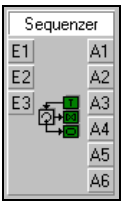
Teil II: HYDRA® Sequenzer

Revision

Versionsstand dieses Dokuments: 1.15

Inhaltsverzeichnis

Kapitel 1 HYDRA Sequenzer



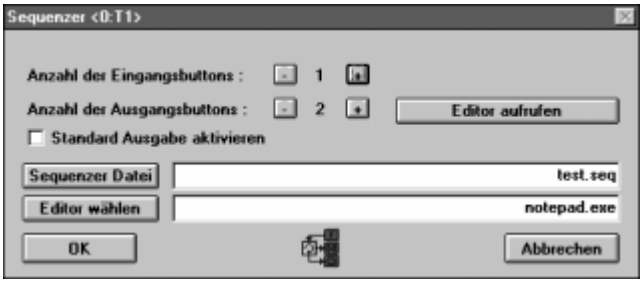
Der Funktionsblock HYDRA Sequenzer stellt eine C-ähnliche Programmiersprache zur Verfügung. Mit diesem Funktionsblock lassen sich sequentielle Abläufe sowie ereignisabhängige Steuerungsaufgaben schnell und einfach realisieren. Sie können auch kleinere Programmieraufgaben mit dem **HYDRA Sequenzer** Block implementieren, da keinerlei zusätzliche Programmierwerkzeuge notwendig sind. Zeitraubende Übersetzer- und Linkerläufe entfallen.

Die Syntax entspricht der Programmiersprache C. Das hat den Vorteil, daß eine spätere Einbindung eines **Sequenzer** Programms in einen HYDRA USER Block problemlos möglich ist.

Die **HYDRA Sequenzer** Programme werden nach dem Start des HYDRA-Systems vollständig auf syntaktische Richtigkeit überprüft und in eine Ablauftabelle übersetzt. Damit wird eine optimale Abarbeitungsgeschwindigkeit und Betriebssicherheit erzielt. Alle Meldungen über Syntax- oder Laufzeitfehler werden über das **HYDRA Control Programm** im Fenster **STD-IO** angezeigt.

Konfiguration

Bei der Auswahl eines HYDRA Sequenzer Funktionsblockes wird folgendes Eingabefenster geöffnet:



Zunächst stellen Sie die Anzahl der Ein- und Ausgänge mit Hilfe der Plus- und Minus-Schalter ein. Der **HYDRA Sequenzer** läßt sich auf maximal 16 Eingänge und 16 Ausgänge konfigurieren. Die Standardeinstellung ist jeweils 1 Eingangs- und 2 Ausgangskanäle.

Das **Sequenzer** Programm wird in einer getrennten Textdatei im ASCII-Format gespeichert, die z.B. mit dem Windows Editor **Notizblock** bearbeitet werden kann. Der **Sequenzer** ist in der Lage, die Textdateien beliebiger Textverarbeitungsprogramme einzubinden.

Die Dateiauswahl geschieht durch Klick auf den Schalter **Sequenzer Datei**. Daraufhin wird das Windows-Fenster **Datei öffnen** aufgerufen. Hier wählen Sie die Textdatei aus; der Name wird in der Dialogbox eingetragen. Er kann aber auch manuell eingegeben werden. Die Standard Dateierweiterung der **Sequenzer** Programme ist .seq, wie z.B. test.seq.

Zu Debug-Zwecken können im **Sequenzer** Programm Ausgaberoutinen, wie z.B. *printf(...)* oder *puts(...)*, verwendet werden. Diese leiten ihre Ausgabe in das Fenster **STD-IO** des **HYDRA Control Programms**, falls Sie die Option **Standard Ausgabe aktivieren** im Eingabefenster des **Sequenzers** angewählt haben. Mit dieser Option werden die der Fehleranalyse dienenden Textausgaben im **STD-IO** Fenster an- und abgeschaltet. Haben Sie **Standard Ausgabe aktivieren** nicht angekreuzt, also deaktiviert, so verhält sich das **Sequenzer** Programm bezüglich der Laufzeit genauso, als wären die entsprechenden Programmzeilen für die Textausgabe nicht vorhanden. Es besteht damit keine Notwendigkeit mehr, die jeweiligen *printf(..)* und *puts(...)* Funktionen nach Abschluß der Testphase auszudokumentieren.

Die Einstellungen werden mit der HYDRA-Schaltbilddatei zusammen gespeichert. Die **Sequenzer** Datei ist unabhängig von der Schaltbilddatei und kann auch getrennt verändert werden. Die Zuordnung der Eingangs- und Ausgangskanäle zu den **Sequenzer** Programmvariablen wird innerhalb des **Sequenzer** Programms festgelegt.

Syntaktischer Aufbau

Syntax:

Syntaktisch ähneln **Sequenzer** Programme einem C-Programm. Sie müssen jede Variable vereinbaren. Vereinbarungen können lokal oder auch global gemacht werden. Wie in der Programmiersprache C werden globale Variable vor der ersten Funktionsdefinition deklariert, lokale Variable werden in der Funktion selbst deklariert.

Beispiel:

```
//globale Variable
float global_x, global_y;
void meine_funktion(PAR)
{
   // lokale Variable
   float local_x, local_y;
   ...
}
```

Außerhalb der Funktion *meine_funktion* sind die Variablen *local_x, local_y* nicht bekannt.

Variablennamen:

Sie können in einem **Variablennamen** alle Buchstaben, Zahlen und das Zeichen '_' verwenden. Das erste Zeichen muß ein Buchstabe sein. Der **Variablenname** darf max. **24 Zeichen** lang sein. Ältere Versionen lassen nur eine Länge von 16 Zeichen zu. Die **Groß-/Kleinschreibung** wird im gesamten **Sequenzer** Programm unterschieden, d.h. die Zeichenfolge "abcdefg" ist **nicht** mit der Zeichenfolge "AbCdEfG" identisch.

Startfunktion:

Jedes **Sequenzer** Programm muß eine Startfunktion Namens *start* besitzen. Diese dient als Programmeintrittspunkt. Von dieser Funktion aus wird die Verarbeitung begonnen. Die Startfunktion kann an jeder beliebigen Stelle im Programm vereinbart werden, sie darf jedoch nicht von anderen Funktionen aufgerufen werden.

Kommentar:

Sie können an jeder beliebigen Stelle im Programmtext Kommentare einfügen. Diese werden mit den beiden Zeichen '**//**' angeführt und bleiben bis zum Zeilenende wirksam. Kommentare, die über mehrere Zeilen gehen, beginnen mit '/*' und schließen mit '*/' ab.

Beispiel:

```
//Kommentar über Eine Zeile

/* Dies ist ein Kommentar der über mehrere
Zeilen geht. */
```

Alle Vereinbarungen und Anweisungen müssen mit einem Semikolon abgeschlossen werden. Logisch zusammengehörige Programmteile werden mit geschweiften Klammern {} um-schlossen (z.B. *if {...} else {...}*).

Datentypen, Variablen und Konstanten

Als **Datentypen** sind folgende Standarddatentypen implementiert:

- float
- int
- HEADER.

Der Datentyp *float* benötigt 32-Bit und repräsentiert Zahlen in Gleitkommadarstellung. Der Wertebereich reicht von $3.4 * 10^{-38}$ bis $3.4 * 10^{+38}$.

int:

Der Datentyp *int* ist ebenfalls 32 Bit lang und repräsentiert die ganzen Zahlen. Er hat jedoch nur einen Wertebereich von 24 Bit. Dies resultiert aus der Tatsache, daß bei der Arithmetik mit *int*-Variablen intern mit gerundeten *float*-Variablen gerechnet wird. Die eindeutige Abbildung von *float*- auf *int*-Werte ist nur bis 24 Bit Genauigkeit möglich.

ganze Zahlen:

Für ganze Zahlen > 2^{24} werden zwei Zahlen zusammengesetzt. Der Wertebereich ist auf 10^6 festgelegt. Als Variable wird ein Datenfeld der Länge 2 verwendet. Die ganze Zahl ergibt sich somit aus:

$$\text{Zahl} = \text{data}[0] + \text{data}[1] * 10^6 .$$

Zur einfacheren Bearbeitung dieser Zahlen stellt der **HYDRA Sequenzer** spezielle Standardfunktionen zur Verfügung, beispielsweise add_to_long(), long_diff() und long_sum().

