen
// Hauptprogramm
endwhile                         // Ende der Hauptprogrammschleife
```

n Verwenden von Tasks

Tasks werden in Autrex einfach als Funktionen programmiert (Hinweise zur Deklaration von Funktionen finden Sie in Abschnitt 2.3). Durch einen einfachen Zuweisungsbefehl wird die Funktion dann als Task ausgeführt:

```
Function MyTask()
[Programmcode der Task]
Endfunc

Task3 = MyTask()                 // Funktion MyTask() als Task 3 starten
Task3.Run = FALSE                // Task 3 anhalten
Task3.Run = TRUE                 // Task 3 weiterlaufen lassen
```

Bitte beachten Sie, dass alle Tasks auf die identischen physikalischen Ressourcen zugreifen. So kann z.B. eine Variable MyPLCState von allen Tasks gleichzeitig beschrieben werden. Achten Sie bei der Programmierung von Tasks darauf, dass nicht mehrere Tasks die gleiche Variable zur gleichen Zeit wollen.

n Raum für Ihre Notizen

Kapitel 3 Befehlsübersicht

In diesem Kapitel finden Sie eine vollständige Übersicht aller Autrex Befehle sowie der vordefinierten Funktionen. Zur besseren Übersicht und zum einfacheren Auffinden der jeweiligen Befehle ist dieses Kapitel nach Funktion der Befehle unterteilt:

(Kapitelübersicht folgt noch)

n Wichtiger Hinweis

Die Befehlsübersicht setzt voraus, daß Sie mit den Grundlagen der Autrex Programmierung bereits vertraut sind. Beachten Sie ggf. die Erläuterungen in Kapitel 2 - Programmieren ab Seite 11.

3.1 Operatoren und Ausdrücke

In diesem Abschnitt behandeln wir alle in Autrex verfügbaren Operatoren, erklären die Verwendung von Ausdrücken und die Zuweisung von Werten.

n Operatoren

Folgende Operatoren können in allen Ausdrücken verwendet werden:

Operand	Funktion	Erster Operator	Zweiter Operator	Ergebnis
=	Zuweisung	Beliebig	Beliebig	keines
&	Logisches UND	Bool'scher Wert	Bool'scher Wert	Bool'scher Wert
|	Logisches ODER	Bool'scher Wert	Bool'scher Wert	Bool'scher Wert
!	Logisches Nicht	Keiner	Bool'scher Wert	Bool'scher Wert
+	Addition	Zahl	Zahl	Zahl
-	Subtraktion	Zahl	Zahl	Zahl
*	Mulitplikation	Zahl	Zahl	Zahl
/	Division	Zahl	Zahl	Zahl
>	Vergleich auf größer	Zahl	Zahl	Bool'scher Wert
>=	Vergleich auf größer oder gleich	Zahl	Zahl	Bool'scher Wert
==	Vergleich auf gleichen Wert	Zahl	Zahl	Bool'scher Wert
!=	Vergleich auf nicht gleichen Wert	Zahl	Zahl	Bool'scher Wert
<>	Vergleich auf nicht gleichen Wert	Zahl	Zahl	Bool'scher Wert
<	Vergleich auf kleiner	Zahl	Zahl	Bool'scher Wert
<=	Vergleich auf kleiner oder gleich	Zahl	Zahl	Bool'scher Wert

n Tabelle 1 –Operatoren

Beachten Sie bitte, dass das Ergebnis eines Ausdrucks mit dem erwarteten Parameter der Abfrage, der Funktion oder der Zuweisung übereinstimmen muss. Wenn Sie also Verknüpfungen innerhalb einer IF-Abfrage verwenden, muss das Ergebnis des Ausdrucks einen bool'schen Wert ergeben.

Zum besseren Verständnis der logischen Verknüpfungen (UND, ODER, NICHT) beachten Sie bitte auch die Wahrheitstabelle in Anhang ???.

n Ausdrücke

Als Ausdruck wird eine Kombination von mit Werten mit Hilfe von den in Tabelle 1 aufgeführten Operatoren bezeichnet. Dabei können alle verfügbaren Operatoren beliebig miteinander kombiniert werden.

Reihenfolge der Auswertung

- Prinzipiell werden alle Ausdrücke von links nach rechts ausgewertet
- Bei arithmetischen Operatoren gilt: Punkt vor Strich
- Bei Klammersetzung gilt: der Klammerinhalt wird zuerst abgearbeitet

Die bedeutet, am Beispiel gezeigt:

```
R100 = (2 + 2) * 10 + 9 / 3 – (5 – 2) + 100
```

wird ausgewertet als:

```
2 + 2 = 4
4 * 10 = 40
40 + 9 / 3 = 40 + 3 = 43
43 – (5 – 2) = 43 – 3 = 40
40 + 100 = 140
R100 = 140
```

Klammerebenen

Autrex unterstützt Klammersetzung bis in die achte Verschachtelungsebene. Fehlende schliessende Klammern führen zu einer Fehlermeldung.

Rechenergebnis vs. bool'scher Wert

Bool'sche Werte können, wie bereits mehrfach erwähnt, nur den Zustand TRUE oder FALSE annehmen. Deshalb müssen Sie auch bei komplexen Konstruktionen darauf achten, dass z.B. die Bedingung eines komplexen Ausdrucks immer einen bool'schen Wert ergibt. Im Folgenden finden sich zwei Ausdrücke: der zweite ergibt – richtigerweise – einen bool'schen Wert, der erste ein Rechenergebnis.

```
If (R100 + 20 – R101) / 10            // Falsch! Ergibt Rechenergebnis!
If (R100 + 20 – R101) / 10 == 100     // Richtig! Ergibt bool'sches Ergebnis!
```

3.2 Strukturbefehle

n FOR...NEXT

FOR *Variable* = *Anfangswert* TO *Endwert* ... NEXT

Mit dem Befehl FOR wird eine Programmschliefe eingeleitet. Zu Beginn der Schleife wird der angegebenen Variable der *Anfangswert* zugeordnet, wobei dieser Wert entweder eine Konstante oder eine andere Variable sein kann.

Vor der Ausführung des folgenden Programmcodes wird überprüft, ob der Wert von *Variable* grösser als der angegebene *Endwert* ist. *Endwert* kann ebenfalls entweder eine Konstante oder eine Variable sein. Ist der Wert von *Variable* grösser als der *Endwert*, wird die Programmausführung nach dem NEXT-Befehl fortgesetzt. Ansonsten wird der Programmcode zwischen FOR und NEXT ausgeführt.

Beim NEXT Befehl erhöht Autrex den Wert der *Variable* um eins und springt zurück zum Anfang der Programmschleife.

Funktionsweise

```
Variable = Anfangswert
Ist Variable > Endwert?
       Ja:    Führe Programm nach dem Befehl NEXT weiter aus
       Nein:  Führe Programm bis zum Befehl NEXT aus
              Variable = Variable + 1
              Springe zurück zum Anfang der Schleife
```

Beispiele

```
// Einfache FOR...NEXT Schleife: Der Ausgang wird 10x umgeschaltet, das
// Programm dazwischen für eine halbe Sekunde angehalten
FOR Zähler = 1 TO 10              // 10 Durchläufe der Schleife
     Out1 = !Out1                 // Ausgang 1 umschalten
     Wait(0.5)                    // Eine halbe Sekunde warten
NEXT

// Löschen eines Datenfelds
#DEFINE    Anfang     R200        // Erste Adresse steht in R200
#DEFINE    Ende       R201        // Letzte Adresse steht in R201
FOR Zähler = Anfang TO Ende       // Vom Wert in R200 bis zum Wert R201
     R[Zähler] = 0                // Variable indirekt auf 0 setzten
NEXT
```

Hinweise

- Verwenden Sie FOR...NEXT immer dann, wenn Sie eine bestimmte Anzahl von Wiederholungen ausführen möchten.
- Alternativ zum FOR...NEXT Befehl können auch Konstruktionen mit WHILE...ENDWHILE oder REPEAT...UNTIL verwendet werden.
- FOR...NEXT wird nicht ausgeführt, wenn der *Anfangswert* größer als der *Endwert* ist.

Siehe auch

n FUNCTION

FUNCTION Funktionsname([*Parametertyp Parametername*]): [*Rückgabewert*]

Mit der Anweisung FUNCTION wird eine Funktion bzw. ein Unterprogramm deklariert. Dieses Unterprogramm kann dann aus dem Autrex Programmcode heraus wie ein ganz normaler Befehl aufgerufen werden.

Da Sie in Autrex einer Funktion einen Parameter übergeben können und auf Wunsch auch ein Funktionsergebnis zurückerhalten, gibt es grundsätzlich vier Varianten des Befehls FUNCTION:

- Die Funktion erhält keine Parameter und gibt auch keinen Wert zurück
- Die Funktion erhält einen Parameter, gibt aber keinen Wert zurück
- Die Funktion erhält keinen Parameter, gibt aber trotzdem einen Wert als Funktionsergebnis zurück
- Die Funktion erhält einen Parameter und gibt einen Wert als Funktionsergebnis zurück

Was auf den ersten Blick kompliziert klinkt, ist in der Praxis ganz einfach: entweder die schreiben bei der Deklaration einer Funktion dazu, dass diese Funktion einen Wert zurück gibt, oder Sie unterlassen es. Autrex erkennt dadurch automatisch, um welchen Typ einer Funktion es sich handelt.

Datentypen des Parameters und des *Rückgabewert*s

Funktionen kennen grundlegend nur zwei Datentypen: Zahlenwerte (VAR) und bool'sche Wert (BOOL). Bei der Deklaration der Funktion müssen Sie natürlich angeben, welcher dieser beiden Datentypen als Parameter bzw. als *Rückgabewert* verwendet werden soll. Die Deklaration des Datentyps wird beim Parameter hierbei vor den Namen des Parameters gestellt, wie z.B. in:

```
FUNCTION MyFunction(VAR MyParameter): BOOL
```

Diese Funktionsdeklaration für eine Funktion namens "MyFunction" erwartet dann als Parameter einen Zahlenwert (VAR) und gibt als Ergebnis einen bool'schen Wert zurück (BOOL).

Gültigkeit des übergebenen Parameters

An Funktionen übergebene Parameter sind innerhalb der Funktion lokal, d.h.

- Nur die Funktion selbst kann auf die übergebenen Parameterdaten zugreifen
- Der vergebene symbolische Name ist ebenfalls lokal, d.h. in mehreren Funktion können identische Namen für die Parameter vergeben werden.
- Sie haben keinen Einfluss darauf, welche Variablen oder Merker intern für die Verarbeitung der Parameter und des Funktionsergebnisses verwendet werden.

Aufruf einer Funktion aus dem Programm

Innerhalb des aufrufenden Programmcodes ist es gleichgültig, ob Sie einen Zahlenwert als Variable, Konstante oder Rechenergebnis übergeben bzw. für einen bool'schen Wert einen Merker, einen Ausgang, einen Eingang oder einen Ausdruck übergeben. Deshalb sind für MyFunction() aus unserem obigen Beispiel alle folgenden Aufrufe gültig:

```
MyFunction(R200) / MyFunction(3100.21) / MyFunction(A1.Act + 100 + R97 * 5)
```

Wert aus einer Funktion zurückgeben

Wurde eine Funktion mit einem *Rückgabewert* deklariert, so wird dieser Wert mit dem Befehl RETURN aus der Funktion an das aufrufende Programm zurück gegeben. Natürlich können auch hierbei wieder Rechenergebnisse, Variablen, Verknüpfungen oder Ausdrück zurück gegeben werden.

Verwenden des *Rückgabewert*s einer Funktion

Gibt eine Funktion einen bool'schen Wert (BOOL) zurück, so ist die Funktion innerhalb des aufrufenden Programms gleichzeitig ein bool'scher Ausdruck, d.h. Sie können den Funktionsaufruf auch direkt in eine Abfrage oder eine Schleifenbedinungen integrieren.

Beispiele

```
// Funktion Quadrat() berechnet das Quadrat einer Zahl
FUNCTION Quadrat(VAR Basis): VAR  // Zahl als Rückgabewertr
RETURN Basis * Basis              // Quadrat der übergebenen Zahl als Ergebnis
ENDFUNC                           // Ende der Funktion

R100 = Quadrat(R100)              // Aufruf aus dem Programm: R100² nach R100
R100 = R100 * R100                // Entspricht hier dem gleichen Ergebnis

// Funktion Wait() wartet für eine angegebene Zahl Sekunden
FUNCTION Wait(VAR Zeit)           // Kein Rückgabewert
Zeit = Sys.Timer + Zeit           // Endzeit berechnen
WHILE (Zeit > Sys.Timer)          // Solange Zeit nicht abgelaufen
ENDWHILE                          // Warte
ENDFUNC                           // Ende der Funktion, RETURN nicht nötig

Wait(0.5)                         // Warte eine halbe Sekunde

// Funktion Blink() steuert ein Blinklicht als unabhängige Task
FUNCTION Blink()                  // Keine Parameter, kein Rückgabewert
WHILE (TRUE)                      // Für immer und ewig
     Out17 = !Out17               // Ausgang 17 umschalten
     Wait(0.75)                   // ¾ Sekunde warten (ruft Funktion Wait() auf!)
ENDWHILE                          // Ende der Schleife
ENDFUNC

// Starten der Task
Sys.Task7 = Blink()               // Funktion Blink() als Task 7 starten

// Achsenrückgabewert in Verknüpfung verwenden
FUNCTION AchsenFehler(): BOOL     // Gibt einen bool'schen Wert zurück
RETURN A1.Error | A2.Error        // Rückgabe: Achse 1 oder Achse 2 Fehler
ENDFUNC