HEADER:

Die **Variablen** vom Typ *HEADER* dienen zur Aufnahme des HYDRA-Datenheaders. Im Datenheader ist die Information über die Einheit und die Skalierung der Daten gespeichert. Variablen vom Typ *HEADER* müssen bei allen Lese- und Schreibfunktionen mit übergeben werden. Sie dürfen nur global vereinbart werden. Um den Typ der Achsen in den *HEADER*-Variablen zu setzen, stehen folgende **Einheiten** zur Verfügung:

Einheiten:

AMPERE	KILOWATT	PROZENT
AMPSPEKTRUM	LITER	PIKOFARAD
BAR	LOGIK	POWERSPEKTRUM
DAVOLT	LONG	STATISTIK
DEZIBEL	LUFTMENGE	SEKUNDEN
DEHNUNG	METER	SQR
DREHZAHL	MEGAOHM	SQRT
DWORD	MILLIAMPERE	STUNDEN
G	MILLIBAR	UHRZEIT
GRADCELSIUS	MILLIGRAMM	USEREINHEITEN
GRAMM	MILLIMETER	USEREDIT1
HERTZ	MIKROFARAD	USEREDIT2
HISTOGRAMM	MIKROMETER	USEREDIT3
INT	MINUTEN	USEREDIT4
JOULE	MILLISEKUNDEN	VOLT
KELVIN	MILLIVOLT	WATT
KILOAMPERE	MPROSEK	WOLT
KILOGRAMM	NANOFARAD	WINKELGRADE
KILOHERTZ	NEWTON	ZENTIMETER
KMH	NEWTONMETER	
KILONEWTON	NIX	
KILOOHM	OHM	

Sie können mehrere Variablen pro Zeile vereinbaren, auch die Zuweisung von Initialwerten bei der Deklaration ist zulässig. Bei einer Deklaration ohne zuweisung wird die Variable mit dem Defaultwert 0 initialisiert. Bei lokalen Variablen in Funktionen, die öfter aufgerufen werden, wird die Defaultinitialisierung nur einmal durchgeführt (wie bei static in C). Die Werte werden als

Fließkommazahlen (*float*) interpretiert. Das Dezi-malzeichen ist ein Punkt, z.B.: *x = 3* ist gleichwertig mit *x = 3.0*.

Ein Variablenname vom Typ **float** oder **int** darf max. 24 Zeichen lang sein. Ein Variablenname vom Typ **HEADER** darf max. 11 Zeichen lang sein.

Beispiel:

```
// Deklarationen
float x, y, z = 5;
// Feld mit 200 float-Werten
float daten[200];
void start(PAR)
{
   ...
}
```

In obigem Beispiel sind die Deklarationen fett gedruckt.

Felder

sind nur für *float*-Variablen definierbar. Die Indizierung der Felder beginnt bei 0. Die ersten sowie die letzten Elemente eines Feldes werden demnach wie folgt angesprochen:

Beispiel:

```
void start(PAR)
{
   float M_werte[100];
   float x_erster;
   float x_letzter;

   x_erster  = M_werte[0];  /*das ist der
   erste Wert */
   x_letzter = M_werte[99]; /*das ist der
   letzte Wert */
   x_letzter = M_werte[100]; /*ergibt eine
   Fehlermeldung */
}
```

Unzulässige Feldzugriffe werden zur Laufzeit erkannt und im **HYDRA Control Programm** im Fenster **STD-IO** gemeldet.

Konstanten: Folgende Konstanten sind vordefiniert:

- PI
- TRUE
- FALSE

PI

ist eine vordefinierte Konstante und hat den Wert 3.14159265358979323846. Sie kann überall dort verwendet werden, wo auch Variablennamen zulässig sind.

Beispiel: `y = sin((2*PI/T)*t);`

TRUE, FALSE:

Für logische Abfragen sind die Konstanten **TRUE** und **FALSE** vordefiniert. Der Wert von **FALSE** entspricht dem *float*-Wert 0.0, **TRUE** dem *float*-Wert 1.0.

Beispiel: Der Ausdruck (a == b) hat den Wert TRUE, falls die Variablen a und b gleich sind, ansonsten den Wert FALSE.

Die Konstanten **TRUE** und **FALSE** sind von den Abfragen „logisch wahr“ und „logisch falsch“ in bezug auf einen Ausdruck zu unterscheiden:**Logisch wahr** ist ein Ausdruck dann, wenn der Betrag dieses Ausdrucks >= 1.0 ist. **Logisch falsch** ist ein Ausdruck dann, wenn der Betrag des Ausdrucks < 1.0 ist.

Initialisierung:

Globale Variablen und Felder (Deklaration außerhalb der Hauptroutine start(PAR)) sowie lokale Variablen und Felder (Deklaration in start(PAR)) werden beim start der Messung mit 0 initialisiert. Lokale Variablen von selbstdefinierten Funktionen werden ebenfalls beim Start einer Messung mit 0 initialisiert.

Nicht initialisiert werden:

- Lokale Felder selbstdefinierter Funktionen beim Start einer Messung
- Lokale Felder selbstdefinierter Funktionen beim erneuten Aufruf dieser Funktion.
- Lokale Variablen selbstdefinierter Funktionen beim erneuten Aufruf dieser Funktion.

Funktionen

Mit Hilfe von **Funktionen** können Sie Problemstellungen in einzelne Teilaufgaben zerlegen. Funktionen können verschachtelt aufgerufen werden. Die maximale Verschachtelungstiefe ist 20. **Rekursive Funktionsaufrufe** sind **nicht erlaubt.** Der Aufbau einer **Sequenzer** Funktion sieht wie folgt aus:

Beispiel:

```
void function ( PAR, int number, float p1, float
p2,
float data[], ...)
{
   ...
}
```

Selbstdefinierte **Sequenzer** Funktionen geben generell keinen Wert zurück! Aus diesem Grund ist das Schlüsselwort *void* notwendig, um die C-Kompatibilität zu gewährleisten.

Sie haben aber die Möglichkeit, bei **Sequenzer** Funktionen Variable über die Parameterliste zu übergeben. Die Parameterliste kann *float, int*, aber auch *float*-Feldvariable beinhalten. Letztere werden durch das Anhängen von '[]' an den Feldnamen gekennzeichnet. Variable vom Typ *HEADER* dürfen bei selbstdefinierten Funktionen nicht als Parameter übergeben werden.

Werden in der aufgerufenen Funktion übergebene Variable vom Typ *float* oder *int* verändert, so sind diese Änderungen in der aufrufenden Funktion nicht bekannt. Diese Variablen werden in der aufgerufenen Funktion lokal gehalten. Nur Änderungen in *float*-Datenfeldern sind der aufrufenden Funktion bekannt.

Soll kein Wert übergeben werden, muß die Parameterliste einer **Sequenzer** Funktion zumindest aus dem reservierten Parameter PAR bestehen. Dieser Parameter dient der Umsetzung des **Sequenzer** Programms in ein C-Programm für die HYDRA USER Blöcke.

Eine selbstdefinierte **Sequenzer** Funktion muß vor ihrem Aufruf im Programmtext definiert worden sein. Ausnahmen bilden Funktionen ohne Übergabeparameter. Diese rufen Sie über die *call*-Funktion ohne vorherige Definition auf.

Der Funktionsname kann bis zu 19 Zeichen lang sein und muß mit einem Buchstaben beginnen. Im Funktionsnamen dürfen alle Buchstaben (a,...,z, A,...,Z), Zahlen (0,..,9) und das Zeichen '_' vorkommen.

Beispiel:

```
/* Aufruf einer selbst definierten Funktion mit
Parametern */
/* Der erste Parameter beim Aufruf muß 'p'
sein */
/* Der Parameter 'p' wird automatisch durch
'PAR' in der */
/* Funktionsparameterliste definiert. */

void test_2(PAR, int x, float y, float z[])
{
   int size;
   int i;
   puts("Variablen in der Funktion fkt");
   printf("x = %g\n",x);
   printf("y = %g\n",y);
   array_size(z,size);
   printf("Arraysize %d\n", size);
   loop(i,0,10)
   {
      printf("%g ",z[i]);
      z[i] = 22;
   }
   puts("");
   x = 0;
   y = 0;
}
/* Programmstart */
void start(PAR)
{
   float x_var, y_var, data[100];
   test_2(p, x_var, y_var, data);
}
```

Operatoren

Als Operatoren sind alle Grundrechenarten sowie einige logische Operatoren implementiert. Nachfolgend werden die zur Verfügung stehenden Operatoren aufgelistet. Die Priorität der Operatoren nimmt mit den Einträgen zu, das heißt der erste Eintrag hat die niedrigste Priorität, der letzte die höchste.

=	Wertzuweisung,
||	logisches ODER,
&&	logisches UND,
|	logisches ODER bitweise (24 Bit),
#	exklusives ODER bitweise (24 Bit),
XOR	exklusives ODER bitweise (24 Bit),
&	logisches UND bitweise (24 Bit),
!=	Vergleich zweier Werte auf Ungleichheit,
==	Vergleich zweier Werte auf Gleichheit (24 Bit),
<=	Vergleich zweier Werte auf kleiner oder gleich,
>=	Vergleich zweier Werte auf größer oder gleich,
<	Vergleich zweier Werte auf kleiner,
>	Vergleich zweier Werte auf größer,
+	Addition zweier Werte,
-	Subtraktion zweier Werte,
*****	Multiplikation zweier Werte,
/	Division zweier Werte,
%	Modulo Division
^	Potenz
+=	Addition und Zuweisung
-=	Subtraktion und Zuweisung
***=**	Multiplikation und Zuweisung
/=	Division und Zuweisung
++	Inkrement Operator
--	Dekrement Operator