IF A1.Inpos & !AchsenFehler()     // Funktion wird als Ausdruck verwendet
```

Hinweise

- Der Funktionsname darf nicht länger als 63 Zeichen sein.
- Der Name darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Funktionsnamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Funktionsnamen ignoriert.
- An Funktionen übergebene Parameter sind stets lokal innerhalb der Funktion.
- Eine Funktion darf niemals mit einem GOTO-Befehl beendet werden. Die einzige gültige Variante eine Funktion zu beenden ist der Befehl RETURN.
- Soll kein Wert zurück gegeben werden, muss der Befehl RETURN am Ende der Funktion nicht zwingend geschrieben werden.
- Falls Sie eine Funktion als Task starten möchten, darf diese Funktion keine Parameter erhalten und auch einen Wert zurück geben. Achten Sie bei der Programmierung einer Funktion als Task daruf, dass eine Schleife in der Funktion programmiert ist – ansonsten wird der Task nach dem ersten Durchlauf automatisch beendet.

Siehe auch

n GOTO

GOTO *Labelname*

Der Befehl GOTO verzweigt die Programmausführung zu dem Befehl, der direkt nach der in *Labelname* angegebenen Sprungmarke zu finden ist.

Funktionsweise

Führe Programm ab *Labelname* aus

Beispiele

```
// Klassische Programmschleife
Loop:                          // Anfang einer Schleife
[Programmcode]
GOTO Loop                      // Springe zu Anfang der Schleife

// Obige Konstruktion programmiert man normalerweise nicht mit Labels, sondern...
While TRUE                     // Für immer und ewig
[Programmcode]
ENDWHILE                       // Springe zu Anfang der Schleife

// Ausspringen aus Schleifen und Abfragen
WHILE (In1 & In17)             // Solange Eingang 1 und Eingang 17
     IF (!A1.Inpos & !Out3)    // Wenn Achse 1 läuft und nicht Ausgang 3
          IF (A1.Error)        // Bei Achsfehler
               GOTO Ende       // Springe aus allen Abfragen
          ENDIF                // Ende der Abfrage
          Out3 = TRUE          // Ausgang 3 einschalten
     ELSE                      // Ansonsten
          G90 A1=200 A2=300    // Starte Achse 1und 2
          Out3 = FALSE         // Schalte Ausgang 3 aus
     ENDIF                     // Ende der Abfrage
ENDIF                          // Ende der Abfrage
Ende:                          // Angesprunges Labels
[Programmcode]
```

Hinweise

- Der Labelname darf nicht länger als 63 Zeichen sein.
- Der Name darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Labelnamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Labelnamen ignoriert.
- Der Labelname muss in der Deklaration stets mit einem Doppelpunkt abgeschlossen werden, beim Aufruf eines Labels ist dies nicht notwendig.

Siehe auch

n IF...ELSE...ENDIF

IF *Ausdruck* ... ELSE ... ENDIF

Der Befehl IF überprüft, ob der angegebene *Ausdruck* den bool'schen Wert TRUE ergibt. Falls dies erfüllt ist, werden alle Befehle bis zu einem ELSE oder ENDIF ausgeführt. Falls nicht, sucht das Programm nach einer ELSE oder ENDIF Anweisung und führt den Programmcode ab dieser Stelle aus.

Funktionsweise

```
Ergibt Ausdruck TRUE?
       Ja:    Führe Programm bis ELSE oder ENDIF aus
       Nein:  Überspringe Programm bis ELSE oder ENDIF
```

Beispiele

```
// Einfache Abfrage mit ELSE Bedingung
IF IN1                            // Wenn Eingang 1 eingeschaltet ist
     OUT1 = TRUE                  // Setze Ausgang 1
     OUT2 = FALSE                 // und lösche Ausgang 2
ELSE                              // ansonsten (wenn Eingang 1 nicht ein)
     OUT1 = FALSE                 // Lösche Ausgang 1
     OUT2 = TRUE                  // und setzte Ausgang 2
ENDIF                             // Ende der Bedingung

// Kombinierte Abfrage Vergleichsbefehl und Ausgang
IF A1.Act + 100 > A1.Soft_End_M & !Out1
                                  // Wenn die aktuelle Achsposition grösser ist als
                                  // der SW-Endschalter und Ausgang 1 ausgeschaltet
     Out1 = TRUE                  // dann schalte Ausgang 1 ein
ENDIF                             // Ende der Bedingung

// Abfrage mit Funktionsergebnis
IF TeachPunktOK()                 // Wenn die Funktion TeachPunktOK() TRUE
     SaveTeachPunkt(ActPunkt)     // liefert dann den akt. Punkt abspeichern
ENDIF
```

Hinweise

- Die Verwendung des Befehls ELSE ist optional und kann weggelassen werden, falls eine Ausführung bei einem bool'schen Ergebnis FALSE nicht benötigt wird.
- Falls gewünscht, kann nach dem Ausdruck THEN zum Einleiten der bedingten Ausführung geschrieben. Der Autrex Compiler ignoriert dieses Wort.
- Der in *Ausdruck* angegebene Wert kann ein einzelner bool'scher Wert - wie ein Merker, Eingang oder Ausgang - sein, eine Vergleichsoperation oder der Verknüpfung von beiden. Wichtig ist lediglich, dass als Endergebnis entweder TRUE oder FALSE festgehalten werden kann.

Siehe auch

n REPEAT...UNTIL

REPEAT... UNTIL *Ausdruck*

Mit dem Befehl REPEAT wird eine Programmschleife eingeleitet. Diese Schleife wird solange durchlaufen, bis am Ende der Schleife der Wert von *Ausdruck* nicht mehr TRUE ergibt.

Funktionsweise

```
Führe Programm bis zum Befehl UNTIL aus
Ergibt Ausdruck TRUE?
       Ja:    Springe zurück zum ersten Befehl nach REPEAT
       Nein:  Setze Programmausführung mit dem Befehl nach UNTIL fort
```

Beispiele

```
// Einfache Warteschleife: Warten auf Eingang 4
REPEAT                              // Wiederhole
[Beliebiger Programmcode hier]
UNTIL IN4                           // bis Eingang 4 eingeschaltet ist

// Achse fahren solange Taste gedrückt ist
G90 A1=50000                        // Achse auf Position 50000 starten
REPEAT                              // Wiederhole
[Beliebiger Programmcode hier]
UNTIL A1.INPOS | A1.ERROR | !DSP.KEY20  // bis Achse in Position, Achsfehler
                                    // oder Taste 20 losgelassen
IF !A1.INPOS & !A1.ERROR            // Wenn nicht in Position und nicht Fehler
     A1.RUN = FALSE                 // Achse anhalten
ENDIF
```

Hinweise

- Bitte beachten Sie, dass unabhängig vom Ergebnis des Ausdrucks REPEAT...UNTIL immer mindestens einmal ausgeführt wird. Um eine Schleife auszuführen und bereits am Anfang der Schleife eine Bedingung zu prüfen, verwenden Sie den Befehl WHILE...ENDWHILE.
- Der in *Ausdruck* angegebene Wert kann ein einzelner bool'scher Wert - wie ein Merker, Eingang oder Ausgang - sein, eine Vergleichsoperation oder der Verknüpfung von beiden. Wichtig ist lediglich, dass als Endergebnis entweder TRUE oder FALSE festgehalten werden kann.

Siehe auch

n RETURN

RETURN [*Wert*]

Mit RETURN wird eine Funktion abgeschlossen. Die Angabe von *Wert* ist nur dann notwendig, wenn die Funktion ein Ergebnis zurück gibt. Falls die Funktion keinen Wert zurück gibt, muss RETURN am Ende einer Funktion nicht zwingend geschrieben werden.

Funktionsweise

```
Springe zurück in das aufrufende Programm
```

Beispiele

```
FUNCTION MyFunction(): BOOL
IF (Out7)                          // Wenn Ausgang 7 bereits eingeschaltet
     RETURN TRUE                   // Springe mit Ergebnis TRUE zurück
ENDIF                              // Ende der Abfrage

IF (In19 & !In20)                  // Wenn Eingang 19 und nicht Eingang 20
     RETURN FALSE                  // Springe mit Ergebnis FALSE zurück
ENDIF

Out7 = TRUE                        // Ausgang 7 einschalten
RETURN TRUE                        // Springe mit Ergebnis FALSE zurück
ENDFUNC                            // Ende der Funktion
```

Hinweise

- Wurde die Funktion mit einem Rückgabewert deklariert, muss der bei RETURN angegebene Wert vom gleichen Datentyp sein, d.h. ist der Rückgabewert BOOL, muss hier ein Merker, Ein-/Ausgang oder ein Ausdruck angegeben werden. Ist der Rückgabewert VAR, ist als Wert bei RETURN ein Zahlenausdruck (Variable, Konstante, Rechenergebnis) erforderlich.
- RETURN muss nicht geschrieben werden, wenn die Funktion keinen Wert zurück gibt.
- Der Befehl RETURN kann in einer Funktion mehrfach und an beliebiger Stelle geschrieben werden.

Siehe auch

n SWITCH...CASE

SWITCH *Wert*...CASE...DEFAULT...ENDCASE

Der Befehl SWITCH ist prinzipiell eine mehrfache IF-Abfrage, die während der Programmausführung mehrere Bedingungen auf einmal prüft. Der angegebene *Wert* muss eine Zahl sein. Es können sowohl Variablen als auch Konstanten angegeben werden, wobei bei diesem Befehl eine Konstante nur bedingt sinnvoll ist.

Nach dem Befehl SWITCH erwartet Autrex mindestens eine CASE Anweisung.

Funktionsweise

```
Ist Wert = erste CASE Bedingung
       Ja:    Führe Programm bis zum nächsten CASE oder ENDCASE aus
              Springe dann zum ersten Befehl nach ENDCASE
       Nein:  Ist Wert = zweite CASE Bedingung?
              Ja:    Führe Programm bis zum nächsten CASE oder ENDCASE aus
                     Springe dann zum ersten Befehl nach ENDCASE
              Nein:  Ist Wert = dritte CASE Bedingung?
                     (beliebige viele weitere CASE Anweisungen)
                     Ansonsten suche DEFAULT oder ENDCASE und führe den dort
                     folgenden Programmcode aus.
```

Beispiel

```
// Überprüfe den Inhalt der Variable R200 auf verschiedene Inhalte und
// ermittle daraus, auf welche Zielposition A1 gefahren werden soll und
// ob die Strombegrenzung aktiviert sein soll.
SWITCH R200                        // Überprüfe Inhalt von R200
     CASE 1:                       // Wenn Inhalt ist 1
          R202 = 112.44            // R202 auf den Wert 112.44 setzen
          G641                     // Strombegrenzung einschalten
     CASE 2:
          R202 = -210.98           // R202 auf Wert −210.98 setzen
          G64                      // Strombegrenzung ausschalten
     DEFAULT:                      // Alle anderen Wert sind ungültig, deshalb
          RETURN                   // verlassen wir die Funktion
ENDCASE                            // Ende der CASE Anweisung

G1 PTP A1=R202                     // Achse 1 auf Inhalt der Variable 202
                                   // verfahren
```

Hinweise

- Funktionell lässt sich jede SWITCH...CASE...ENDCASE Struktur auch über IF...ELSE...ENDIF programmieren; die Variante mit SWITCH ist jedoch sowohl übersichtlicher im Programm als auch schneller in der Programmausführung.

Siehe auch

n WHILE...ENDWHILE

WHILE *Ausdruck*... ENDWHILE

Mit dem Befehl WHILE wird eine Programmschleife eingeleitet. Diese Schleife wird nur dann ausgeführt, wenn der *Ausdruck* TRUE ergibt. Ansonsten wird die Programmausführung mit dem ersten Befehl nach ENDWHILE fortgesetzt.

Funktionsweise

```
Ergibt Ausdruck TRUE?
       Ja:    Führe Programm bis zum Befehl ENDWHILE aus und springe
              zurück auf den Befehl WHILE
       Nein:  Setze Programmausführung mit dem Befehl nach ENDWHILE fort
```

Beispiele

```
// Einfache Warteschleife: Warten auf Eingang 4
WHILE (!IN4)                      // Solange Eingang 4 nicht gesetzt
[Beliebiger Programmcode hier]
ENDWHILE                          // wiederhole

// Achse fahren solange Taste gedrückt ist
G90 A1=50000                      // Achse auf Position 50000 starten
WHILE DSP.KEY20 & (!A1.INPOS & !A1.ERROR)
                                  // Solange Achse nicht in Position,
                                  // kein Achsfehler und Taste 20 gedrückt
[Beliebiger Programmcode hier]
ENDWHILE                          // Wiederhole
IF !A1.INPOS & !A1.ERROR          // Wenn nicht in Position und nicht Fehler
     A1.RUN = FALSE               // Achse anhalten
ENDIF
```

Hinweise

- Bitte beachten Sie, dass die Schleife nur dann ausgeführt wird, wenn das Ausdruck TRUE ergibt. Um eine Schleife zu programmieren, die mindestens einmal ausgeführt wird, verwenden Sie den Befehl REPEAT...UNTIL.
- Der in *Ausdruck* angegebene Wert kann ein einzelner bool'scher Wert - wie ein Merker, Eingang oder Ausgang - sein, eine Vergleichsoperation oder der Verknüpfung von beiden. Wichtig ist lediglich, dass als Endergebnis entweder TRUE oder FALSE festgehalten werden kann.

Siehe auch

3.3 Compiler-Direktiven

Mit den Compiler-Direktiven können Sie bestimmen, ob und wie einzelne Teile des Quellcodes compiliert werden sollen. Die „typische" Compilerdirektive ist dabei natürlich die #IF...#ELSE...#ENDIF Verschachtelung, mit der Sie in Abhängigkeit von symbolischen Definitionen bestimmte Teile des Quelltexts ein- oder ausblenden können.

Des weiteren werden mit Compiler-Direktiven Symbole definiert oder grundsätzliche Eigenschaften des Systems eingestellt, wie z.B. die Anzahl der Nachkommastellen in Variablen.

Grundsätzlich beginnen alle Compiler-Direktiven mit einem Rautensymbol #. Aus diesem Grund darf das Zeichen # auch nicht in Labels oder Symbolnamen verwendet werden.

n Wirkungsweise von Compiler-Direktiven

Compiler-Direktiven werden nur vom Compiler ausgewertet und erzeugen keinen ausführbaren Programmcode. Genau darin liegt der Vorteil der Compiler-Direktiven: Sie können z.B. ein Programm mit verschiedenen Werkzeugoptionen entwickeln und dann in der Kundenversion ausschliesslich die Optionen mitcompilieren, die auch tatsächlich benötigt werden.

So können #IF...#ELSE...#ENDIF Konstruktionen eingesetzt werden, um Programmspeicherplatz zu sparen. Weiterhin wirkt sich natürlich ausgeblendeter Code positiv auf die Ausführungsgeschwindigkeit des Programms aus.

n #DECIMAL

#DECIMAL Nachkommastellen

Mit der Compiler-Direktive #DECIMAL geben Sie an,wie viele Nachkommastellen Sie in nummerischen Werten wünschen. Eine einmal gemachte Definition der Nachkommastellen kann in einem Programm nicht mehr verändert werden und gilt für alle Zahlenwerte in diesem Programm.

Autrex unterstützt bis zu 3 Nachkommastellen. Demnach sind alle Werte von „0" (keine Nachkommastellen) bis „3" (drei Nachkommastellen) gültige Werte für die Compiler-Direktive #DECIMAL.

Funktionsweise

Lege die Anzahl der *Nachkommastellen* fest

Beispiele

```
#DECIMAL 2                      // Zwei Nachkommastellen in diesem Programm
                                // verwenden
R100 = 2.02                     // Variable 100 auf den Wert 2.02 setzen
R100 = 2                        // Variable 100 auf den Wert 2 setzen
R100 = 2.00                     // identisch mit vorigem Befehl
R100 = 2.999                    // Führt zu Compilerfehler, da mehr Kommastellen
                                // angegeben werden als zuvor definiert wurden
```

Hinweise

- Im Programm müssen die Nachkommastellen nicht zwingend geschrieben werden. Wird innerhalb einer Zuweisung keine Nachkommastelle verwendet, so muss das dezimale Komma auch nicht geschrieben werden.
- Wird eine Zuweisung mit mehr Kommastellen als definiert angegeben, so führt dies zu einem Compilerfehler.
- Wird die Compiler-Direktive #DECIMAL nicht geschrieben, so arbeitet Autrex mit der Standardeinstellung von einer Nachkommastelle (1/10 Genauigkeit).
- Intern speichert Autrex ausschliesslich 32-Bit Integerwert. Die Festkomma-Zahlenformat wird emuliert, indem der interne ganzzahlige Wert entsprechend oft mit 10 multipliziert wird.

Wertigkeitsbereiche

Je nach eingestellter Anzahl der Nachkommastellen können maximal folgende Zahlenwerte gespeichert werden:

Nachkommastellen	Grösster positiver Wert	Kleinster negativer Wert
0	2147483647	-2147483648
1	214748364.7	-214748364.8
2	21474836.47	-21474836.48
3	2147483.647	-2147483.648

n Tabelle 2 –Wertigkeitsbereiche in Abhängigkeit der Nachkommastellen

Siehe auch

n #DEFINE

#DEFINE *Symbolname [Wert]*

Mit der Compiler-Direktive #DEFINE wird der angegebene *Symbolname* auf den angegebenen *Wert* gesetzt. Der Wert kann hierbei einen beliebigen Datentyp enthalten, also eine Variable, einen Bitmerker, einen Ausgang oder einen Eingang. Es ist jedoch nicht zulässig, mehr als einen Wert in eine #DEFINE Direktive einzubinden. Um dies zu realisieren, verwenden Sie bitte die Direktive #MACRO.

Falls in der #DEFINE Direktive kein *Wert* angegeben wird, so setzt Autrex das Symbol automatisch auf den bool'schen Wert TRUE.

Automatische Ressourcenvergabe

Falls Sie es wünschen, kann Autrex bei Variablen und Merkern selbsttätig eine Variablen- bzw. Merkernummer zuordnen. In diesem Fall lassen Sie bei der Definition des Symbols die Nummer der Variable bzw. des Merkers einfach weg.

Funktionsweise

```
Ist ein Wert angegeben?
       Ja:    Symbolname = Wert
       Nein:  Symbolname = TRUE
```

Beispiele

```
#DEFINE NotausKreis In17          // Definiere das Symbol „NotausKreis“ als In17
IF (!NotausKreis)                 // Wenn der Notauskreis offen ist
     [Notaus Behandlung]          // Behandlung für den Notaus-Fall
ENDIF

#DEFINE Zähler R100               // Definiere das Symbol „Zähler“ als Register 100
For Zähler = 1 To 10              // Zehn Schleifendurchläufe
     Out3 = !Out3                 // Ausgang 3 umschalten
     Wait(0.5)                    // Eine halbe Sekunde warten
Next

#RESERVE R100-R500                // Variablen 100 bis 500 dürfen nicht von Autrex
                                  // automatisch verwendet werden
#DEFINE Zähler R                  // Definiere das Symbol „Zähler“ auf eine
                                  // beliebige freie Variable
For Zähler = 1 To 10              // Zehn Schleifendurchläufe
     [Programmcode]
Next
```

Hinweise

- Der Symbolname darf nicht länger als 63 Zeichen sein.
- Der Name darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Symbolnamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Symbolnamen ignoriert.
- Sollen bestimmte Merker- und Variablenbereiche nicht von Autrex automatisch verwaltet werden, verwenden Sie die Compiler-Direktive #RESERVE um diese Bereiche zu reservieren.

Siehe auch

n #DISPLAY

#DISPLAY Displayobjekt Displaytyp

Wie in Kapitel 4.11 - DSP: Das Displayobjekt (Seite 62) beschrieben wird, werden Tastaturabfragen in Autrex durch Zugriffe auf ein Objekt durchgeführt. Zur einfacheren Programmierung können Sie mit der Compilerdirektive #DISPLAY einem *Displayobjekt* einen bestimmten *Displaytyp* zuordnen. Als Folge können alle Tasten dieser Frontplatte bzw. dieses Bedienpults dann statt über die Tastennummer über einen sprechenden Tastennamen abgefragt werden. Eine Übersicht der verfügbaren, vordefinierten Displays finden Sie in Anhang B - Tastaturabfragen (Seite 74).

Funktionsweise

Typ von *Displayobjekt = Displaytyp*

Beispiel

```
// Normale Tastaturabfrage (Standard-Autrex)
IF DSP1.KEY31                          // Ist die Taste 31 gedrückt?
ENDIF

// Tastaturabfrage mit zugewiesenem Displaytyp
#DEFINE DSP1 HT
IF DSP1.N-PLUS                         // Ist die Taste N+ gedrückt?
ENDIF
```

Hinweise

- Sie können eigene Displaytypen definieren. Hierzu editieren Sie die Datei DISPLAYS.AXD, kopieren eine Displaydefinition und fügen die kopierte Displaydefinitionsdatei in Ihr Projekt ein. Ändern Sie dann die Displaybezeichnung und konfigurieren die Tasten nach Ihren Wünschen.
- Tastennamen dürfen nicht länger als 63 Zeichen sein.
- Der Name einer Taste darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Tastennamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Tastennamen ignoriert.

Siehe auch

Kapitel 4.11 - DSP: Das Displayobjekt (Seite 62)
Anhang B - Tastaturabfragen (Seite 74)

n #IF...#ELSE...#ENDIF

#IF *Ausdruck* ... #ELSE ... #ENDIF

Die Compiler-Direktive #IF überprüft, ob der angegebene *Ausdruck* den bool'schen Wert TRUE ergibt. Falls dies erfüllt ist, werden der Programmcode bis zu einem #ELSE oder #ENDIF compiliert. Falls nicht, sucht das Programm nach einer #ELSE oder #ENDIF Direktive und compiliert den Programmcode ab dieser Stelle weiter.

Funktionsweise

```
Ergibt Ausdruck TRUE?
       Ja:    Compiliere Programmcode bis ELSE oder ENDIF
       Nein:  Ignoriere Programmcode bis ELSE oder ENDIF
```

Beispiele

```
#DEFINE Achsen  2                  // Symbol „Achsen“ auf den Wert „2“ setzen

#IF ACHSEN > 2                     // Wenn mehr als zwei Achsen vorhanden sind
     AchsInit(2)                   // Zweite Achse initialisieren
#ENDIF

#DEFINE Interface_Montforts        // Symbol “Interface-Montforts” definieren
#IF Interface_Montforts            // Wenn das Symbol definiert ist
     A14 = TRUE                    // Ausgang 14 einschalten
     A19 = FALSE                   // Ausgang 19 ausschalten
#ELSE                              // Ansonsten
     A15 = TRUE                    // Ausgang 15 einschalten
#ENDIF
```

Hinweise

- Im Gegensatz zum IF...ELSE...ENDIF Strukturbefehl wird bei der #IF...#ELSE...#ENDIF Compilerdirektive der eingebundene Code tatsächlich nicht mit compiliert und auch nicht in der Steuerung gespeichert.
- Die Verwendung der Direktive #ELSE ist optional und kann weggelassen werden, falls eine Ausführung bei einem bool'schen Ergebnis FALSE nicht benötigt wird.
- Falls gewünscht, kann nach dem Ausdruck THEN zum Einleiten der bedingten Ausführung geschrieben. Der Autrex Compiler ignoriert dieses Wort.
- #IF...#ELSE...#ENDIF Konstruktionen können bis zu 16 Ebenen tief geschachtelt werden.

Siehe auch

n #MACRO...#ENDMACRO

#MACRO Makroname Makrotext ... #ENDMACRO

Die Compiler-Direktive #MACRO definiert ein einfaches Textmakro. Makros sind in erster Linie sinnvoll, um sich für immer wiederkehrende Formulierungen und Abfragen die Tipparbeit zu sparen.

Als *Makrotext* kann ein beliebiger Ausdruck angegeben werden. Der Inhalt des Ausdrucks innerhalb der Makrodefinition nicht überprüft, sondern lediglich zwischengespeichert. Im Programmcode wird bei Aufruf des Makros dann exakt der definierte Makrotext eingefügt.

Makro oder Funktion?

In vielerlei Hinsicht ähneln sich Makros und Funktionen – sie dienen beide zur Vereinfachung immer wiederkehrender Aufgaben und der Reduzierung des Tippaufwands. Prinzipiell hat die Verwendung von Funktionen jedoch einen massiven Vorteil: Makros belegen bei jedem Aufruf Programmspeicher für den defnierten Makrotext, Funktionen benötigen diesen Speicher nur einmal.

Dennoch kann die Verwendung von Makros in speziellen Fällen sinnvoll sein, z.B. wenn man nur den Teil einer kombinierten Abfrage als Makro automatisieren möchte oder der Makrotext so kurz ist, dass sich eine eigene Funktion hierfür nicht lohnt.

Funktionsweise

Makroname = Makrotext

Beispiele

```
#MACRO LampeRot                  // Definiere das Makro „LampeRot“
Out19 = TRUE                     // Ausgang 19 ein, Ausgang 20 und 21 aus
Out20 = FALSE
Out21 = FALSE
#ENDMACRO                        // Ende des Makros

LampeRot                         // Makro im Programm verwenden. Der Makrotext
                                 // wird an dieser Stelle vollständig eingetragen

// Das gleiche als Funktion programmiert:

Function LampeRot()              // Definiere die Funktion „LampeRot“
Out19 = TRUE                     // Ausgang 19 ein, Ausgang 20 und 21 aus
Out20 = FALSE
Out21 = FALSE
EndFunc                          // Ende der Funktion

LampeRot()                       // Funktion im Programm aufrufen. Es wird nur
                                 // der Funktionsaufruf eingetragen, nicht der
                                 // vollständige Inhalt der Funktion.
```

Hinweise

- Der Makroname darf nicht länger als 63 Zeichen sein.
- Der Name darf nur die Buchstaben von A-Z sowie Unterstriche, Binderstriche und die Ziffern 0-9 enthalten. Alle anderen Zeichen, wie z.B. das Leerzeichen, Umlaute oder sonstige Sonderzeichen, sind innerhalb von Makronamen nicht erlaubt.
- Die Groß- und Kleinschreibung wird in Makronamen ignoriert.

Siehe auch

n #MC90...#ENDMC90

#MC90 [Programmcode in MC90-Sprache] #ENDMC90

Um einen Programmteil innerhalb eines Autrex-Programms in der bisherigen MC90-Sprache zu programmieren, schalten Sie den Compiler mit der Direktive #MC90 in den Kompatibilitätsmodus. In der Folge dieser Direktive können Sie beliebig viele Befehle in der bisherigen MC90-Sprache formulieren. Dieser Block wird mit der Compiler-Direktive #ENDMC90 abgeschlossen.

Bitte beachten Sie, dass innerhalb der MC90-Programmblöcke alle mit #DEFINE defnierten Symbole nicht zur Verfügung stehen.

Funktionsweise

Schalte den Autrex-Compiler in den Kompatibilitätsmodus

Beispiele

```
IF (In17 & R200 == 100)                    // Wenn Bedingung erfüllt…
#MC90                                      // MC90 Programmblock starten
91-E 4000Q 1020V 91M;                      // Programmcode in MC-Sprache
#ENDMC90                                   // Ende des MC90 Programmblocks
```

Hinweise

- Es stehen alle MC90-Befehle des eMC200-Systems zur Verfügung. Nicht unterstützt wird der Zugriff auf Systemfunktionen, wie z.B. das Interrupt-Handling.
- In MC90-Blöcken dürfen keine Unterprogramme definiert oder aufgerufen werden.
- Sprünge, auch aus einem MC90-Programmblock in einen Autrex-Programmblock, sind gültig.

Siehe auch

n #RESERVE

#RESERVE Startvariable-Endvariable
#RESERVE Startmerker-Endmerker

Wie bei der Compiler-Direktive #DEFINE beschrieben, erlaubt Autrex die automatische Zuordnung von freien Variablen- und Merkeradressen zu Symbolen.

Falls Sie jedoch in einem Programm Variablen indirekt adressieren oder ganze Variablenbereiche z.B. als Werkstückspeicher verwenden, kann Autrex nicht automatisch ermitteln, dass die entsprechenden Bereiche bereits belegt sind. Mit der Compiler-Direktive #RESERVE teilen Sie Autrex deshalb mit, dass ein bestimmter Bereich des Variablen- bzw. Merkerspeichers bereits verwendet wird und nicht durch Autrex automatisch verwaltet werden darf.

Funktionsweise

Kennzeichne den Bereich von *Start* bis *Ende* als reserviert

Beispiele