Beispiel:

```
/* Programmbeispiel */
float x;
float y;
float z;

void start(PAR)
{
   z = x + y;          /* Addition von x und
   y, Ergebnis in z */
   z = x * y;          /* Multiplikation von
   x und y, Ergebnis z */
   z = x / y;          /* Division von x
   durch y, Ergebnis in z */
   z = x – y;          /* Subtraktion y von
   x, Ergebnis in z */
   z = x < y;          /* Vergleich auf x
   kleiner y, ergibt z = 1, wenn x < y,
   sonst z = 0 */

   z = x > y;  /* Vergleich auf x größer y,
   ergibt z = 1, wenn x >  y, sonst z = 0 */

   if (x > y ) goto (Label); /* verzweigt
   auf Label, wenn x > y ist */

   z = (x==5);         /* Vergleich auf
   Gleichheit, wenn x = 5 dann ist z=1,
   sonst 0 */

   z = x & 1;          /* bitweise UND, wenn
   x ungerade dann ist z = 1, sonst 0 */

   z = x && 1;  /* logisches UND, wenn x
   ungleich 0, dann ist z = 1, sonst 0 */

   z = x | 2;          /* bitweise ODER, wenn
   x = 4 dann ist z = 6 */
   z = x || y;         /* logisches ODER,
   wenn x oder y ungleich 0, ist z = 1 */

   z = x # 3;  /* bitweise EXOR, wenn x = 7
   dann ist z = 4 */

   z += x;     /* entspricht z = z + x */
   z -= x;     /* entspricht z = z – x */
```

```
   z *= x;     /* entspricht z = z * x */
   z /= x;     /* entspricht z = z / x */

   z++;        /* Wert von z um 1 erhöhen */
   z--;        /* Wert von z um 1 verringern
   */

}
```

Die auf Bitebene arbeitenden logischen Operatoren (&, |, #, ==) arbeiten mit Werten im *float*-Format, jedoch nur mit ganzen Zahlen. Dies ist aus Gründen der Datenkonsistenz erforderlich, beschränkt jedoch die Rechengenauigkeit dieser Operatoren auf 24 Bit (0 - 16777215). Weiter gilt zu beachten, daß die Operatoren nur auf natürliche Zahlen anwendbar sind, d.h. es findet vor der Berechnung eine Rundung statt. Die Ergebnisse sind wiederum natürliche Zahlen im *float*-Format.

Kontrollstrukturen und Schleifen

Für die Steuerung des Programmablaufs stehen Ihnen eine Reihe von Kontrollstrukturen und Schleifen zur Verfügung.

if-else-Anweisung

Die **if-else**-Anweisung können Sie bei Entscheidungen verwenden. Der **else**-Teil ist optional. Die allgemeine Syntax lautet:

Syntax:

```
if (float Ausdruck)
  Anweisung1;
else
  Anweisung2;
```

Die *Anweisung1* wird nur ausgeführt, wenn der Wert von *Ausdruck* logisch wahr ist, d.h. der Betrag von *Ausdruck* >= 1 ist. Andernfalls wird *Anweisung2* ausgeführt.

***Beispiel*:**

```
if(temp_1 > 55)
   soll = 0;
else
   soll = 10;
```

Nach der **else-**Anweisung kann wiederum eine **if-**Abfrage beginnen, so daß verkettete **if-else-if** Konstruktionen entstehen.

Beispiel:

```
if(temp_1 > 55)
   soll = 0;
else if(temp_1 > 20)
   soll = 5;
else
   soll = 10;
```

Der *Ausdruck* der **if**-Abfrage ist vom Typ *float*.

while-Schleife

Syntax:

```
while(float Ausdruck)
   Anweisung;
```

Die *Anweisung* wird solange wiederholt ausgeführt, bis der *Ausdruck* logisch falsch ist. Hat der *Ausdruck* vor dem Eintritt in die Schleife den Wert 0 (FALSE), so wird die Anweisung erst gar nicht durchgeführt.

Beispiel:

```
float data[100];
int i;
i = 0;
while(i < 100)
{
   data[i] = 0;
   i = i + 1;
}
```

Alle Elemente des Feldes *data* werden in obigem Beispiel auf Null initialisiert. Die **while**-Schleife wird verlassen, wenn *i* den Wert 100 hat.

do-while-Schleife

Syntax:

```
do
       Anweisung;
while (float Ausdruck);
```

Zuerst wird die *Anweisung* ausgeführt, anschließend der *Ausdruck* überprüft, ob dieser logisch wahr oder logisch falsch ist. Die Schleife wird solange wiederholt, wie der *Ausdruck* logisch wahr ist.

Die **do-while**-Schleife überprüft ihr Abbruchkriterium (*Ausdruck* ist logisch falsch) im Gegensatz zur **while**-Schleife erst am Ende der Schleife. Sie wird daher mindestens einmal durchlaufen.

Beispiel:

```
float data[100];
int i;
i = 0;
do
```

```
{
   data[i] = 0;
   i = i + 1;
}
while(i < 100);
```

loop-Schleife

Syntax:

```
loop(   float VAR,
        float start,
        float end)
        Anweisung;
```

Die **loop**-Schleife verwendet als Kontrollvariable *VAR*, die auf den Startwert *start* initialisiert wird. Die Kontrollvariable wird bei jeder Ausführung der *Anweisung* **um eins** erhöht, bis sie den Wert *end* erreicht hat. Sind die Werte von *start* und *end* gleich, wird die *Anweisung* nicht mehr ausgeführt.

Beispiel:

```
float data[100];
int i;
loop(i, 0, 100)
        data[i] = 0;
```

Alle Elemente des Feldes *data*[] werden in obigem Beispiel auf Null initialisiert. Zu beachten ist, daß die Kontrollvariable *i* nur bis 100 laufen darf, da sonst die Feldgrenze von *data*[] überschritten wird.

for-Schleife

Syntax:

```
for(    float Ausdruck1,
        float Ausdruck2,
        float Ausdruck3)
        Anweisung;
```

Die *Anweisung* in der **for**-Schleife wird solange ausgeführt, bis *Ausdruck2* nicht mehr erfüllt ist. *Ausdruck1* ist die Initialisierungsfunktion, welche beim Eintritt in die Schleife durchgeführt wird. *Ausdruck3* ist die Folgefunktion, die nach jedem Aufruf der *Anweisung* bearbeitet wird.

Beispiel:

```
float data[100];
int i;
for(i = 0, i < 100, i = i + 1)
data[i] = 0;
```

Alle Elemente des Feldes *data* werden in obigem Beispiel auf Null initialisiert. Nach dem Verlassen der Schleife hat *i* den Wert 100, der Anweisungsteil der Schleife wird allerdings für den Wert *i=100* nicht mehr durchlaufen.

continue-Anweisung

Die **continue**-Anweisung dient dazu, die nächste Wiederholung der umgebenden Schleife unmittelbar zu beginnen. Die **continue**-Anweisung kann in allen **for-, loop-, while-** und **do-while**-Schleifen vorkommen.

Beispiel:

```
float data[100];
int i;
...
for(i = 0, i < 100, i = i + 1)
{
   if(data[i] < 0) continue;
   data[i] = -0.1;
}
```

In obigem Beispiel werden alle Elemente vom Feld *data* auf -0.1 initialisiert, die vor der Schleife einen Wert größer gleich Null hatten.

break-Anweisung

Mit der **break**-Anweisung können Sie die umgebende Schleife unmittelbar verlassen. Die **break**-Anweisung kann in allen **for-, loop-, while-** und **do-while**-Schleifen eingesetzt werden.

Beispiel:

```
float data[100];
int i;
...
for(i = 0, i < 100, i = i + 1)
{
   if(data[i] < 0) break;
   data[i] = 0.0;
}
```

In obigem Beispiel werden alle Elemente vom Feld *data* auf 0.0 initialisiert. Die Schleife wird verlassen, wenn ein Element des Feldes *data* einen Wert kleiner Null hat.

Sequenzer Standardfunktionen

Der **HYDRA Sequenzer** stellt Ihnen zahlreiche Standardfunktionen zur Verfügung. Sie ermöglichen eine einfache und schnelle Implementierung der **Sequenzer** Programme.

Die Standardfunktionen werden in die Gruppen Arithmetik, Ein-/Ausgabe, Zeitfunktionen, Datenverwaltung, Zugiff auf HEADER, STD-IO und Blockbefehle eingeteilt.

Diese Funktionen werden Ihnen in den folgenden Kapiteln in alphabetischer Reihenfolge vorgestellt.

Hinweissymbole:

Um die Übergabeparameter besser von den Ergebnisparametern unterscheiden zu können, verwenden wir in diesem Kapitel im Syntax-Kasten folgende Symbole:

- **->** Übergabeparameter
- **<-** Ergebnisparameter
- **<->** sowohl Übergabe- als auch Ergebnisparameter.

Arithmetik-Funktionen

Die Programme für den **HYDRA Sequenzer** können sich vieler mathematischer Standardfunktionen bedienen, die Sie auch verschachtelt aufrufen können.

abs

Syntax:

float **abs**(**<-** float x);

Diese mathematische Funktion gibt den Absolutwert der Variablen x zurück. Der Übergabewert x bleibt unverändert.

Beispiel:

```
y = abs(sin(x)); // y wird der Betrag von sin(x) zugewiesen
```

arccos, arcsin, arctan, arcosh, arsinh, artanh

Syntax:

float **arccos**(**<-** float x);

float **arcsin**(**<-** float x);

float **arctan**(**<-** float x);

float **arcosh**(**<-** float x);

float **arsinh**(**<-** float x);

float **artanh**(**<-** float x);

Diese Funktionen berechnen den Arkuscosinus, Arkussinus, Arkustangens, Areacosinus hyperbolikus, Areasinus hyperbolikus, Areatangens hyperbolikus von x im Bogenmaß [rad]. Der Übergabewert x bleibt unverändert.

Beispiel:

```
y = arcsin(x); /* Arkussinus von x
Der Wert von y ist im Bogenmaß skaliert. */
```

add_to_long

add_to_long() ist eine spezielle mathematische Funktion zur Berechnung von ganzen Zahlen, die größer als 2^{24} sind. Diese ganzen Zahlen werden aus zwei Zahlen zusammengesetzt (s. Kapitel 6.3).

Syntax:

add_to_long(**<->** float y_data_array[],
-> float x_data);

add_to_long() addiert zur „ganzen Zahl“ *y_data_array*[], die größer als 2^{24} sein kann, die ganze Zahl *x_data* ($<2^{24}$) hinzu. Das Ergebnis steht in *y_data_array*[].

Beispiel:

```
float y_data_array[2];
float x_data;
...
y_data_array[0] = 1587;
y_data_array[1] = 15;   /* 15*(10**6) */
x_data = 1;

add_to_long(y_data_array, x_data);
...
```

In obigem Beispiel wird zur Zahl 1587 + 15 * 10^6 die Zahl 1 addiert. Das Ergebnis lautet 1588 + 15 * 10^6.

Für *x_data* = 1000001 erhalten Sie das Ergebnis 1588 + 16 * 10^6.

bit

Syntax:

int **bit**(**<-** int x);

Die Funktion **bit()** gibt eine Zahl zurück, bei der das n-te Bit auf Eins und alle anderen Bits auf Null gesetzt sind. Der Übergabewert n bleibt unverändert (n = 0,...,23).

Diese Funktion können Sie zum Maskieren von bit-Variablen verwenden. Durch bitweise UND-Verknüpfung (&) können einzelne bits abgefragt werden.

Beispiel 1:

```
y = bit(n);
/* y wird ein Wert zugewiesen bei dem
das n-te Bit gesetzt ist */
```

Für n = 3 ergibt sich y = 0...0100_b.

Beispiel 2:

```
int n_bit = 0;
int x_binaer_zahl = 5; /* 5 entspricht
0...0101b */
int y;
...
y = bit(n_bit) & x_binaer_zahl;
printf(„y = %f\n“, y);
...
```

Im Beispiel wird überprüft, ob bei der Binärzahl x_binaer_zahl das 0-te Bit gesetzt ist. Das Ergebnis ist 1.

Für n_bit = 1 ist y = 0.

Für n_bit = 2 ist y = 4.

bitmask_to_bool

Syntax:

bitmask_to_bool(**->** int x_bitmask,
<- float y_bool_array[]);

bitmask_to_bool() überprüft, welche Bits der Variablen *x_bitmask* auf Eins gesetzt sind. Das Ergebnis wird dem Datenfeld *y_bool_array*[] zugewiesen. Ist das erste Bit gesetzt, wird der Wert von *y_bool_array*[0] auf TRUE gesetzt, ansonsten auf FALSE. Mit allen 24 Bits wird in der gleichen Weise verfahren. Das Ergebnis von Bit n (TRUE oder FALSE) der Variablen *x_bitmask* wird in *y_bool_array*[n] abgelegt (n = 0,...,23).

Beispiel:

```
int x_bitmask;
float y_bool_array[24];

...
x_bitmask = 5;
bitmask_to_bool(x_bitmask, y_bool_array);
...
```

x_bitmask = 5 entspricht $0...0101_b$. Als Ergebnis erhalten Sie:

Im Datenfeld *y_bool_array*[] haben die Elemente 0 und 2 den Wert 1.0. Die anderen 22 Elemente erhalten den Wert 0.0.

boolnot

Syntax:

float **boolnot**(**<-** float x);

Diese Standardfunktion gibt das negierte boolsche Ergebnis des logischen Ausdrucks x zurück. x kann auch eine Zahl sein. Ist der Wert von x < 1, so enthält das Ergebnis den boolschen Wert TRUE (1.0). Ist der Wert von x >= 1, wird der boolsche Wert FALSE (0.0) als Ergebnis zurückgegeben. Der Übergabewert x bleibt unverändert (s. Kapitel 6.3. “Logische Konstanten“).

Beispiel 1:

```
y = boolnot(x);
/* y wird der negierte boolsche Wert der
Variablen x zugewiesen */
```

Beispiel 2:

```
float x;
float z;
...
x = 5.0;
if (boolnot(x > 40))
   z = 3.0;
else
   z = 9.0;   ...
```

Im diesem Beispiel wird bei der if-Abfrage zuerst der Ausdruck x > 4.0 ausgewertet. Da x = 5.0 ist der Ausdruck x > 4.0 logisch wahr (TRUE = 1.0). Danach wird auf das Ergebnis die Funktion **boolnot()** angewendet. Man erhält als Ergebnis den Wert FALSE = 0.0. Somit wird z der Wert 9.0 zugewiesen.

bool_to_bitmask

Syntax:

bool_to_bitmask(**->** float x_bool_array[],
<- int y_bitmask);

bool_to_bitmask() setzt die Bits in *y_bitmask* auf Eins, deren korrespondierende Elemente im Datenfeld *x_bool_array*[] den Wert TRUE enthalten. Ist das erste Element auf TRUE gesetzt, wird Bit 0 von *y_bitmask* gesetzt, ansonsten nicht. Mit allen 24 Bits wird in der gleichen Weise verfahren.

Beispiel:

```
int y_bitmask;
float x_bool_array[24];

...
x_bool_array[0] = TRUE;
bool_to_bitmask(x_bool_array, y_bitmask);
...
```

In diesem Beispiel wird Bit 0 der Bitmaske gesetzt. Das Ergebnis lautet: $0...01_b$ (24 bit).

ceil

Syntax:

int **ceil(->** float x);

Diese Funktion gibt den auf einen Integerwert aufgerundeten Wert von *x* zurück. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
float x;
int y;

...
x = 2.01;
y = ceil(x);
printf(„y = %d\n“, y);
...
```

Das Ergebnis ist y = 3.

diff

Syntax:

diff(**<-** float d[],
-> float x[],
-> float y[],
-> int len);

Die mathematische Standardfunktion **diff()** subtrahiert die Elemente zweier Datenfelder *x*[] und *y*[] mit der Anzahl *len* Werten. Das Ergebnisdatenfeld ist *d*[]. Es gilt also
d[n] = *x*[n] - *y*[n]. Die Subtraktion wird für

n = 0, ... , len-1 durchgeführt. Die Länge der Datenfelder muß größer gleich *len* sein.

Beispiel:

```
void start(PAR)
{
   int len = 50;
   float sub[50];
   float x[50], y[50];

   ...
   diff(sub, x, y, len);

/* Im Datenfeld sub steht die Differenz von
x[] und y[] */
}
```

exp

Syntax:

float **exp**(**<-** float x);

Durch Aufruf der Funktion $\mathbf{e^x}$ erhalten Sie die Exponentialfunktion von *x*. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = exp(sin(x));
/* y wird die Exponentialfunktion von sin(x)
zugewiesen */
```

fkt_ramp

Syntax:

fkt_ramp(**<-** float yrray[],
-> float start,
-> float stop,
-> int start_index,
-> int data_count);

Diese Funktion generiert eine Rampe, die ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Die Rampe beginnt mit dem Wert von *start* und endet mit dem Wert von *stop.* Sie besteht aus der Anzahl *data_count* Daten. Die Mindestanzahl der Rampendaten beträgt zwei (*data_count* = 2). Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index* + *data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start;
float stop;
int start_index;
int data_count;
...
start = -0.7;
stop = 3.5;
start_index = 800;
data_count = 200;
fkt_ramp(yarray, start, stop, start_index,
data_count);
...
```

In obigem Beispiel wird eine 200 Werte umfassende Rampe, die bei -0,7 beginnt und bei 3,5 endet, berechnet. Die 200 Daten werden in das Datenfeld *yarray*[], ab dem Index 800, eingetragen.

fkt_rectangle

Syntax:

fkt_rectangle(**<-** float yarray[],
-> float start_periode,
-> float rect_frequency_hz,
-> float data_frequency_hz,
-> float amplitude,
-> float offset,
-> int start_index,
-> int data_count);

Diese Funktion generiert ein Rechtecksignal mit *data_count* Daten, das ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Das Rechteck hat eine Phase von *start_periode*. Eine ganze Periode entspricht einer Phase von 1. Als Phase muß eine Zahl zwischen 0 und 1 gewählt werden. Die Amplitude des Rechtecksignals wird in der Variablen *amplitude* angegeben, die Frequenz in Hertz enthält die Variable *rect_frequency_hz* und der Gleichanteil steht in der Variablen *offset*. Für *data_frequency_hz* muß der Kehrwert des Stützstellenabstands in Sekunden übergeben werden. Dies entspricht der Abtastfrequenz in Hertz, mit der das kontinuierliche Rechtecksignal abgetastet wird. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index* + *data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start_periode = 0.5;
float rect_frequency_hz = 100;
float data_frequency_hz = 5000;
float amplitude = 5.0;
float offset = -1.8;
int start_index = 800;
int data_count = 200;

...
fkt_rectangle( yarray,start_periode,
rect_frequency_hz,    data_frequency_hz,
amplitude, offset,
      start_index, data_count);
...
```

In diesem Beispiel wird ein 200 Werte umfassendes Rechtecksignal mit einer Phase einer halben Periode, einer Frequenz von 100 Hz, einer Amplitude von 5.0 und einem Gleichanteil von -1,8 generiert. Die Werte werden im Abstand von 0,2 ms berechnet. Die 200 Daten werden in das Datenfeld *yarray*[] ab dem Index 800 eingetragen.

fkt_round_rectangle

Syntax:

fkt_round_rectangle(**<-** float yarray[],
-> float start_periode,
-> float rect_frequency_hz,
-> float data_frequency_hz,
-> float amplitude,
-> float offset,
-> float start_index,
-> int data_count
-> int round_factor);