```
#RESERVE R100-R500              // Variablen 100 bis 500 dürfen nicht von Autrex
                                // automatisch verwendet werden
#RESERVE R1000-R1020            // Variable 1000 bis 1020 ebenfalls reserviert
#DEFINE Zähler R                // Definiere das Symbol „Zähler“ auf eine
                                // beliebige freie Variable, aber nicht auf eine
                                // Variable aus dem zuvor reservierten Bereich
For Zähler = 1 To 10            // Zehn Schleifendurchläufe
     [Programmcode]
Next

#RESERVE M123-M189              // Merker von 123 bis 189 dürfen nicht von Autrex
                                // automatisch verwendet werden
#DEFINE Alarm M                 // Beliebigen Merker als Symbol „Alarm“
                                // definieren, aber nicht aus dem Bereich von
                                // Merker 123 bis Merker 189
```

Hinweise

- Bei der automatischen Vergabe von Ressourcen läuft Autrex von oben nach unten, sprich: es werden zunächst die obersten Variablen- bzw. Merkernummern automatisch vergeben, solange bis Autrex auf einen reservierten Bereich stösst. Dieser Bereich wird dann in der automatischen Vergabe der Variablen- bzw. Merkernummern übersprungen.
- Die Direktive #RESERVE kann beliebig oft im Programmcode verwendet werden.
- Verschiedene reservierte Bereiche dürfen sich überlappen.
- Die #RESERVE Direktive muss am Anfang des Programms stehen. Finden sich #RESERVE Direktiven nach dem ersten Programmcode, führt dies zu einer Warnung und die Direktive wird ignoriert.

Siehe auch

3.4 Achsprogrammierung in DIN

Autrex implementiert einen Teil der DIN-Programmiersprache zur Achsansteuerung. Diese Implementation dient in erster Linie der Kompatibilität in der Achsprogrammierung zu den verbreitesten Werkzeugmaschinen.

Intern werden alle DIN-Befehle wiederum in die Autrex-Programmiersprache umgesetzt. Sie können also Autrex entweder mit DIN oder direkt in Autrex programmieren – Ihre Wahl hat keine Auswirkungen auf die Funktionsweise oder die Ausführungsgeschwindigkeit der Achsbefehle.

n Intergration in Autrex

Die Achsprogrammierung in DIN kann nahezu beliebig mit allen anderen Autrex-Befehlen gemischt werden. Lediglich nach dem Start eine Achsbewegung in DIN ist es nicht erlaubt, in der gleichen Zeile zusätzliche Autrex-Befehle anzugeben.

n DIN-Befehlssatz

Folgende DIN-Befehle werden in Autrex unterstützt:

Befehl	Funktion
G0	Eilgang (maximale Geschwindigkeit gemäß Achsparametern)
G1	Vorschub (programmierte Geschwindigkeit)
G641	Strombegrenzung ein
G64	Strombegrenzung aus
G90	Absolute Positionierung
G91	Relative Positionierung
PTP	„Point-To-Point" – Positioniere und warte, bis die Achsen die Zielposition erreicht haben oder ein Fehler aufgetreten ist.
LIN	Führe die Positionierung als lineare Interpolation durch
ADIS	Programmiere Verschleifung zweier Achsen in einer Bewegung

n Tabelle 3 –Unterstützte DIN-Befehle zur Achsprogrammierung

Alle DIN-Befehle können – auch innerhalb einer Zeile – kombiniert werden. Grundsätzlich darf nach dem Einleiten einer Positionierung mit G90, G91, PTP oder LIN kein weiterer Befehl mit Ausnahme der Achszielpositionen geschrieben werden.

n Achsen Systemobjekt

Auch bei einer Programmierung der Achsbewegung in DIN-Sprache wirken sich alle Aktionen natürlich direkt auf die Achsen Systemobjekte aus. Detaillierte Informationen zu diesen Objekten erhalten Sie im Kapitel ?.

n G0 / G1 – Geschwindigkeitsauswahl

Mit dem DIN-Befehl G0 schalten Sie die Geschwindigkeit für die nächsten Bewegungen auf die maximale parametrierte Geschwindigkeit für die jeweiligen Achsen, mit G1 schalten Sie zurück auf „Vorschub" – d.h. auf die programmierte Geschwindigkeit der Achsen.

Beispiele

```
G0 PTP G90 A1=R100 A2=R101            // Schalte in Eilgang, fahre Achse 1 auf
                                      // die in Variable 100 gespeicherte Position
                                      // Achse 2 auf die in Variable 101
                                      // gespeicherte Position und warte bis die
                                      // Achsen in Position sind
G1                                    // Schalte zurück in programmierte Speed

// Kombination DIN und Autrex Sprachelemente
If MaxSpeed                           // Wenn Merker MaxSpeed gesetzt
     G0                               // Schalte in Eilgang
Else                                  // ansonsten
     G1                               // Fahre mit programmierter Geschwindigkeit
Endif
PTP G90 A1=R100 A2=R101               // Positionierung starten
```

Hinweise

- Der Geschwindigkeitswechsel wirkt sich stets auf die nächste programmierte Bewegung aus, nicht aber auf ggf. noch laufende Bewegungen.

Siehe auch

n G641 / G64 – Strombegrenzung ein/aus

Der DIN-Befehl G641 schaltet die Strombegrenzung für die nächsten Bewegungen ein, mit dem Befehl G64 wird die Strombegrenzung abgeschaltet.

Beispiele

```
G641 PTP G91 A1=200                   // Strombegrenzung ein, Achse 1 um die
                                      // relative Strecke 200 verfahren
G64                                   // Strombegrenzung wieder ausschalten

// Kombination DIN und Autrex Sprachelemente
If In12 & In17                        // Wenn Eingang 12 und Eingang 17
     G641                             // Strombegrenzung einschalten
Else                                  // ansonsten
     G64                              // Strombegrenzung auschalten
Endif
PTP G91 A1=200                        // Positionierung starten
```

Hinweise

- Das Ein- oder Ausschalten der Strombegrenzung wird sofort ausgeführt – unabhängig davon, ob gegenwärtig noch eine Positionierung läuft. Sie müssen in Ihrem Programm dafür Sorge tragen, dass die laufende Bewegung abgeschlossen ist, bevor Sie die Strombegrenzung umschalten.

Siehe auch

n G90 / G91 – Achse absolut/relativ verfahren

Mit dem DIN-Befehl G90 wird eine absolute Fahrbewegung eingeleitet. Der Befehl G91 hingegen leitet eine relative Fahrbewegung ein.

Beispiele

```
G90 A1=R100 A2=R101                    // Achse 1 auf die Absolutposition aus
                                       // Variable 100, Achse 2 auf die Position
                                       // aus Variable 101 verfahren.
G91 A1=100 A2=R100                     // Achse 1 um die realtive Strecke 100,
                                       // Achse 2 um die in Variable 100
                                       // gespeicherte relative Strecke verfahren
G90 A1=R100 G91 A2=R101                // Kombinierte absolute/relative Bewegung

// Kombination DIN und Autrex Sprachelemente
G90 A1=R100 A2=R101                    // Bewegung starten
While !A1.Inpos & !A1.Error            // Solange weder Inpos noch Achsfehler...
     Out17 = A1.Dist < 100             // Schalte Ausgang 17 ein, wenn die Achse 1
Endwhile                               // weniger als 100 vom Ziel entfernt ist
```

Hinweise

- Mit den DIN-Befehlen G90 bzw. G91 werden Positionierungen gestartet, jedoch nicht abgewartet, dass diese Positionierungen auch beendet werden. Um nach dem Start eine Positionierung auf deren Ende zur warten, verwenden Sie zusätzlich den DIN-Befehl PTP.
- Vor dem Start der Bewegung überprüft Autrex, ob alle vorhergehenden Bewegungen für diese Achse vollständig beendet sind, das Leistungsteil bereit meldet und keine Achsstörung vorliegt. Nur dann wird die Bewegung ausgeführt. Ansonsten wartet Autrex in der gleichen Programmzeile auf das Ende der vorigen Bewegung bzw. die Behebung der Fehlerzustände.
- Relative und absolute Fahrbewegungen können in der gleichen Positionieranweisung gemischt werden.

Siehe auch

n PTP – Bewegung einleiten und auf Ende warten

In Kombination mit einem der DIN-Befehle G90 oder G91 startet der Befehl PTP (Point-To-Point) eine Fahrbewegung und wartet in der gleichen Programmzeile auf das Ende der Bewegung.

Beispiele

```
PTP G90 A1=R100 A2=R101                // Achse 1 auf die Absolutposition aus
                                       // Variable 100, Achse 2 auf die Position
                                       // aus Variable 101 verfahren. Warte dann,
                                       // bis die Achsen die Zielposition erreicht
                                       // haben
```

Hinweise

- Nach dem Befehl PTP dürfen ausser der Achspositionierung mit G90 bzw. G91 oder der Einleitung einer Interpolation mit LIN keine Anweisungen mehr folgen.

Siehe auch

n LIN – Lineare Interpolation

Mit dem DIN-Befehl LIN wird in Kombination mit einem der DIN-Befehle G90 oder G91 eine linear interpolierte Bewegung eingeleitet.

Beispiele

```
LIN G90 A1=R100 A2=R101              // Achse 1 auf die Absolutposition aus
                                     // Variable 100, Achse 2 auf die Position
                                     // aus Variable 101 linear interpoliert
                                     // verfahren

PTP LIN G90 A1=R100 A2=R101          // Gleiche Bewegung wie oben, jedoch wartet
                                     // Autrex in der gleichen Zeile auf das Ende
                                     // der Positionierung
```

Hinweise

- Nach dem Befehl LIN dürfen ausser der Achspositionierung mit G90 bzw. G91 keine Anweisungen mehr folgen.

Siehe auch

n ADIS – Verschleifung programmieren

Normalerweise werden bei der Programmierung von Achsbewegungen mit G90 oder G91 alle beteiligten Achsen gleichzeitig gestartet. Durch den Befehl ADIS innerhalb der Positionieranweisung kann jedoch bewirkt werden, dass die zweite und alle folgenden Achsen erste dann gestartet werden, wenn die vorhergehende Achse Ihre Bewegung bis auf einen angegebenen Restweg durchgeführt hat. Diese Funktion nennt man „Verschleifen".

Beispiele

```
G90 A1=R100 ADIS=R101 A2=R102        // Achse 1 auf die Absolutposition aus
                                     // Variable 100. Die Achse 2 wird erst dann
                                     // auf die Absolutposition in R102 gestartet
                                     // wenn der Restweg der ersten Achse nicht
                                     // mehr grösser ist als der Wert in R101.

// Mehrfache Verschleifung
G90 A1=R100 ADIS=500 A2=R101 ADIS=200 A3=R102
                                     // Hier wird zunächst die Achse 1 auf die
                                     // Position in R100 gestartet. Liegt der
                                     // Restweg unter 500, wird die Achse 2 auf
                                     // auf die Position in R101 gestartet. Liegt
                                     // der Restweg der Achse 2 unter 200, wird
                                     // die Achse 3 gestartet.
```

Hinweise

- Die Verschleifung bezieht sich stets auf die Achse, der Zielposition nach dem Befehl ADIS angegeben wird.
- Interpolierte Bewegungen und Verschleifungen können nicht kombiniert werden.

Siehe auch

3.5 Vordefinierte Funktionen

Einige Funktionen werden über die Dateien AUTREX.AXS und AUTREX.AXD bereits automatisch in Ihre Programmierumgebung integriert. Diese Funktion sind vollständig in Autrex programmiert und können natürlich auch von Ihnen verändert werden.

Name	Funktion
WAIT(Sekunden)	Wartet die angegebene Anzahl Sekunden bis die Programmausführung fortgesetzt wird
TASK_DIAG()	Vordefinierter Diagnosetask. Beim Systemstart wird automatisch die Task 15 mit auf diese Diagnosefunktion gesetzt.

n Tabelle 4 –Vordefinierte Funktionen in Autrex

n Raum für Ihre Notizen

Kapitel 4 Systemobjekte

In der Erläuterung der Datentypen in Autrex als auch in der Befehlsübersicht wurden bereits mehrfach die Systemobjekte erwähnt, die einen direkten Zugriff auf alle Ressourcen des Systems erlauben. In diesem Kapitel finden Sie nun eine Übersicht der verfügbaren Systemobjekte sowie eine Erläuterung der einzelnen Funktionen und Datenfelder.

n Liste der Systemobjekte

Folgende Systemobjekte sind in Autrex verfügbar:

Objekt	Funktion
SYS	Zugriff auf alle Parameter und Funktionen der Steuerung an sich
TASK1 bis TASK16	Zugriff auf bis zu 16 unabhängige Paralleltasks
PRO1 bis PRO2	Zugriff auf bis zu 2 Profibus-Erweiterungsmodule (Profibus DP Slave)
MODEM	Zugriff auf ein angeschlossenes Steuerungsmodem
A1 bis A8	Zugriff auf alle Funktionen und Zustände von bis zu 8 Achsen
OUT1 bis OUT512	Zugriff auf bis zu 512 digitale Ausgänge
IN1 bis IN512	Zugriff auf bis zu 512 digitale Eingänge
CBOX1 bis CBOX19	Zugriff auf den Status von bis zu 19 c:Box Modulen
AIN1 bis AIN16	Zugriff auf bis zu 16 analoge Eingänge
AOUT1 bis AOUT16	Zugriff auf bis zu 16 analoge Ausgänge
DSP1 bis DSP2	Zugriff auf alle Funktionen und Zustände von bis zu 2 Displayeinheiten
CNET	Zugriff auf einen eMC200 Steuerungsvernetzungsring
COGNEX	Zugriff auf ein angeschlossenes Cognex Bilderkennungssystem
FLASH	Zugriff auf die integrierte Silicon Disc oder eine externe Speichercassette

n Tabelle 5 –Verfügbare Systemobjekte in Autrex

Jedes Systemobjekt kann wieder über eine Anzahl Unterobjekte verfügen, wie z.B. bei den Taskobjekten die jeweils zugeordneten Timer, die aktuelle Programmzeile usw. Eine detaillierte Übersicht über alle Systemobjekte finden Sie auf den nächsten Seiten.

n Indizierter Zugriff auf Objekte

Alle Systemobjekte, die mehr als einmal in Autrex vorhanden sind, können sowohl direkt als auch indirekt (indiziert) angesprochen werden. Die indizierte Adressierung ermöglicht die Verwendung von Systemobjekten z.B. in Programmschleifen.

```
Achse 1 direkt: A1          Achse 1 indiziert: A[1]
Eingang 15 direkt: IN15     Eingang 15 indiziert: IN[15]
```

So können Sie z.B. mit folgender Schleife alle Ausgänge löschen:

```
#DEFINE Zähler R                    // Zählervariable definieren
FOR Zähler = 1 TO 512               // Alle Ausgänge…
     OUT[Zähler] = FALSE            // ...löschen
NEXT
```

4.1 SYS: das Basisobjekt

Über das SYS Objekt können alle Systemelemente der Steuerung angesprochen werden. Das SYS Objekt besteht aus folgenden Elementen und Unterobjekten:

Element	Datentyp	Bedeutung	Zugriff
ADDRESS	Zahl	Steuerungsadresse. Gültige Werte liegen zwischen 1 und 255. Eine geänderte Steuerungsadresse wird erst mit dem nächsten Reset der Steuerung gültig.	Lesen/Schreiben
AXIS	Zahl	Gibt die Anzahl der angeschlossenen Achscontroller zurück.	Nur lesen
BAUDRATE	Zahl	Aktuelle Baudrate für die serielle Kommunikation mit dem PC. Gültige Werte sind 4 (9600 Baud), 6 (19200 Baud) und 7 (38400 Baud). Eine geänderte Baudrate wird erst mit dem nächsten Reset der Steuerung gültig.	Lesen/Schreiben
DATE	Zahl	Aktuelles Datum der Echtzeituhr. Kann gelesen oder geschrieben werden. Das Datum ist im Format JJMMTT codiert, also z.B. 030417 für den 17.04.2003.	Lesen/Schreiben
DEBUG	Boolean	Wenn TRUE, wird der MC90 Programmcode der aktuell ausgeführten Zeile im Display angezeigt.	Lesen/Schreiben
DISPLAYS	Zahl	Gibt die Anzahl der angeschlossenen Displays zurück.	Nur lesen
FASTBUS	Objekt	Erweiterte Informationen über die Kommunikation des Systems mit über den FastBus angeschlossenen Modulen. Eine detaillierte Beschreibung finden Sie weiter unten.	Nur lesen
FREEMEM	Boolean	Anzahl der freien Bytes im SPS-Programmspeicher.	Nur lesen
OVERRIDE	Zahl	Gibt die aktuelle Stellung des Overrides in Prozent zurück.	Nur lesen
PARTS	Zahl	Gesamtstückzähler	Lesen/Schreiben
PROFIBUS	Zahl	Gibt die Anzahl der angeschlossenen Profibus DP-Slaves zurück.	Nur lesen
RESET	Boolean	Wird dieses Element vom SPS-Programm auf TRUE gesetzt, führt die Steuerung einen Neustart (RESET) durch	Nur schreiben
STEP	Boolean	Wird dieses Element vom SPS-Programm auf TRUE gesetzt, wird ein Einzelschritt im Programm ausgeführt. Nur gültig, wenn SYS.STOP den Zustand TRUE hat.	Nur schreiben
STOP	Boolean	Wird dieses Element vom SPS-Programm auf TRUE gesetzt, wartet das System vor jeder Code-Zeile auf eine Bestätigung via Tastatur (N+).	Nur schreiben
TIME	Zahl	Aktuelle Uhrzeit der Echtzeituhr. Kann gelesen oder geschrieben werden. Die Uhrzeit ist im Format HHMMSS codiert, also z.B. 162349 für 16:23.49.	Lesen/Schreiben
TIMER	Zahl	Interner Systemtimer, wird als Zeitbasis für alle anderen SPS-Timer verwendet.	Nur lesen
VERSION	Objekt	Unterobjekt mit den Versionsnummern aller intelligenten angeschlossenen Module. Eine detaillierte Beschreibung finden Sie weiter unten.	Nur lesen

n Tabelle 6 –Elemente und Unterobjekte des SYS Objekts

Beispiele

```
IF (SYS.ADDRESS != 3)                    // Wenn Steuerungsadresse nicht 3
     SYS.ADDRESS = 3                     // Steuerungsadresse auf 3 setzen
     SYS.RESET = TRUE                    // Steuerung neu starten
ENDIF

IF (SYS.BAUDRATE != 7)                   // Wenn Baudrate nicht auf 38400 Baud
     SYS.BAUDRATE = 7                    // Baudrate setzen
     SYS.RESET = TRUE                    // Steuerung neu starten
ENDIF
```

n Unterobjekt FASTBUS (SYS.FASTBUS)

Im Unterobjekt FASTBUS des SYS Objekts werden zusätzliche Informationen über via FastBus angeschlossene Komponenten bereitgestellt.

Element	Datentyp	Bedeutung	Zugriff
COUNT	Zahl	Anzahl der am FastBus angeschlossenen Module.	Nur lesen
CRC	Zahl	Anzahl der Checksummen-Fehler in der Kommunikation über den Fastbus.	Lesen/Schreiben
TIMEOUT	Zahl	Anzahl der Timeout-Fehler in der Kommunikation über den Fastbus.	Lesen/Schreiben

n Tabelle 7 –Elemente des Unterobjekts FASTBUS (SYS.FASTBUS)

n Unterobjekt VERSION (SYS.VERSION)

Im Unterobjekt VERSION des SYS Objekts werden die Versionsnummern alle am System angeschlossenen intelligenten Module bereitsgestellt.

Element	Datentyp	Bedeutung	Zugriff
CBOX	Zahl	Versionsnummer der c:Box Module	Nur lesen
CPU	Zahl	Versionsnummer der CPU	Nur lesen
CPUCO	Zahl	Versionsnummer des CPU Co-Prozessors	Nur lesen
DSP1	Zahl	Versionsnummer des ersten angeschlossenen Displays	Nur lesen
DSP2	Zahl	Versionsnummer des zweiten angeschlossenen Displays	Nur lesen
MOC	Zahl	Versionsnummer der Servomotocontroller	Nur lesen
PROFIBUS	Zahl	Versionsnummer des Profibus DP-Slaves	Nur lesen
RES1	Zahl	Reserviert für spätere Verwendung / neue Module	Nur lesen
RES2	Zahl	Reserviert für spätere Verwendung / neue Module	Nur lesen
RES3	Zahl	Reserviert für spätere Verwendung / neue Module	Nur lesen

n Tabelle 8 –Elemente des Unterobjekts VERSION (SYS.VERSION)

4.2 TASK: Das Taskobjekt

In Autrex existiert das Task-Objekt in 16-facher Ausführung: für jede Task gibt es ein separates Objekt. Die einzelnen Task-Objekte werden einfach nummeriert angesprochen, also TASK1 für die erste Task, TASK2 für die zweite Task usw.

Element	Datentyp	Bedeutung	Zugriff
FUNC	Funktion	Funktion, die als separate Task ausgeführt wird. Wird vom SPS-Programm gesetzt, bevor RUN auf TRUE gesetzt wird.	Nur Schreiben
RUN	Boolean	TRUE wenn die Task aktiv ist. Kann vom SPS-Programm gesetzt werden, um eine Task zu starten oder anzuhalten.	Lesen/Schreiben
TIM1	Objekt	Timer-Objekt für den ersten SPS-Timer dieser Task	Lesen/Schreiben
TIM2	Objekt	Timer-Objekt für den zweiten SPS-Timer dieser Task	Lesen/Schreiben
TIM3	Objekt	Timer-Objekt für den dritten SPS-Timer dieser Task	Lesen/Schreiben
TIM4	Objekt	Timer-Objekt für den vierten SPS-Timer dieser Task	Lesen/Schreiben

n Tabelle 9 –Elemente des TASK Objekts

Beispiele

```
TASK4.FUNC = MeineTask()              // Funktion "MeineTask()" als Task 4
                                      // definieren
TASK4.RUN = TRUE                      // Task 4 starten
```

n Unterobjekte TIM1 bis TIM4 (TASKx.TIMx)

Über die Unterobjekte TIM1 bis TIM4 eines jeden Task-Objekts können die SPS-Timer abgefragt oder parametriert werden.

Element	Datentyp	Bedeutung	Zugriff
DELAY	Zahl	Zeit in 1/10 Sekunden bis zum Ablauf des Timers (Verzögerungszeit)	Nur Schreiben
RUN	Boolean	TRUE wenn der Timer läuft. Kann vom SPS-Programm gesetzt werden, um einen Timer zu starten oder zu stoppen.	Lesen/Schreiben
TICK	Zahl	Interner Zeitstempel	Nur Lesen

n Tabelle 10 –Elemente der Unterobjekte TIM1 bis TIM4 (TASKx.TIMx)

Beispiele

```
TASK4.TIM1.DELAY = 10                 // 1 Sekunde (10 x 1/10) Verzögerung
TASK4.TIM1.RUN = TRUE                 // Timer starten
OUT4 = TRUE                           // Ausgang 4 einschalten
WHILE TASK4.TIM1.RUN                  // Warte bis Timer abgelaufen
ENDWHILE
OUT4 = FALSE                          // Ausgang 4 wieder abschalten

// Innerhalb der aktuellen Task kann die Tasknummer weggelassen werden!
TIM1.RUN = TRUE                       // Timer 1 der aktuellen Task starten
TASK6.TIM1.RUN = TRUE                 // Timer 1 einer anderen Task starten
```


4.3 PRO: Das Profibus Objekt

Das PRO Objekt erlaubt Ihnen den Zugriff und die Konfiguration auf bis zu zwei angeschlossene eMC200PROFI Profibus DP-Slaves. Das PRO Objekt existiert in Autrex in zweifacher Ausführung: PRO1 für das erste angeschlossene Profibus-Modul, PRO2 für das zweite.

Element	Datentyp	Bedeutung	Zugriff
ACTIVE	Boolean	TRUE wenn eine gültige Profibus-Adresse an den DP-Slave übermittelt wurde.	Nur lesen
ADDRESS	Zahl	Profibus-Adresse des DP-Slaves. Der DP-Slave wird erst am Profibus angemeldet, nachdem das SPS-Programm die Profibus-Adresse gesetzt hat.	Lesen/Schreiben
DETECT	Boolean	TRUE sobald der Profibus DP-Slave ACTIVE meldet und ein angeschlossener Profibus-Master erkannt wurde.	Nur Lesen
READY	Boolean	TRUE sobald der Profibus DP-Slave DETECT meldet und der Slave-Controller gültige Parameterdaten über den Profibus erhalten hat (wird vom Profibus-Master durch Konfiguration aus der GSD-Datei erledigt)	Nur lesen
RUN	Boolean	TRUE sobald der Profibus DP-Slave READY meldet und der Datenaustausch über den Profibus begonnen hat.	Nur lesen

n Tabelle 11 –Elemente des PRO Profibus Objekts

Beispiele

```
PRO1.ADDRESS = 4                        // Profibus-Adresse auf 4 setzen
TIM1.DELAY = 20                         // Überwachungstimer: 2 Sekunden
TIM1.RUN = TRUE                         // Timer starten
WHILE TIM1.RUN & !PRO1.DETECT           // Warten bis Profibus erkannt wurde oder
ENDWHILE                                // Überwachungszeit abgelaufen

IF !PRO1.DETECT                         // Wenn kein Profibus erkannt
     [Meldung an Benutzer]              // Fehlerbehandlung
ENDIF
```

4.4 MODEM: Das Modemobjekt

Autrex unterstützt die direkte Konfiguration und den Zugriff auf ein angeschlossenes Fernwartungsmodem. Die entsprechenden Funktionen sind im Objekt MODEM gekapselt:

Element	Datentyp	Bedeutung	Zugriff
SMS	Objekt	Zugriff auf das Unterobjekt SMS zum Versand von SMS-Nachrichten über das eMC200MODEM.	Lesen/Schreiben

n Tabelle 12 –Elemente des MODEM Objekts

n Unterelement SMS (MODEM.SMS)

Element	Datentyp	Bedeutung	Zugriff
CALL	Boolean	Wird dieses Element vom SPS-Programm auf TRUE gesetzt, sendet das angeschlossene eMC200MODEM Modul die SMS an die parametrierte Rufnummer.	Nur Schreiben
DIAL	Text	Rufnummer zum Versand der SMS.	Nur Schreiben
TEXT	Text	Zu sendender Text in der SMS.	Nur Schreiben

n Tabelle 13 –Elemente des Unterelements SMS (MODEM.SMS)

Beispiel