Diese Funktion generiert ein, an den Sprungstellen mit Sinus-Halbwellen verrundetes, Rechtecksignal mit *data_count* Daten, das ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Das Rechteck hat eine Phase von *start_periode*. Eine ganze Periode entspricht einer Phase von 1. Für die Phase muß eine Zahl zwischen 0 und 1 angegeben werden. Die Amplitude des Rechtecksignals wird in der Variablen *amplitude* angegeben, die Frequenz in Hertz enthält die Variable *rect_frequency_hz* und der Gleichanteil die Variable *offset*. Das Maß der Verrundung *round_factor* geht von 0 bis 0,5. Ein *round_factor* von 0 ergibt keine

Verrundung. Ein *round_factor* von 0,5 ergibt ein Cosinussignal. Für *data_frequency_hz* muß der Kehrwert des Stützstellenabstandes in Sekunden übergeben werden. Dies entspricht der Abtastfrequenz in Hertz, mit der das kontinuierliche Rechtecksignal abgetastet wird. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index* + *data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start_periode = 0.5;
float rect_frequency_hz = 100;
float data_frequency_hz = 5000;
float amplitude = 5.0;
float offset = -1.8;
float round_factor = 0.3;
int start_index = 800;
int data_count = 200;

...
fkt_round_rectangle(yarray, start_periode,
                    sin_frequency_hz,
data_frequency_hz,
       amplitude, offset, start_index,
       data_count, round_factor);
...
```

In obigem Beispiel wird ein 200 Werte umfassender Sinus mit einer Phase einer halben Periode, einer Frequenz von 100 Hz, einer Amplitude von 5.0 und einem Gleichanteil von -1,8 generiert. Die Sinuswerte werden im Abstand von 0,2 ms berechnet. Der Verrundungsfaktor von 0,3 ergibt eine mittlere Verrundung. Die 200 Daten werden in das Datenfeld yarray ab dem Index 800 eingetragen.

fkt_triangle

Syntax:

fkt_triangle(**<-** float yarray[],
-> float start_periode,
-> float tri_frequency_hz,
-> float data_frequency_hz,
-> float amplitude,
-> float offset,
-> int start_index,
-> int data_count);

Diese Funktion generiert ein Dreiecksignal mit *data_count* Daten, das ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Das Dreieck hat eine Phase von *start_periode*. Eine ganze Periode entspricht einer Phase von 1. Als Phase muß eine Zahl zwischen 0 und 1 gewählt werden. Die Amplitude des Dreiecksignals wird in der Variablen *amplitude* angegeben, die Frequenz in Hertz enthält die Variable *tri_frequency_hz* und der Gleichanteil steht in der Variablen *offset*. Für *data_frequency_hz* muß der Kehrwert des Stützstellenabstands in Sekunden übergeben werden. Dies entspricht der Abtastfrequenz in Hertz, mit der das kontinuierliche Rechtecksignal abgetastet wird. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index* + *data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start_periode = 0.5;
float tri_frequency_hz = 100;
float data_frequency_hz = 5000;
float amplitude = 5.0;
float offset = -1.8;
int start_index = 800;
int data_count = 200;

...
fkt_triangle(  yarray,start_periode,
tri_frequency_hz,     data_frequency_hz,
amplitude, offset,
      start_index, data_count);
...
```

In diesem Beispiel wird ein 200 Werte umfassendes Dreiecksignal mit einer Phase einer halben Periode, einer Frequenz von 100 Hz, einer Amplitude von 5.0 und einem Gleichanteil von -1,8 generiert. Die Werte werden im Abstand von 0,2 ms berechnet. Die 200 Daten werden in das Datenfeld *yarray*[] ab dem Index 800 eingetragen.

fkt_round_triangle

Syntax:

fkt_round_triangle(**<-** float yarray[],
-> float start_periode,
-> float tri_frequency_hz,
-> float data_frequency_hz,
-> float amplitude,
-> float offset,
-> float start_index,
-> int data_count
-> int round_factor);

Diese Funktion generiert ein, an den Sprungstellen mit Sinus-Halbwellen verrundetes, Dreiecksignal mit *data_count* Daten, das ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Das Dreieck hat eine Phase von *start_periode*. Eine ganze Periode entspricht einer Phase von 1. Für die Phase muß eine Zahl zwischen 0 und 1 angegeben werden. Die Amplitude des Dreiecksignals wird in der Variablen *amplitude* angegeben, die Frequenz in Hertz enthält die Variable *tri_frequency_hz* und der Gleichanteil die Variable *offset*. Das Maß der Verrundung *round_factor* geht von 0 bis 0,5. Ein *round_factor* von 0 ergibt keine Verrundung. Ein *round_factor* von 0,5 ergibt ein Cosinussignal. Für *data_frequency_hz* muß der Kehrwert des Stützstellenabstandes in Sekunden übergeben werden. Dies entspricht der Abtastfrequenz in Hertz, mit der das kontinuierliche Dreiecksignal abgetastet wird. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index* + *data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start_periode = 0.5;
float tri_frequency_hz = 100;
float data_frequency_hz = 5000;
float amplitude = 5.0;
float offset = -1.8;
float round_factor = 0.3;
int start_index = 800;
int data_count = 200;

...
```

```
fkt_round_triangle(yarray, start_periode,
                   sin_frequency_hz,
data_frequency_hz,
       amplitude, offset, start_index,
       data_count, round_factor);
...
```

In obigem Beispiel wird ein 200 Werte umfassender Sinus mit einer Phase einer halben Periode, einer Frequenz von 100 Hz, einer Amplitude von 5.0 und einem Gleichanteil von -1,8 generiert. Die Sinuswerte werden im Abstand von 0,2 ms berechnet. Der Verrundungsfaktor von 0,3 ergibt eine mittlere Verrundung. Die 200 Daten werden in das Datenfeld yarray ab dem Index 800 eingetragen.

fkt_sinus

Syntax:

fkt_sinus(**<-** float yarray[],
-> float start_periode,
-> float sin_frequency_hz,
-> float data_frequency_hz,
-> float amplitude,
-> float offset,
-> int start_index,
-> int data_count);

Diese Funktion generiert ein Sinussignal mit *data_count* Daten, das ab dem Index *start_index* in das Datenfeld *yarray*[] eingetragen wird. Der Sinus hat eine Phase von *start_periode* in rad. Seine Amplitude enthält die Variable *amplitude*, die Frequenz beträgt *sin_frequency_hz* in Hertz und der Gleichanteil wird in der Variablen *offset* angegeben. Für *data_frequency_hz* muß der Kehrwert des Stützstellen-abstandes in Sekunden übergeben werden. Dies entspricht der Abtastfrequenz in Hertz, mit der das kontinuierliche Sinussignal abgetastet wird. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *start_index + data_count* groß sein.

Beispiel:

```
float yarray[1000];
float start_periode = 90 / 360;
float sin_frequency_hz = 100;
float data_frequency_hz = 5000;
float amplitude = 5.0;
float offset = -1.8;
```

```
int start_index = 800;
int data_count = 200;

...
fkt_sinus(yarray, start_periode,
sin_frequency_hz,
  data_frequency_hz, amplitude, offset,
  start_index, data_count);
...
```

In diesem Beispiel wird ein 200 Werte umfassender Sinus mit einer Phase von 90°, einer Frequenz von
100 Hz, einer Amplitude von 5.0 und einem Gleichanteil von -1,8 generiert. Die Sinuswerte werden im Abstand von 0,2 ms berechnet. Die 200 Daten werden in das Daten-feld yarray ab dem Index 800 eingetragen.

fkt_sweep

Syntax:

fkt_sweep(<- float yarray[],
-> float standartdrehzahl,
-> float stopdrehzahl,
-> float startphase,
-> float dauer,
-> int data_count);

Diese Funktion generiert ein Datenfeld gemäß der Funktion:

y= 1/drehzahl * cos(drehzahl * t + ι), wobei die Frequenzen von *startdrehzahl* bis *stopdrehzahl* gesweept werden. Mit *startphase* wird die Phasenlänge ι zum Zeitpunkt t = 0 übergeben, mit *dauer* wird bestimmt, wie viele Sekunden der Sweep dauern soll. *Data_count* gibt an, wieviele Stützstellen generiert werden.

Beispiel: Bei einer Paketgröße von 100 Werten und einer Sollwertrate von 100Hz ist *data_count* = 100 und dauer = 0.1

Die Startphase für das nächste zu errechnende Paket steht im Wert *yarray[data_count].* Deshalb muß die Größe des Datenfeldes *yarray* mind. *Data_count* + 1 sein.

floor

Syntax:

int **floor**(**<-** float x);

Die mathematische Funktion **floor()** gibt den auf einen Integerwert abgerundeten Wert von *x* zurück. Der Übergabewert x bleibt unverändert.

Beispiel:

```
float x = 3.9;
int y;

...
y = floor(x);
printf(„y = %d\n“, y); /* Das Ergebnis ist 3
*/
...
```

frac

Syntax:

float **frac**(**<-** float x);

Die Funktion **frac()** gibt die Nachkommastellen der Zahl *x* zurück. Auch bei dieser Standardfunktion des **Sequenzers** bleibt der Übergabewert *x* unverändert.

Beispiel:

```
float x;
float y;
HEADER hd;

void start(PAR);
{
   while(1)
   {
   read_var(1, x, hd); /* lese x */
   y = frac(x);
   write_var(1, y, hd); /* schreibe y */
   if (y < 0.5) stop;
   }
}
```

In obigem Beispiel werden Werte gelesen und ausgegeben, solange die Nachkommastellen von x größer oder gleich 0.5 sind.

Wird am Eingang 1 zum Beispiel der Wert 3.6 eingelesen, wird am Ausgang 1 der Wert 0.6 ausgegeben.

reciprocal_value

Syntax:

reciprocal_value(**<-** float yarray[],
-> float xarray[],
-> int data_count,
-> float faktor);

Diese Funktion berechnet für *data_count* Elemente des Datenfeldes *xarray*[] den Kehrwert und multipliziert ihn mit dem Wert von *faktor*. Die *data_count* Daten werden im Datenfeld *yarray*[] eingetragen:

yarray[n] = (1 / *xarray*[n]) * *faktor*

für n = 0, ..., *data_count* - 1.

Die Anzahl der Elemente der Datenfelder *yarray*[] und *xarray*[] müssen mindestens *data_count* groß sein.

Beispiel:

```
float xarray[1000];
float yarray[1000];
float faktor;
int data_count;
...
faktor = 0.7;
data_count= 1000;
reciprocal_value(yarray, xarray, data_count,
faktor);
...
```

In obigem Beispiel werden von 1000 Daten aus dem Datenfeld *xarray*[] die Kehrwerte gebildet, mit dem Faktor 0,7 multipliziert und in das Datenfeld *yarray* [] eingetragen.

linear_interpolation

Syntax:

linear_interpolation(**<-** float yarray[],
-> int ylen,
-> float xarray[],
-> int xlen,
-> float xfirst_value_next_block);

Diese Funktion interpoliert ein Datenpaket der Länge *xlen* in ein Datenpaket der Länge *ylen*. Durchgeführt wird eine lineare Interpolation. Das ursprüngliche Datenpaket wird im Datenfeld *xarray*[] an die Funktion übergeben. Das interpolierte Datenpaket wird in das Datenfeld *yarray*[] eingetragen.

Für die Interpolation wird bei ***xlen < ylen*** der erste Wert des nächsten Datenpakets benötigt. Dieser Wert wird in der Variablen *xfirst_value_next_block* an die Funktion übergeben. Liegt das nächste Datenpaket nicht vor, wird der letzte Wert des aktuellen Datenpakets *xarray(xarray*[*xlen* - 1]*)* in *xfirst_value_next_block* übergeben.

Für ***xlen > ylen*** wird der erste Wert des nächsten Datenpakets nicht benötigt. Übergeben Sie in diesem Fall einen Dummywert, z.B. *xfirst_value_next_block* = 0.0.

Die Anzahl der Elemente des Datenfeldes *xarray*[] muß mindestens *xlen* + 1 groß sein. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *ylen* groß sein.

Beispiel:

```
HEADER h_xarray;
HEADER h_yarray;

void inter(PAR)
{
   float xarray[1001];
   float xarray_new[1001];
   float yarray[2000];
   int xlen = 350;
   int ylen = 1000;
   float xfirst_value_next_block;
   float xdelta;

   /* Datenpaket vom Eingang 1 einlesen */
   read(1, xarray, xlen, h_xarray);
   while(1)
   {
   /* aktuellen Datenheader kopieren wegen
   aktuellem x0 */
   copy_header(h_yarray, h_xarray);
   /* aktuellen Abstand der Werte auf der
   x-Achse holen */
   get_xdelta(h_xarray, xdelta);
   /* neuen Abstand der Werte auf der x-
   Achse setzen */
   set_xdelta(h_yarray, xdelta * xlen /
   ylen);
   /* ursprüngl. Datenpaket am Ausgang1
   ausgeben */
   write(1, xarray, xlen, h_xarray);
   /* nächstes Datenpaket vom Eingang 1
   einlesen */
   read(1, xarray_new, xlen, h_xarray);
   /* ersten Wert des nächsten Datenpaketes
   übernehmen */
   xfirst_value_next_block = xarray_new[0];
   /* Interpolation durchführen */
   linear_interpolation(yarray, ylen,
   xarray, xlen, xfirst_value_next_block);

   /* interpoliertes Datenpaket am Ausgang 2
   ausgeben */
   write(2, yarray, ylen, h_yarray);
   /* nächstes Datenpaket ins Datenfeld
   xarray[] kopieren */
   copy_channel(xarray, xarray_new, xlen, 1,
   1);
   }
```

```
}
```

Im Beispiel werden Datenpakete der Länge 1000 durch lineare Interpolation auf Datenpakete der Länge 350 reduziert. Die Daten werden kontinuierlich am Eingang 1 eingelesen und unverändert am Ausgang 1 ausgegeben. Die interpolierten Daten werden am Ausgang 2 ausgegeben.

log, ln

Syntax:

float **log**(**<-** float x);,float **ln**(**<-** float x);

Die mathematischen Standardfunktionen **log()** und **ln()** geben den dekadischen bzw. den natürlichen Logarithmus von *x* zurück. Der Übergabewert *x* bleibt unverändert.

Beispiel*:*

```
/* Es wird der Logarithmus des Betrags von x
berechnet */
y = log(abs(x));
```

long_diff

long_diff() ist eine spezielle mathematische Funktion zur Berechnung von ganzen Zahlen, die größer als 2^{24} sind. Diese ganzen Zahlen werden aus zwei Zahlen zusammengesetzt (s. Kapitel 6.3).

Syntax:

long_diff(**->** float x1_data_array[],
-> float x2_data_array[],
<- float y_differenz);,

Diese Funktion subtrahiert von der „ganzen Zahl“ *x1_data_array*[] die ganze Zahl *x2_data_array*[]. Beide Zahlen können größer 2^{24} sein. Die Differenz wird der Variablen *y_differenz* zugewiesen.

Beispiel:

```
float x1_data_array[2];
float x2_data_array[2];
float y_differenz;

...
x1_data_array[0] = 1587;
x1_data_array[1] = 15;
x2_data_array[0] = 1487;
```

```
x2_data_array[1] = 15;
long_diff(x1_data_array, x2_data_array,
y_differenz);
...
```

In obigem Beispiel wird von der Zahl 1587 + 15 * 10^6 die Zahl 1487 + 15 * 10^6 subtrahiert. Das Ergebnis lautet 100.

long_sum

Syntax:

long_sum(**->** float x1_data_array[],
-> float x2_data_array[],
<- float y_summe_array[]);

Auch die Standardfunktion **long_sum()** ist eine spezielle mathematische Funktion zur Berechnung von ganzen Zahlen, die größer als 2^{24} sind. Diese ganzen Zahlen werden aus zwei Zahlen zusammengesetzt (s. Kapitel 6.3).

Diese Funktion addiert die „ganzen Zahlen“ *x1_data_array*[] und *x2_data_array*[]. Beide Zahlen können größer 2^{24} sein. Die Summe wird der Variablen *y_summe_array*[] zugewiesen.

Beispiel:

```
float x1_data_array[2];
float x2_data_array[2];
float y_summe_array[2];

...
x1_data_array[0] = 1587;
x1_data_array[1] = 15;
x2_data_array[0] = 1487;
x2_data_array[1] = 15;
long_diff(x1_data_array, x2_data_array,
  y_summe_array);
...
```

In obigem Beispiel wird die Zahl 1587 + 15 * 10^6 mit der Zahl 1487 + 15 * 10^6 addiert. Das Ergebnis lautet 3074 + 30 * 10^6.

max_min

Syntax:

max_min(**->** float data[],
-> int len,
<- float max,
<- float min);

Diese Funktion ermittelt das Maximum und das Minimum der Daten im Datenfeld *data*[] mit der Anzahl von *len* Werten. Die Ergebnisse stehen in den Variablen *max* bzw. *min*.

Beispiel:

```
void start(PAR)
{
   int len = 40;
   float max, min;
   float data[40];
   ...
   max_min(data, len, max, min);
   ...
}
```

mean_value

Syntax:

mean_value(**->** float data[],
-> int len,
<- float mean_v);

mean_value() berechnet den Mittelwert *mean_v* aus den Daten im Feld *data*[] mit der Anzahl von *len* Werten.

Beispiel:

```
void start(PAR)
{
   int len = 40;
   float mean_v;
   float data[40];
   ...
   mean_value(data, len, mean_v);
   ...
}
```

mul

Syntax:

mul(**<-** float m[],
-> float x[],
-> float y[],
-> int len);

Bei Aufruf der Funktion **mul()** werden die Elemente zweier Datenfelder *x*[] und *y*[] mit der Anzahl *len* Werten miteinander multipliziert. Das Ergebnisdatenfeld ist *m*[]. Es gilt *m*[n] = *x*[n] * *y*[n] (n = 0, ..., len - 1). Die Länge der Datenfelder muß größer gleich *len* sein.

Beispiel:

```
void start(PAR)
{
   int len = 50;
   float prod[50];
   float x[50], y[50];
   ...
   mul(prod, x, y, len);
   /* Das Ergebnis steht in prod[] */
   ...
}
```

not

Syntax:

int **not**(**<-** int x);

Die Funktion gibt Ihnen den bitweise negierten Wert von *x* zurück. Gesetzte Bits werden zurückgesetzt und umgekehrt. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
int x = 5;  /* entspricht 0...0101b */
int y;

...
y = not(x); /* Ergebnis 1...1010b */
...
```

Das Ergebnis lautet $1...1010_b$ (24 Bit), dies entspricht 16777211.

rand

Syntax:

float **rand(->** float x);

rand() gibt eine Zufallszahl (float) zwischen 0 und *x* mit 6 Nachkommastellen aus. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = rand(10.5);
/* y wird eine Zufallszahl zwischen 0 und
10.5 zugewiesen */
```

scale

Syntax:

scale(**<-** float y[],
-> float x[],
-> int len,
-> float faktor,
-> float offset);

Diese Standardfunktion skaliert das Datenfeld *x*[] mit *len* Werten. Das Ergebnisdatenfeld ist *y*[].

Es gilt *y*[n] *= faktor * (x*[n] - *offset)* (n = 0, ..., len - 1). Die Länge der Datenfelder muß größer gleich *len* sein.

Beispiel:

```
void start(PAR)
{
   int len = 50;
   float x[50], y[50];
   float faktor, offset;
   faktor = 2.5;
   offset = 0.3;
   ...
   scale(y, x, len, faktor, offset);
   ...
}
```

sign

Syntax:

float **sign**(**<-** float x);

sign() ermittelt das Vorzeichen von *x*. Die Funktion gibt den Wert -1.0 aus, wenn *x* < 0, den Wert 0.0, wenn *x* = 0 und den Wert 1.0, wenn *x* > 0 ist. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = sign(x);
/* Das Vorzeichen von x wird ausgegeben. */
```

sin, cos, tan

Syntax:

float **sin**(**<-** float x);

float **cos**(**<-** float x);

float **tan**(**<-** float x);

Diese Standardfunktionen berechnen den Sinus, Cosinus und Tangens von *x*. Der Wert von *x* ist im Bogenmaß anzugeben. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = sin(x);
/* y wird der Sinus von x zugewiesen. */
```

sinh, cosh, tanh

Syntax:

float **sinh**(**<-** float x);

float **cosh**(**<-** float x);

float **tanh**(**<-** float x);

Diese Funktionen berechnen den Sinushyperbolikus, Cosinushyperbolikus und Tangenshyperbolikus von *x*. Der Wert von *x* ist im Bogenmaß anzugeben. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = sinh(x);
```

```
/* y wird der Sinushyperbolikus von x
zugewiesen. */
```

sqr

Syntax:

float **sqr**(**<-** float x);

sqr() liefert das Quadrat von *x* zurück. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = sqr(x);
/* das Quadrat von x wird ermittelt und y
zugewiesen. */
```

sqrt

Syntax:

float **sqrt**(**<-** float x);

Bei Aufruf dieser Funktion wird die Quadratwurzel von *x* berechnet. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
y = sqrt(x);
/* die Quadratwurzel von x wird ermittelt
und y zugewiesen. */
```

Für die Quadratbildung von jedem Wert eines Datenfeldes kann die Funktion **mul();** verwendet werden.

sum

Syntax:

sum(**<-** float s[],
-> float x[],
-> float y[],
-> int len);

Die Funktion **sum()** addiert die Elemente zweier Datenfelder *x*[] und *y*[] mit Anzahl der *len* Werte. Das Ergebnisdatenfeld ist s. Es gilt *s*[n] = *x*[n] + *y*[n] (n = 0, ..., len - 1).

Beispiel:

```
void start(PAR)
{
   int len = 50;
   float summe[50];
   float x[50], y[50];
   ...
   sum(summe, x, y, len);
   ...
}
```

Funktionen für die Datenverwaltung

In diesem Kapitel werden die Standardfunktionen des **Sequenzers** beschrieben, die Sie für die Verwaltung der Daten einsetzen können. Hierzu gehören beispielsweise die copy-Befehle, Sprungbefehle, wie goto und return, oder die Start- und Stop-Befehle.

array_char

Syntax:

array_char(**<->** float xarray[],
-> int len,
-> const char *str);

Die Funktion **array_char()** ersetzt alle ASCII-Zeichen im Feld *xarray*[] durch Leerzeichen („ “), ausgenommen der Zeichen im String *str*[]. Diese bleiben unverändert. Die Anzahl der Elemente des Feldes *xarray*[] wird an die Variable *len* übergeben.

Beispiel:

```
void start(PAR)
{
   int len = 40;
   float str[40];
   float x = 1.23;
   float y = 4.56;

   /* erzeugt einen String, der im Feld
   str[] abgelegt wird */
   float_printf(str, len,  "x=%.7g :
   y=%.7g", x, y);
   /* in str[] steht jetzt „x=1.23 : y=4.56“
```

```
    alle Zeichen außer den angegebenen durch
   ' ' ersetzen */
   array_char(str, len, "1234567890.e+-");
   puts("nach array_char() str =");
   /* gibt den String str[] im HYDRA Control
   Fenster aus */
   array_puts(str, len);
}
```

In diesem Beispiel wird aus dem String „x=1.23 :y=4.56“ der String „ 1.23 4.56“ erzeugt.

array_size

Syntax:

array_size(**->** float xarray[],
<- int len);

Bei Aufruf dieser Funktion erhalten Sie die Anzahl der Elemente des Feldes *xarray*[] in der Variablen *len* zurück.

Beispiel:

```
void start(PAR)
{
   int len;
   float str[40];

   array_size(str, len);
   /* Ermittlung der Feldlänge. Die
   Feldlänge wird in len eingetragen. */

}
```

call

Syntax:

call(function);

Mit dem Befehl **call()** rufen Sie eine Funktion mit dem Namen *function* auf. Diese Funktion darf außer dem Parameter PAR keine weiteren Parameter in ihrer Parameterliste enthalten. Wenn die Funktion erst weiter unten im Programmtext definiert wurde, muß der **call()**-Aufruf verwendet werden, andernfalls können Sie die Funktion direkt aufrufen. Nach Beendigung der Funktion, die Sie mit **call()** aufgerufen haben, wird das **Sequenzer** Programm mit dem

Befehl fortgesetzt, der dem **call()**(function)-Aufruf direkt folgt. Die maximale Verschachtelungstiefe der Funktionsaufrufe beträgt 20.

Beispiel:

```
float x;
float y;
HEADER hd;

/* hier beginnt die Ausführung des Programms
*/
void start(PAR)
{
   x = 3.5;
   y = 4.5;
   call (addiere); /* y hat nun den Wert 8.0
   */
   write(1, y, 1, hd);
   stop; /*Terminierung des Sequenzer
   Programms */
}
/* hier beginnt die Funktion */
void addiere(PAR)
{
   y = x + y;
}
```

copy

Syntax:

copy(**<-** float dst[],
-> float src[],
-> int len);

Die Funktion **copy()** kopiert die Daten vom Feld *src*[] in das Feld *dst*[]. Die Anzahl der Werte, die kopiert werden sollen, wird in der Variable *len* übergeben. Die Länge des Datenfeldes *dst* [] darf nicht kleiner als der Wert von *len* sein.

Beispiel:

```
void start(PAR)
{
   int len = 40;
   float src[40];
   float dst[40];
   ...
   copy(dst, src, len);
   ...
}
```

copy_channel

Syntax:

copy_channel(**<-** float yarray[],
-> float xarray[],
-> int data_count,
-> int xchannel_count,
-> int xchannel);

Diese Funktion kopiert *data_count* Daten des Kanals *xchannel* aus dem Datenfeld *xarray*[] und trägt sie in das Datenfeld *yarray*[] ein. Das Datenfeld *xarray*[] besitzt *xchannel_count* Kanäle.

Die Anzahl der Elemente des Datenfeldes *xarray*[] muß mindestens *xchannel_count* * *data_count* groß sein. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *data_count* groß sein. Der Wert von x*channel* darf höchstens so groß wie x*channel_count* sein.

Beispiel:

```
float xarray[16000];
float yarray[1000];
int xchannel_count = 8;
int xchannel = 5;
int data_count;
...
data_count = 1000;
/* Aufruf der Funktion mit Variablen als
Parameter */
copy_channel(yarray, xarray, data_count,
xchannel_count, xchannel);
/* Oder mit direkter Angabe der Werte für
data_count, xchannel_count, xchannel */
copy_channel(yarray, xarray, 1000, 8, 5);
...
```

Im Beispiel werden die ersten 1000 Daten des Kanals 5 aus Datenfeld *xarray*[] in das Datenfeld *yarray*[] kopiert.

copy_header

Syntax:

copy_header(**<-** HEADER y_header
-> HEADER x_header);

Nach Aufruf der Funktion **copy_header()** wird der Datenheader *x_header* in den Datenheader *y_header* kopiert.

Beispiel:

```
HEADER x_header;
HEADER y_header;

void start(PAR)
{
   ...
   copy_header(y_header, x_header);
   ...
}
```

copy_range

Syntax:

copy_range(**<-** float dst[],
-> int dst_ind,
-> float scr[],
-> int scr_ind,
-> int len);

Diese Verwaltungsfunktion kopiert die Daten im Feld *src*[], ab dem Index *src_ind*, in das Feld *dst*[], ab dem Index *dst_ind*. Die Anzahl der zu kopierenden Werte wird in der Variablen *len* übergeben. Die Länge des Datenfeldes *dst*[] darf nicht kleiner als der Wert von *len + dst_ind* sein.