```
MODEM.SMS.DIAL = “+4917212345678“      // Anzuwählende Mobilrufnummer
SWITCH (Störung)                       // Abhängig davon welche Störung
CASE 1: MODEM.SMS.TEXT = "Totalausfall" // Meldungstext für Störung 1
CASE 2: MODEM.SMS.TEXT = "Schmiermittel leer"
CASE 3: MODEM.SMS.TEXT = "Notaus während Betrieb"
DEFAULT: MODEM.SMS.TEXT = "Schwere Störung"
ENDSWITCH
MODEM.CALL = TRUE                      // SMS absetzen
```


4.5 A: Das Achsenobjekt

Eines der wichtigsten Objekt in Autrex ist das Achsenobjekt A, dass in Autrex achtfach vorhanden ist. Die Achse 1 wird über A1 angesprochen, Achse 2 über A2 usw.

Über das Achsenobjekt werden nicht Positionierungen gestartet, sondern auch alle Statusinformationen der angeschlossenen Achsen abgefragt. Auch bei Programmierung der Positionierungen in DIN-Sprache werden alle Befehle in Zugriffe auf das jeweilige Achsenobjekt umgesetzt.

Element	Datentyp	Bedeutung	Zugriff
ACC	Zahl	Programmierte Beschleunigung (Rampe) in mm/sec²	Lesen/Schreiben
ACT	Zahl	Aktuelle Achsposition	Nur Lesen
CURR	Zahl	Wert Stromreduzierung, wird aktiviert mit G641	Lesen/Schreiben
CURR_ACT	Boolean	Stromreduzierung aktiv (G641/G64)	Lesen/Schreiben
DIST	Zahl	Restweg bis zum Erreichen der Zielposition (nur während einer Bewegung)	Nur Lesen
END	Zahl	Aktuelle Zielposition (während einer Bewegung) bzw. letzte Zielposition (letzte Zielposition). Indentisch mit dem Element TARGET.	Nur Lesen
ERROR	Boolean	Allgemeine Fehlerbedingung für die entsprechende Achse	Lesen/Schreiben
FE	Boolean	Wird vom SPS-Programm auf TRUE gesetzt, um die Schleppfehlerüberwachung der Achse zu aktivieren. Tritt ein Schleppfehler auf, setzt das System dieses Element wieder zurück.	Lesen/Schreiben
INPOS	Boolean	Steht auf TRUE, sobald die Achse in Position ist.	Nur Lesen
LSM	Boolean	Endschalter Minus (Limit Switch Minus) bedeckt	Nur Lesen
LSP	Boolean	Endschalter Plus (Limit Switch Plus) bedeckt	Nur Lesen
PRESENT	Boolean	TRUE wenn das entsprechende Achsmodul im System vorhanden ist.	Nur Lesen
READY	Boolean	TRUE wenn das Leistungsteil der Achse bereit meldet.	Nur Lesen
REF	Boolean	Referenzschalter bedeckt	Nur Lesen
RUN	Boolean	TRUE während die Achse läuft, FALSE wenn die Achse in Position ist. Wird dieser Wert vom SPS-Programm aus auf TRUE gesetzt, wird die Achse gestartet.	Lesen/Schreiben
START	Zahl	Letzte Startposition der Achse	Nur Lesen
SW_LSM	Zahl	Position Software Endschalter Minus (Software Limit Switch Minus)	Lesen/Schreiben
SW_LSP	Zahl	Position Software Endschalter Plus (Software Limit Switch Plus)	Lesen/Schreiben
TARGET	Zahl	Aktuelle Zielposition (während einer Bewegung) bzw. letzte Zielposition (letzte Zielposition). Identisch mit dem Element END.	Lesen/Schreiben
VEL	Zahl	Programmierte Zielgeschwindigkeit (Vorschub-Geschwindigkeit,. aktiviert mit G1) in mm/sec	Lesen/Schreiben

n Tabelle 14 –Elemente des Achsen Objekts A

Vereinfachtes Ansprechen des Objekts

Die Standardeigenschaft des Objekts ist die Zielposition (Ax.TARGET). Diese Eigenschaft kann direkt über den Objektnamen angesprochen werden. Die Zeile

```
A1.TARGET = 500
```

entspricht also der Zeile

```
A1 = 500
```

Beispiele

```
A1 = R100                          // Zielposition der Achse 1 auf den Inhalt der
                                   // Variable 100 setzen
A1.RUN = TRUE                      // Achse 1 starten
WHILE !A1.INPOS & !A1.ERROR        // Solange die Achse läuft und kein Fehler
ENDWHILE                           // auftritt

A1.VEL = R101                      // Sollgeschwindigkeit setzen
G1 G91 A1 = R102                   // Bewegung mit programmierter Geschwindigkeit
                                   // starten (Zielposition in Variable 102)
WHILE !A1.INPOS & !A1.ERROR        // Solange die Achse läuft und kein Fehler
     IF (A1.DIST < 100)            // Wenn Restweg kleiner als 100
          OUT3 = TRUE              // Ausgang 3 einschalten
     ENDIF
ENDWHILE

A1.ACC = R200                      // Beschleunigung (Rampe) setzen
G0 PTP LIN A1 = R333 A2 = R334     // Linear interpolierte Bewegung der Achsen 1
                                   // und 2 im Eilgang, warten auf Ende der Bewegung

A1.SW_LSP = A1.ACT                 // Aktuelle Position ist jetzt Endschalter Plus

// Mehrere Positionen aus einem Variablenarray anfahren
#DEFINE Index R                    // Variable Index deklarieren
FOR Index = 400 TO 419             // 20 Position aus Variablenarray
     A1 = R[Index]   // Zielposition Achse 1
     A2 = R[Index + 20]            // Zielposition Achse 2

     IF A1 > A1.SW_LSP-100         // Wenn weniger als 100 Weg
          G1                       // programmierte Geschwindigkeit
     ELSE                          // ansonsten
          G0                       // Eilgang
     ENDIF
     A1.RUN = TRUE                 // Achsen starten
     A2.RUN = TRUE

     REPEAT                        // Warten auf Achsen in Position
     UNTIL A1.INPOS & A2.INPOS & !A1.ERROR & !A2.ERROR
NEXT

// Gleiche Funktion in DIN-Sprache
#DEFINE Index R                    // Variable Index deklarieren
FOR Index = 400 TO 419             // 20 Position aus Variablenarray
     IF R[Index] > A1.SW_LSP-100   // Wenn weniger als 100 Weg
          G1                       // programmierte Geschwindigkeit
     ELSE                          // ansonsten
          G0                       // Eilgang
     ENDIF
     PTP A1=R[Index] A2=R[Index+20]
NEXT
```


4.6 OUT: Das Objekt für digitale Ausgänge

Eines der einfachsten Objekte in Autrex ist das Ausgangsobjekt OUT. Es existiert 512mal im System, wobei jedes Objekt einen digitalen Ausgang repräsentiert.

Element	Datentyp	Bedeutung	Zugriff
STATE	Boolean	Status des Ausgangs	Lesen/Schreiben

n Tabelle 15 –Elemente des Ausgangsobjekts OUT

Vereinfachtes Ansprechen des Objekts

Da dieses Objekt nur eine Eigenschaft (STATE) kennt, kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
OUT5.STATE = TRUE
```

entspricht also der Zeile

```
OUT5 = TRUE
```

Beispiel

```
OUT3 = TRUE                          // Ausgang 3 einschalten
OUT3 = !OUT3                         // Ausgang 3 umschalten
IF (OUT3)                            // Wenn Ausgang 3 eingeschaltet
     OUT4 = FALSE                    // Ausgang 4 ausschalten
ENDIF
```

n Bereichsaufteilung

Die verfügbaren Eingänge sind in mehrere Bereiche untergliedert, über die jeweils spezielle Ausgänge angesprochen werden können:

Von Ausgang	Bis Ausgang	Addressiert
1	128	Ausgänge im Schaltschrank (eMC200A16 und eMC200A8E8)
129	192	Virtuelle Ausgänge des ersten Profibus DP-Slaves
193	280	Reserviert
281	432	Ausgänge auf angeschlossenen c:Box Modulen (Modul 1-19)
433	496	Virtuelle Ausgänge des zweiten Profibus DP-Slaves
497	512	Reserviert

n Tabelle 16 –Bereichsaufteilung der Ausgänge

4.7 IN: Das Objekt für digitale Eingänge

Parallel zum Ausgangsobjekt OUT gibt es natürlich das Eingangsobjekt IN. Es existiert 512mal im System, wobei jedes Objekt einen digitalen Eingang repräsentiert.

Element	Datentyp	Bedeutung	Zugriff
STATE	Boolean	Status des Eingangs	Lesen/Schreiben
SIM	Boolean	Eingangssimulation ein/aus. Wenn auf TRUE gesetzt, kann der Status des Eingangs über das SPS-Programm beschrieben werden. Der physikalische Status des Eingangs wird dann nicht mehr ausgewertet.	Lesen/Schreiben

n Tabelle 17 –Elemente des Eingangsobjekts IN

Vereinfachtes Ansprechen des Objekts

Die Standardeigenschaft des Eingangsobjekts ist der aktueller Zustand. Deshalb kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
IF IN5.STATE
```

entspricht also der Zeile

```
IF IN5
```

Beispiele

```
IF (!IN17)                          // Wenn Eingang 17 nicht beschaltet
     IN17.SIM = TRUE                // Eingang 17 simulieren
     IN17 = TRUE                    // Eingang 17 simuliert einschalten
ENDIF

IF (IN10 & !IN11)                   // Wenn Eingang 10 und nicht Eingang 11
     OUT3 = TRUE                          // Ausgang 3 einschalten
ENDIF
```

n Bereichsaufteilung

Die verfügbaren Eingänge sind in mehrere Bereiche untergliedert, über die jeweils spezielle Ausgänge angesprochen werden können:

Von Ausgang	Bis Ausgang	Addressiert
1	128	Eingänge im Schaltschrank (eMC200E16 und eMC200A8E8)
129	192	Virtuelle Eingänge des ersten Profibus DP-Slaves
193	200	Reserviert
201	264	Achscontroller-Eingänge (ES+, ES-, REF, RDY) für 16 Achsen
265	280	Reserviert
281	432	Eingänge auf angeschlossenen c:Box Modulen (Modul 1-19)
433	496	Virtuelle Eingänge des zweiten Profibus DP-Slaves
497	512	Reserviert

n Tabelle 18 –Bereichsaufteilung der Eingänge

4.8 CBOX: Das Objekt für c:Box Module

Um den Status angeschlossener c:Box Module abzufragen, existiert in Autrex Systemen das Objekt CBOX in 19facher Ausführung.

Element	Datentyp	Bedeutung	Zugriff
ERROR	Boolean	TRUE wenn auf den Ausgängen dieses c:Box Moduls ein Kurzschluss aufgetreten ist.	Nur Lesen
PRESENT	Boolean	TRUE wenn dieses c:Box Modul am System vorhanden ist.	Nur Lesen
VDC_IN	Boolean	TRUE wenn 24 VDC Versorgungsspannung für die Eingänge dieses Moduls anliegen.	Nur Lesen
VDC_OUT	Boolean	TRUE wenn 24 VDC Versorgungsspannung für die Ausgänge dieses Moduls anliegen.	Nur Lesen

n Tabelle 19 –Elemente des Objekts CBOX

Beispiel

```
IF (!CBOX3.PRESENT)                    // Wenn die dritte c:Box nicht vorhanden ist
     [Fehler anzeigen]                 // Fehlerbehandlung
ELSE                                   // Ansonsten
     IF !CBOX3.VDC_IN | !CBOX3.VDC_OUT // Wenn keine Versorgungsspannung anliegt
          [Fehler anzeigen]            // Fehlerbehandlung
     ENDIF
ENDIF
```

n Verwenden von c:Box Eingängen als Achseingänge

Sie können mit Autrex digitale Eingänge so parametrieren, dass die angeschlossenen Achscontroller diese Eingänge als Achseingänge (Endschalter Plus, Endschalter Minus, Referenzschalter) betrachten. Hierzu müssen Sie lediglich in Achsparametern den jeweilige Nummer des c:Box Eingangs eintragen:

Achsparameter	Bedeutung
49	c:Box Eingang, der als Endschalter Plus verwendet werden soll (0 = Eingang auf dem Achscontroller verwenden).
50	c:Box Eingang, der als Endschalter Minus verwendet werden soll (0 = Eingang auf dem Achscontroller verwenden).
51	c:Box Eingang, der als Referenzschalter verwendet werden soll (0 = Eingang auf dem Achscontroller verwenden).

n Tabelle 20 –Verwenden von c:Box Eingängen als Achseingänge

4.9 AOUT: Das Objekt für analoge Ausgänge

Äquivalent zu den digitalen Ausgängen unterstützt Autrex bis zu 16 analoge Ausgänge. Hierzu sind im System die Objekte AOUT1 bis AOUT16 vorhanden.

Element	Datentyp	Bedeutung	Zugriff
Value	Zahl	Analoger Ausgangswert	Lesen/Schreiben

n Tabelle 21 –Elemente des analogen Ausgangsobjekts AOUT

Vereinfachtes Ansprechen des Objekts

Da dieses Objekt nur eine Eigenschaft (VALUE) kennt, kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
AOUT5.VALUE = 100
```

entspricht also der Zeile

```
AOUT5 = 100
```

Beispiel

```
AOUT3 = R20 + 100                    // Wert des dritten Analogausgangs auf den
                                     // Inhalt der Variable 20 + 100 setzen
```

n Wertebereich Analogausgänge

Die Analogausgänge des eMC200 Systems werden in 12 Bit aufgelöst. Dies bedeutet, dass vom Minimalwert bis zum Maximalwert ingesamt 4096 Abstufungen zur Verfügung stehen. Je nach hardwareseitig eingestellter Konfiguration entspricht damit ein Wert von 0 entweder –10 VDC oder 0 VDC, ein Wert von 4095 entspricht +10 VDC:

4.10 AIN: Das Objekt für analoge Eingänge

Für die bis zu 16 analogen Eingänge stellt Autrex jeweils ein Systemobjekt zur Verfügung: AIN1 bis AIN16.

Element	Datentyp	Bedeutung	Zugriff
Value	Zahl	Analoger Eingangswert	Nur Lesen

n Tabelle 22 –Elemente des analogen Eingangsobjekts AIN

Vereinfachtes Ansprechen des Objekts

Da dieses Objekt nur eine Eigenschaft (VALUE) kennt, kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
R100 = AIN5.VALUE
```

entspricht also der Zeile

```
R100 = AIN5
```

Beispiel

```
R100 = AIN3                          // Wert des dritten Analogeingangs in der
                                     // Variable 100 abspeichern.
```

n Wertebereich Analogeingänge

Die Analogeingänge des eMC200 Systems werden in 12 Bit aufgelöst. Dies bedeutet, dass vom Minimalwert bis zum Maximalwert ingesamt 4096 Abstufungen zur Verfügung stehen. Ein Wert von 0 entspricht damit immer der niedrigsten hardwareseitig eingestellten Eingangsspannung, ein Wert von 4095 der höchten eingestellten Eingangsspannung.

4.11 DSP: Das Displayobjekt

Als Unterstützung für bis zu zwei angeschlossene Displays/Tastaturen bietet Autrex das Display-Objekt DSP in doppelter Ausführung (DSP1/DSP2). Über dieses Objekt wird sowohl die Display-Anzeige für Text- und Grafikausgabe als auch die Tastatur für Benutzereingaben angesprochen.

Element	Datentyp	Bedeutung	Zugriff
Beep	Zahl	Mit dem Element BEEP kann an am Display vorhandener Lautsprecher angesprochen werden. Ein Wert von 0 schaltet den Lautsprecher aus, ein Wert von –1 schaltet den Lautsprecher dauerhaft ein. Weitere gültige Werte entnehmen Sie bitte der Tabelle weiter unten.	Nur Schreiben
Bitmap	Zahl	Nummer eine darzustellenden, im Display abgespeicherten Bitmap (falls vom jeweiligen Display unterstützt). Wird vom SPS-Programm gesetzt und führt zu einer sofortigen Ausgabe der gewählten Bitmap.	Nur Schreiben
Clear	Boolean	Wird dieser Wert vom SPS-Programm auf TRUE gesetzt, löscht das System den Bildschirm des Displays.	Nur Schreiben
Data	Objekt	Mit diesem Unterobjekt können Zahlenwerte formatiert in den Text eingebunden werden. Eine detaillierte Beschreibung fnden Sie weiter unten.	Nur Schreiben
Diag	Boolean	Diagnoseschalter: deaktiviert die Displayausgabe und die Tastatureingabe für alle Tasks ausser der Task 16.	Lesen/Schreiben
Input	Boolean	Wird dieser Wert vom SPS-Programm auf TRUE gesetzt, erwartet das Display eine nummerische Eingabe über die Tastatur. Der Ursprungswert der Eingabe ist vom SPS-Prgramm in das Element VALUE zu schreiben. Nach Abschluss der Eingabe steht der geänderte Wert wieder im Element VALUE zur Verfügung, und INPUT wird automatisch auf FALSE gesetzt.	Lesen/Schreiben
Key	Objekt	Bietet Zugriff auf den Status aller Tasten am Display. Hierzu gehören neben dem Status "Gedrückt/Nicht gedrückt" auch etwaige LEDs auf der jeweiligen Taste. Eine detaillierte Beschreibung fnden Sie weiter unten.	Lesen/Schreiben
Line	Objekt	Bietet Zugriff auf alle Textzeilen des Displays. Eine detaillierte Beschreibung fnden Sie weiter unten.	Nur Schreiben
Present	Boolean	TRUE wenn das Display vorhanden ist	Nur Lesen
Sign	Boolean	Wenn TRUE, wird bei Zahlenausgaben auch ein positives Vorzeichen (+) mit dargestellt. Wenn FALSE, wird nur das negative Vorzeichen (-) dargestellt.	Lesen/Schreiben
Task	Zahl	Ordnet die tatsächliche Displayausgabe dem Ausgabepuffer einer bestimmten Task zu.	Lesen/Schreiben
Value	Zahl	Vor Auslösen der Eingabefunktion: Ursprungswert für die Eingabe. Nach erfolgter Eingabe: geänderter Wert.	Lesen/Schreiben

n Tabelle 23 –Elemente des Displayobjekts DSP

n Unterobjekt LINE (DSPx.LINEx)

Über das Unterobjekt LINE, dass für jedes Display-Objekt 20mal bereitgestellt wird, können die einzelnen Textzeilen des Displays beschrieben werden.

Element	Datentyp	Bedeutung	Zugriff
Text	Text	Auszugebender Text	Nur Schreiben
Double	Boolean	Text in doppelter Schriftgröße ausgeben. Muss gesetzt werden, bevor das Element TEXT geschrieben wird.	Nur Schreiben
Unterline	Boolean	Text unterstrichen ausgeben. Muss gesetzt werden, bevor das Element TEXT geschrieben wird.	Nur Schreiben
Invert	Boolen	Text invertiert ausgeben. Muss gesetzt werden, bevor das Element TEXT geschrieben wird.	Nur Schreiben

n Tabelle 24 –Elemente des Unterobjekts LINE (DSPx.LINEx)

Vereinfachtes Ansprechen des Objekts

Die Standardeigenschaft des Unterobjekts LINE ist das Element Text. Deshalb kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
DSP1.LINE4.TEXT = "Hallo"
```

entspricht also der Zeile

```
DSP1.LINE4 = "Hallo"
```

Beispiele

```
DSP1.CLEAR = TRUE                          // Display 1 löschen
DSP1.LINE1 = "SuperPortal 3000"            // Erste Textzeile beschreiben
DSP1.LINE2 = "Copyright (c) by me"         // Zweite Textzeile beschreiben
DSP1.LINE3 = "Das Super Protal!"           // Dritte Textzeile beschreiben
DSP1.LINE5.DOUBLE = TRUE                   // Zeile 5 in doppelte Grösse
DSP1.LINE5 = "Press [N+]"                  // Fünfte Textzeile doppelt gross
                                           // ausgeben
```

n Unterobjekt DATA (DSPx.DATAx)

Um formatierte Textausgaben mit nummerischen Werten zu mischen, steht Ihnen in Autrex eine komfortable und mächtige Funktion zur Verfügung: die Programmierung von Darstellungsmasken. Zum Füllen der programmierten Ausgabemaske mit Daten stellt Autrex das Unterobjekt DATA in 4facher Ausführung bereit.

Element	Datentyp	Bedeutung	Zugriff
Value	Zahl	Auszugebende Zahl	Nur Schreiben

n Tabelle 25 –Elemente des Unterobjekts DATA (DSPx.DATAx)

Vereinfachtes Ansprechen des Objekts

Da dieses Objekt nur eine Eigenschaft (VALUE) kennt, kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
DSP1.DATA2.VALUE = R100
```

entspricht also der Zeile

```
DSP1.DATA2 = R100
```


Anwendung des DATA Unterobjekts bei der Ausgabe

Um die in das Unterobjekt DATA eingetragenen Werte mit Text zu kombinieren, wird in die Textzeile, in der die Zahlenwerte erscheinen sollen, eine Textmaske geschrieben. Diese Textmaske besteht aus normalen Textzeichen sowie aus dem Platzhalterzeichen $ (Dollar). Statt dem Dollarzeichen stellt das Display die in DATA eingetragenen Werte dar.

```
DSP1.DATA1 = A1.ACT                    // Aktuelle Istposition der Achse 1 soll
                                       // als Datenwert dargestellt werden
DSP1.LINE5 = "Achse 1: $$$$$.$"        // Ausgabe in Zeile 5 mit einer Kommastelle
                                       // und dem vorhergehenden Text "Achse 1:"

// Positionen der ersten vier Achsen in einer Schleife anzeigen
#DEFINE Zähler R                       // Zählervariable definieren
DSP1.SIGN = TRUE                       // Vorzeichen immer anzeigen
FOR Zähler = 1 TO 4                    // Die ersten vier Achsen darstellen
     DSP1.DATA1 = Zähler               // Erster Ausgabewert ist die Achsnummer
     DSP1.DATA2 = A[Zähler].Act        // Zweiter Wert ist die Position der Achse
     DSP1.LINE[Zähler + 2] = "Achse $: $$$$$$.$mm"
NEXT                                   // Ausgabe in die Zeilen 3-6
```

Das vorstehende Beispiel führt dann zu folgender Displayausgabe:

```
Achse 1:   -100.3mm
Achse 2: +90001.2mm
Achse 3:     +0.0mm
Achse 4:   -123.4mm
```

Weitere Erläuterung der Formatierungsmaske

Der Zahlenwert wird immer mit genau so viel Stellen angezeigt wie im Formatierungstext als "$"-Symbol angegeben wurden. Ist der Zahlenwert zu gross und kann nicht dargestellt wird, gibt das Display stattdessen Fragezeichen aus – ein Wert von 100 mit einer Formatierungsmaske "$$" stellt also zwei Fragezeichen dar.

Die gewünschte Anzahl der Nachkommastellen wird – unabhängig von der mit #DECIMAL gewählten Auflösung – in der Maske mit angegeben. Dies führt zu folgenden Ergebnissen in Abhängigkeit von der Formatierungsmaske:

Zahlenwert	Formatmaske	Ausgabe
123.4	$$	??
123.4	$$$	123
123.4	$$$.$	123.4
123.4	$$$.$$	123.40

n Tabelle 26 – Auswirkung der Formatierungsmaske auf die Nachkommastellen

Vorzeichen in der Formatierungsmaske

Bitte beachten Sie, dass das Vorzeichen ebenfalls ein Zeichen in der Formatmaske benötigt. Vorstehende Tabelle sieht mit negativem Vorzeichen wie folgt aus:

Zahlenwert	Formatmaske	Ausgabe
-123.4	$$$	???
-123.4	$$$.$	???.?
-123.4	$$$$.$	-123.4

n Tabelle 27 – Vorzeichen in der Formatierungsmaske

n Unterobjekt KEY (DSPx.KEYx)

Über das Unterobjekt KEY des Display-Objekts kann nicht nur der Status einer Taste abgefragt werden, sondern auch etwaige LEDs ein- bzw. ausgeschaltet werden. Zur Unterstützung diverser Frontplatten und Bedienteile stellt Autrex das KEY Objekt 64mal für jedes Display zur Verfügung. Verfügt die Frontplatte bzw. das Bedienteile über weniger Tasten, sind die nicht verwendeten KEY Objekte als reserviert zu betrachten.

Element	Datentyp	Bedeutung	Zugriff
State	Boolean	Aktueller Status dieser Taste, TRUE wenn gedrückt. Im Gegensatz zum Element TEST wird hierbei der Status der Taste nicht verändert.	Lesen/Schreiben
Test	Boolean	Aktueller Status dieser Taste, TRUE wenn gedrückt. Falls die Taste gedrückt ist, wird der Status intern zurückgesetzt, d.h. TEST hat eine Impulsfunktion.	Nur Lesen
Red	Boolean	Status der roten LED dieser Taste, TRUE = LED an.	Lesen/Schreiben
Green	Boolean	Status der grünen LED dieser Taste, TRUE = LED an.	Lesen/Schreiben

n Tabelle 28 –Elemente des Unterobjekts KEY (DSPx.KEYx)

Vereinfachtes Ansprechen des Objekts

Die Standardeigenschaft des Unterobjekts KEY ist das Element Test. Deshalb kann im Programm auf die Angabe der Eigenschaft verzichtet werden und das Objekt direkt angesprochen werden. Die Zeile

```
IF DSP1.KEY17.TEST
```

entspricht also der Zeile

```
IF DSP1.KEY17
```

Beispiele

```
DSP1.KEY1.RED = TRUE                    // Rote LED der ersten Taste anzünden
IF DSP1.KEY17                           // Wenn Taste 17 gedrückt
     DSP1.LINE1 = "Taste 17"            // Textausgabe
ENDIF

// Testprogramm: grüne LED der gedrückten Tasten einschalten
#DEFINE Taste R                         // Variable "Taste" definieren
WHILE !DSP1.KEY32                       // Abbrechen mit Taste 32 (N+)
     FOR Taste = 1 TO 31                // alle anderen 31 Tasten prüfen
          DSP1.KEY[Taste].GREEN = DSP1.KEY[Taste].State
                                        // Status auf grüne LED übertragen
     NEXT
ENDWHILE
```

n Vereinfachte Tastaturabfrage

Für einige Displays und Frontplatten sind vordefinierte Definitionen verfügbar, durch die alle Tasten nicht nur über die Tastennummer, sondern auch über einen sprechenden Tastennamen angesprochen werden können. Weitere Informationen finden Sie im Anhang B - Tastaturabfragen (Seite 74).

n Multitasking und die Displayausgabe

Während Autrex 16 unabhängig voneinander laufende Programme unterstützt, können "nur" zwei Displays angeschlossen werden. Dennoch ist gewährleistet, dass jede Task selbstständig auf das Display schreiben kann: jede Task verfügt über einen eigenen internen Display-Puffer, in den alle Textausgaben geschrieben werden. Dieser Puffer ist stets aktuell und wird vom System verwaltet.

Um den Inhalt dieses Puffers auch tatsächlich auf ein Display auszugeben, muss sich der entsprechende Task das Display reservieren. Dies geschieht durch Beschreiben des Elements TASK eines Displayobjekts:

```
DSP1.TASK = 3
```

schaltet die Ausgabe der Task 3 auf das erste Display,

```
DSP2.TASK = 10
```

schaltet die Ausgabe der Task 10 auf das zweite Display. Nach dem Umschalten des Displays auf eine andere Task überträgt das System automatisch den gesamten Inhalt des internen Display-Puffers der gewählten Task an das Display. Der tatsächliche Anzeige auf dem Display entspricht also unmittelbar nach der Umschaltung dem Inhalt des Display-Puffers der gewählten Task.

n Multitasking und Tastatureingaben

Tastatureingaben sind – unabhängig von der für das Display aktivierten Task – immer für alle Tasks gleichermaßen gültig. Sprich: alle Tastendrücke werden gleichzeitig an alle Tasks mitgeteilt.

Einzige Ausnahme ist der Diagnosebetrieb, der durch Setzen der Eigenschaft DIAG auf TRUE aktiviert wird: ist diese Option angewählt, erhält nur noch die Task 16 Informationen über gedrückte Tasten.

4.12 CNET: Das Objekt für Steuerungsnetze

Autrex unterstützt die direkte Vernetzung mehrere eMC200 Systeme über ein spezialisiertes eMC200SER Modul. Hierzu wird für jedes Steuerungssystem ein CNET Objekt zur Verfügung gestellt.

Element	Datentyp	Bedeutung	Zugriff
Address	Zahl	Adresse der Steuerung, mit der kommuniziert werden soll. Entspricht der eingestellten Steuerungsaddresse wie in 4.1 - SYS: das Basisobjekt (Seite 50) beschrieben.	Lesen/Schreiben
Read	Boolean	Wird vom SPS-Programm gesetzt, um den gewünschten Variablenwert von der entfernten Steuerung zu lesen. Sobald die Daten verfügbar sind, setzt das System dieses Element wieder auf FALSE.	Lesen/Schreiben
ReadError	Boolean	Wird vom System auf TRUE gesetzt, wenn beim Lesen eines Variablenwerts von einer entfernten Steuerung ein Fehler aufgetreten ist. Sollte vom SPS-Programm wieder gelöscht werden.	Lesen/Schreiben
Value	Zahl	Beim Lesen von einer entfernten Steuerung: Wert der gelesenen Variable. Beim Schreiben auf eine entfernte Steuerung: zu schreibender Wert.	Lesen/Schreiben
Variable	Zahl	Nummer der Variable, die in der entfernten Steuerung gelesen oder geschrieben werden soll.	Lesen/Schreiben
Write	Boolean	Wird vom SPS-Programm gesetzt, um den parametrierten Variablenwert in einer Variable der entfernten Steuerung zu schreiben. Sobald die Daten geschrieben wurden, setzt das System dieses Element wieder auf FALSE.	Lesen/Schreiben
WriteError	Boolean	Wird vom System auf TRUE gesetzt, wenn beim Schreiben eines Variablenwerts in eine entfernte Steuerung ein Fehler aufgetreten ist. Sollte vom SPS-Programm wieder zurückgesetzt werden.	Lesen/Schreiben

n Tabelle 29 –Elemente des CNET Objekts zur Steuerungsvernetzung

Beispiel

```
CNET.Address = 5                        // Mit Steuerung Nr. 5 kommunizieren
CNET.Variable = 100                     // Variable 100 auslesen
CNET.Read = TRUE                        // Auslesen durchführen
While CNET.Read & !CNET.ReadError       // Bis Wert gelesen oder Fehler aufgetreten
Endwhile

IF CNET.ReadError                       // Wenn ein Fehler aufgetreten
     ADR5Vorhanden = FALSE              // Bit setzen: Steuerung 5 nicht da
ELSE                                    // Ansonsten
     ADR5Status = CNET.Value            // Variablenwert 100 abspeichern
     CNET.Variable = 400                // Variable 400 in Steuerung 5 schreiben
     CNET.Value = R123                  // Auf den Wert der Variable 123 setzen
     CNET.Write                         // Schreibbefehl abschicken
ENDIF
```


4.13 COGNEX: das Objekt für Bilderkennung

Autrex unterstützt zusammen mit der eMC200 Familie Bilderkennungssysteme der Herstellers Cognex für den direkten Anschluss an das eMC200 Steuerungssystem. Dazu wird das Objekt COGNEX von Autrex bereitgestellt.

Element	Datentyp	Bedeutung	Zugriff
Degree	Zahl	Winkel Bilderkennung, 0 = Fehler Bilderkennung	Lesen/Schreiben
Part	Zahl	Werkstücknummer für Bilderkennung, 0 = Fehler Bilderkennung	Lesen/Schreiben
Read	Boolean	Wird vom SPS-Programm auf TRUE gesetzt, um die Bilderkennung durchzuführen. Im Anschluss daran werden die Elemente X, Y, DEGREE und PART vom System ausgefüllt und das Element READ wieder auf FALSE gesetzt.	Lesen/Schreiben
X	Zahl	X-Position Bilderkennung, 0 = Fehler Bilderkennung	Lesen/Schreiben
Y	Zahl	Y-Position Bilderkennung, 0 = Fehler Bilderkennung	Lesen/Schreiben

n Tabelle 30 –Elemente des COGNEX Objekts zur Bilderkennung

Beispiel

```
COGNEX.READ = TRUE                          // Bilderkennung durchführen
IF COGNEX.PART == 0                         // Fehler bei Bilderkennung?
     [Fehlerbehandlung]                     // Fehler behandeln
ELSE                                        // Ansonsten
     PTP G90 A1=COGNEX.X A2=COGNEX.Y        // Position aus Bilderkennung
                                            // anfahren
ENDIF
```


4.14 FLASH: Das Objekt für Flashsicherung

Autrex bietet Ihnen die Möglichkeit, Variablendaten und SPS-Programme auf der integrierten SiliconDisc der eMC200 Familie oder auf einer externen Datencassette des Typs eMC200DSD abzuspeichern. Hierzu stellt Autrex zwei FLASH Objekte zur Verfügung: FLASH1 für das integrierte Flash Memory (SiliconDisc), FLASH2 für die externe Datencassette (eMC200DSD).

Element	Datentyp	Bedeutung	Zugriff
PLC	Objekt	Unterobjekt zum Lesen und Speichern von SPS-Programmen im Flash-Speicher.	Lesen/Schreiben
Present	Boolean	Gibt bei FLASH2 an, ob eine externe Datencassette gesteckt ist. Immer TRUE bei FLASH1.	Nur Lesen
Var	Objekt	Unterobjekt zum Lesen und Speichern von Variablendaten.	Lesen/Schreiben
Sys	Objekt	Unterobjekt zum Lesen und Speichern eines Systemdatensatzes	Lesen/Schreiben

n Tabelle 31 –Elemente des FLASH Objekts zur Datensicherung

n Unterobjekt PLC (FLASHx.PLC)

Zum Abspeichern und Laden des SPS-Programms steht für jedes FLASH Objekt das Unterobjekt PLC zur Verfügung.

Element	Datentyp	Bedeutung	Zugriff
Read	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um das SPS-Programm aus dem Flash in das RAM zu übertragen. Es wird automatisch ein Steuerungs-Reset durchgeführt, das aus dem Flash geladene SPS-Programm wird gestartet.	Nur Schreiben
Save	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um das aktuelle SPS-Programm aus dem RAM in das Flash zu übertragen.	Nur Schreiben
Swap	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um das aktuelle SPS-Programm im mit dem SPS-Programm im Flash zu tauschen. Das neu aus dem Flash geladene SPS-Programm wird gestartet.	Nur Schreiben

n Tabelle 32 –Elemente des Unterobjekts PLC (FLASHx.PLC)

Beispiele

```
Flash1.PLC.Save = TRUE                  // Speichert das aktuelles SPS-Programm
                                        // im internen Flash der Steuerung
If Flash2.Present                       // Wenn eine externe Datencassette erkannt
     Flash2.PLC.Read = TRUE             // Lade das SPS-Programm aus der externen
Endif                                   // Datencassette (automatischer Reset!)

Flash1.PLC.Swap = TRUE                  // Vertausche das SPS-Programm im RAM mit
                                        // dem SPS-Programm im internen Flash der
                                        // Steuerung
```

n Unterobjekt VAR (FLASHx.VAR)

Zum Verwalten von Variablendaten stellt Autrex für jedes FLASH Objekt das Unterobjekt VAR zur Verfügung. Mit diesem Objekt können Variablendaten abgespeichert und/oder geladen werden.

Element	Datentyp	Bedeutung	Zugriff
Block	Zahl	In dieses Element wird vom SPS-Programm der zu schreibende bzw. zu lesende Variablenblock eingetragen.	Lesen/Schreiben
End	Zahl	Letzte zu lesende Variable.	Lesen/Schreiben
Memory	Zahl	Grösse der für Variablen verfügbaren Flash-Speichers in Byte.	Lesen/Schreiben
Read	Boolean	Wird vom SPS-Programm auf TRUE gesetzt die gewünschten Daten (Elemente START und END) aus dem Flash in den aktiven Variablenspeicher zu laden.	Nur Schreiben
Save	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um die parametrierten Variablen (Elemente START und SIZE) aus dem aktiven Variablenspeicher in das Flash zu übertragen.	Nur Schreiben
Size	Zahl	Gibt die Grösse eine Variablenblocks an. Beim Abspeichern von Variablen werden so viele Variablen gespeichert wie im Element SIZE eingetragen. Achtung! Dieser Wert darf während des Programmablaufs nur einmal verändert werden. Das System unterstützt keine variablen Blockgrössen auf einem Flash-Baustein. Weiterhin muss der Wert für SIZE durch 64 teilbar sein.	Lesen/Schreiben
Start	Zahl	Erste zu schreibende bzw. zu lesende Variable.	Lesen/Schreiben

n Tabelle 33 –Elemente des Unterobjekts VAR (FLASHx.VAR)

Beispiele

```
// Variablen ab Variable 500 abspeichern
FLASH1.VAR.SIZE = 128                  // Ein Variablenblock = 128 Variablen
                                       // Achtung! Nachträgliches Ändern von SIZE
                                       // führt zum Verlust der bisherigen Daten!
FLASH1.VAR.BLOCK = 5                   // Als Flash-Block 5 abspeichern
FLASH1.VAR.START = 500                 // Ab Variable 500 speichern (bis Var. 627)
FLASH1.VAR.SAVE = TRUE                 // Abspeichern durchführen

// Variablenblock in Variable 1020 bis 1110 zurückladen
FLASH1.VAR.BLOCK = 5                   // Flash-Block 5 auslesen
FLASH1.VAR.START = 1020                // Ab Variable 1020 speichern
FLASH1.VAR.END = 1110                  // Bis Variable 1110 lesen – hierdurch
                                       // werden nicht alle Variablen, sondern nur
                                       // die ersten 91 zurückgeladen
FLASH1.VAR.READ = TRUE                 // Variablen in den aktiven Speicher laden
```


n Unterobjekt SYS (FLASHx.SYS)

Mit Hilfe des Unterobjekts SYS kann ein Systeminformationsblock abgespeichert werden. Diese Block wird automatisch von der Steuerung ins RAM geladen, wenn eine Urlöschung durchgeführt wird (durch Drücken des Reset-Schalters bei Einschalten der Steuerung). Dieser Systemblock enthält die gespeicherten Achsparameter sowie die Variablen R0 bis R499.

Aus diesem Grund ist es empfehlenswert, bei der Vergabe der Variablennummern systemkritische Daten stets in die Variablen unter 500 abzuspeichern.

Element	Datentyp	Bedeutung	Zugriff
Read	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um das SPS-Programm aus dem Flash in das RAM zu übertragen. Es wird automatisch ein Steuerungs-Reset durchgeführt, das aus dem Flash geladene SPS-Programm wird gestartet.	Nur Schreiben
Save	Boolean	Wird vom SPS-Programm auf TRUE gesetzt um das aktuelle SPS-Programm aus dem RAM in das Flash zu übertragen.	Nur Schreiben

n Tabelle 34 –Elemente des Unterobjekts SYS (FLASHx.SYS)

Wichtiger Hinweis

Das Unterobjekt SYS ist nur für das erste FLASH Objekt verfügbar!

Beispiel

```
FLASH1.SYS.SAVE = TRUE                  // Systemdaten im Flash abspeichern
```

n Raum für Ihre Notizen

Anhang A Wahrheitstabelle

In diesem Anhang finden Sie eine Übersicht, wie sich logische Verknüpfungen – UND, ODER, NODER, NUND oder XODER – auf das Ergebnis auswirken.

Verknüpfung	Parameter 1	Parameter 2	Ergebnis
ODER	Aus	Aus	Aus
ODER	Ein	Aus	Ein
ODER	Aus	Ein	Ein
ODER	Ein	Ein	Ein
NODER (Nicht-Oder)	Aus	Aus	Ein
NODER (Nicht-Oder)	Aus	Ein	Aus
NODER (Nicht-Oder)	Ein	Aus	Ein
NODER (Nicht-Oder)	Ein	Ein	Ein
NUND (Nicht-Und)	Aus	Aus	Aus
NUND (Nicht-Und)	Ein	Aus	Ein
NUND (Nicht-Und)	Aus	Ein	Aus
NUND (Nicht-Und)	Ein	Ein	Aus
UND	Aus	Aus	Aus
UND	Ein	Aus	Aus
UND	Aus	Ein	Aus
UND	Ein	Ein	Ein
XODER (Exklusiv-Oder)	Aus	Aus	Aus
XODER (Exklusiv-Oder)	Ein	Aus	Ein
XODER (Exklusiv-Oder)	Aus	Ein	Ein
XODER (Exklusiv-Oder)	Ein	Ein	Aus

n Tabelle 35 – Wahrheitstabelle

Anhang B Tastaturabfragen

Wie in Kapitel 4.11 - DSP: Das Displayobjekt (Seite 62) erläutert,. werden die Tasten in Autrex als Unterobjekte zum DSP Objekt angesprochen. Um die Tastaturabfrage jedoch etwas einfacher zu machen, können Sie dem jeweiligen Display auch über einen Display-Namen ansprechen und die Tasten über den Tastennamen ansprechen.

Hierzu sind folgende Frontplatten/Bedienteile in der Autrex-Datei DISPLAYS.AXD vordefiniert:

- MC200BED (Typ BED)
- HT / HT-II (Typ HT)

Auswahl der Tastenbelegung

Zur Auswahl des gewünschten Tastaturbelegung verwenden Sie die Compiler-Direktive #DISPLAY, beschrieben im Kapitel 3.3 - Compiler-Direktiven (Seite 35)

n Tastennamen für Displaytyp BED

Tastenname	Tastennummer	Beschriftung
Stop	1	STOP
Auto	9	AUTO
Ovr	17	%
Diag	25	DIAG
Off	2	OFF
On	3	ON
Mode	14	MODE
Sft	15	SFT
M	23	M
N_Minus	21	N-
N_Plus	29	N+
Del	28	DEL
Sign	20	+/-
Num0	12	0
Num1	11	1
Num2	19	2
Num3	17	3
Num4	10	4
Num5	18	5
Num6	26	6
Num7	9	7
Num8	17	8
Num9	25	9

n Tabelle 36 – Tastennamen für den Displaytyp BED

n Tastennamen für Displaytyp HT

Tastenname	Tastennummer	Beschriftung
Minus	1	- (nur HT)
F0	9	F0 (nur HT)
Plus	25	+ (nur HAT)
Auto	2	AUTO
Man	10	MAN
Diag	18	DIAG
Help	26	HELP
On	3	ON
Run	11	RUN
Stop	19	STOP
Off	27	OFF
F1	4	F1
F2	5	F2
F3	6	F3
F4	7	F4
Sft	8	Sft
Num0	15	0
Num1	14	1
Num2	22	2
Num3	30	3
Num4	13	4
Num5	21	5
Num6	29	6
Num7	12	7
Num8	20	8
Num9	28	9
Del	31	DEL
Sign	23	+/-
M	16	M
N_Minus	24	N-
N_Plus	32	N+

n Tabelle 37 – Tastennamen für den Displaytyp HT

Anhang C Benötigte PC-Software

Um in Autrex zu programmieren, benötigen Sie zwingend die VMC Workbench X2.

n Dokumentationen

Zu allen Programmen der VMC Workbench erhalten Sie eine Online-Dokumentation, wenn Sie im Hilfemenü "?" auf den Menüpunkt "Index" klicken, oder einfach die F1-Taste drücken.

n Installation der Software

Legen Sie die VMC Workbench CD-ROM in das CD-Laufwerk Ihres Computers ein. Das Installationsprogramm startet automatisch. Wählen Sie "VMC installieren", und folgen Sie den Anweisungen auf dem Bildschirm.

Anhang D Abbildungen und Tabellen

n Tabellen

n Abbildungen

Fehler! Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

n Raum für Ihre Notizen