Beispiel:

```
void start(PAR)
{
   int len = 12, dst_ind, src_ind;
   float src[40];
   float dst[40];
   dst_ind = 10;
```

```
   src_ind = 20;

   ...
   /* kopiert von src[], ab dem Index 20,
    12 Werte nach dst[], ab dem Index 10 */
   copy_range(dst, dst_ind, src, src_ind,
   len);
   ...
}
```

float_printf

Syntax:

float_printf(**<-** float sout[],

-> int len,

-> const char *format [,

-> arg1,

-> arg2,

...]);

Die Funktion **float_printf()** erzeugt eine **formatierte Ausgabe in das Datenfeld *sout*[]**. Sie wandelt numerische Werte in Zeichendarstellung um. Die Länge *len* gibt die Anzahl der Elemente im Feld *sout*[] an. Jedes ausgegebene Zeichen wird in ein Feldelement von *sout*[] geschrieben.

Die Zeichenkette *format* enthält zwei Arten von Objekten:

- Gewöhnliche ASCII-Zeichen, die einfach ausgegeben werden,
- Format-Elemente, die jeweils die Umwandlung und Ausgabe des folgenden Arguments *argx* veranlassen (x = 1,2...). Jedes Format-Element beginnt mit dem Zeichen „**%**“. Anschließend folgen die Formatanweisungen, die in der Funktion **printf()**erläutert sind.

Die Anzahl der tatsächlich benötigten Elemente des Feldes *sout*[] wird in *len* zurückgegeben.

Beispiel:

```
float string[40];
float len = 40;
float x = 3.4, y = 5.6;

void start(PAR)
{
   float_printf(string, len, "%5.2f\n", x);
   /* Ergebnis: „␣3.40“ */
```

```
   float_printf(string, len, "%8.2e\n", y);
   /* Ergebnis: „5.60e+00“ */
   float_printf(string,len,
   %7.3g\t%7.3g\n",x,y);
   /*Ergebnis: „␣␣3.4␣->␣␣5.6␣“ */
   float_printf(string, 40, "Wert für x,
      y:%5.2f, %5.2f\n", x, y);
   /* Ergebnis: „Wert␣für␣x,␣y:␣3.40,␣␣5.60“
   */
}
```

float_scanf

Syntax:

float_scanf(**->** float string[],

-> int len,

-> const char *format [,

<- arg1,

<- arg2...]);

Die Funktion **float_scanf()** wandelt die Zeichenkette *string*[] in die entsprechenden numerischen Werte um und speichert diese in den Variablen *arg1, arg2, ...* Diese Variablen müssen vom Typ *float* sein. Die Anzahl der Werte darf nicht größer als der Wert von *len* werden.

Die Zeichenkette *format* enthält Format-Elemente, die die Interpretation der Eingabezeichenfolge steuern. Jedes Format-Element wandelt ein Eingabefeld um und beginnt mit dem Zeichen **%**. Anschließend folgen die Formatanweisungen. Die Eingabefelder in der Eingabezeichenkette *string*[] werden durch Zwischenraum-Zeichen getrennt (Leerzeichen, Tab., etc.).

Gegenüber den Standard C-Routinen kann mit der Funktion **float_scanf()** nur das *float*-Format **%f** verwendet werden. Alle anderen C-üblichen Formate dürfen nur in Zusammenhang mit dem Unterdrückungsoperator ***** eingesetzt werden. Damit sind im wesentlichen folgende Formatanweisungen möglich:

- %f - als Eingabe wird ein dezimal dargestellter Gleitkommawert erwartet.
- %*c - im Eingabestring wird ein Zeichen überlesen.
- %*s - im Eingabestring wird eine Zeichenkette bis zum nächsten Zwischenraum-Zeichen überlesen.

Beispiel:

```
void start(PAR)
{
```

```
    int len = 50;
    float string[50];
    float x, y;
    x = 2.5;
    y = 0.3;

    /* Werte in String schreiben */
    float_printf(string,len,  "x= %5.2f y=
    %5.2f\n", x, y);
    /* in string[] steht „x=␣␣2.50␣y=␣␣0.30“ */

    x=0.0;
    y=0.0;

    /* Werte von String wieder zurücklesen */
    float_scanf(string, len, "%*s %f %*s %f",
    y, x);
    /* Ergebnis: y= 2.50, x = 0.30“ */
}
```

get_ini_float

Syntax:

get_ini_float(**->** const char *str_application,
-> const char *str_key,
<- float y_value,
-> const char *str_filename);

Diese Verwaltungsfunktion holt eine Zeichenkette aus einer INI-Datei und konvertiert sie in einen *float*-Wert. Der Name des Abschnitts der INI-Datei ist *str_application* (z.B. H404). Der Name der Variablen des angegebenen Abschnitts ist *str_key* (z.B. ABTASTRATE). Der Name der INI-Datei ist *str_filename* (z.B. SEQUENZ.INI).

Sie können auch ein Laufwerk und einen Pfad angeben. Bei Pfadangaben ist zu beachten, daß das Zeichen „\“ mit Doppel – Backslash („\\“) geschrieben werden muß. Wird kein Laufwerk und kein Pfad angegeben, so wird im aktuellen Laufwerk und Verzeichnis nach der angegebenen Datei gesucht. Der Wert wird der Variablen *y_value* zugewiesen.

Beispiel:

```
float y_value;
...
get_ini_float("H404", "ABTASTRATE",
    y_value, "SEQUENZ.INI");
...
```

Enthält die Datei SEQUENZ.INI beispielsweise folgenden Eintrag:

[H404]

ABTASTRATE = 1.5

so ist das Ergebnis:

y_value = 1.5.

goto

Syntax:

goto(**->** labelname);

Die Funktion **goto()** unterbricht das **HYDRA Sequenzer** Programm und setzt die Bearbeitung an der Programmarke *labelname* fort.

Beispiel:

```
float x;
float y;
HEADER hd;

void start(PAR)
{
   Marke1:
   /* lese x */
   read(1, x, 1, hd);
   if (x >= 5.0) call(Error);
   /* schreibe x */
   write(1, x, 1, hd);
   goto(Marke1);
}
void Error(PAR)
{
   write(2, x, 1, hd);
}
```

In obigem Beispiel werden Werte auf Eingangskanal 1 gelesen und auf Ausgangskanal 1 ausgegeben, solange deren Wert kleiner als 5.0 ist. Alle anderen Werte werden auf Ausgangskanal 2 ausgegeben.

Programmmarken und die entsprechenden *goto*-Befehle sollten nur in Ausnahmefällen verwendet werden. Obiges Beispiel läßt sich leicht mit einer *while*-Schleife realisieren. Sprünge zwischen Funktionen müssen vermieden werden.

insert_channel

Syntax:

insert_channel(**<-** float yarray[],
-> float xarray[],
-> int data_count,
-> int ychannel_count,
-> int ychannel);

Diese **Sequenzer** Funktion kopiert *data_count* Daten aus dem Datenfeld *xarray*[] in den Kanal *ychannel* des Datenfeldes *yarray*[] mit *ychannel_count* Kanälen.

Die Anzahl der Elemente des Datenfeldes *xarray*[] muß mindestens *data_count* groß sein. Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens *data_count* * *ychannel_count* groß sein. Der Wert von *ychannel* darf höchstens so groß wie *ychannel_count* sein und beginnt bei 1.

Beispiel:

```
float xarray[1000];
float yarray[8000];
int ychannel_count = 8;
int ychannel = 2;
int data_count;
...
data_count = 1000;
insert_channel(yarray, xarray, data_count,
               ychannel_count, ychannel);
...
```

In obigem Beispiel werden 1000 Daten aus Datenfeld *xarray*[] in den Kanal 2 des Datenfeldes *yarray*[] kopiert.

move_channel

Syntax:

move_channel(**<-** float yarray[],
-> int ystartindex,
-> int ychannel_count,
-> int ychannel,
-> float xarray[],
-> int xstartindex,
-> int xchannel_count,
-> int xchannel,
-> int len);

Diese Funktion kopiert *len* Daten des Kanals *xchannel* aus dem Datenfeld *xarray*[] mit *xchannel_count* Kanälen ab dem Feldindex *xstartindex*. Die *len* Daten werden in den Kanal *ychannel*[] des Datenfeldes *yarray*[] mit *ychannel_count* Kanälen ab dem Feldindex *ystartindex* eingetragen.

Die Anzahl der Elemente des Datenfeldes *yarray*[] muß mindestens (*ystartindex* + *len*) * *ychannel_count* groß sein. Die Anzahl der Elemente des Datenfeldes *xarray*[] muß mindestens (*xstartindex* + *len*) * *xchannel_count* groß sein. Der Wert von *ychannel* darf höchstens so groß wie *ychannel_count* sein und beginnt bei 1. Der Wert von *xchannel* darf höchstens so groß wie *xchannel_count* sein.

Beispiel:

```
float xarray[6000];
float yarray[1500];
int xchannel_count = 8;
int ychannel_count = 6;
int xchannel = 5;
int ychannel = 2;
int xstartindex = 500;
int ystartindex = 0;
int len;

...
len = 250;
move_channel(yarray, ystartindex,
ychannel_count,
      ychannel, xarray, xstartindex,
      xchannel_count, xchannel, len);
...
```

In diesem Beispiel werden 250 Daten des Kanals 5 aus Datenfeld *xarray*[] ab dem Feldindex 500 in den Kanal 2 des Datenfeldes [] ab dem Feldindex 0 kopiert.

return

Syntax:

return;

return() beendet die Abarbeitung einer Funktion und setzt die Verarbeitung mit der dem Funktionsaufruf folgenden Anweisung fort. Die maximale Verschachtelungstiefe der Funktionsaufrufe ist 20. Am Ende einer Funktion braucht kein **return()** stehen. Innerhalb einer Funktion können Sie mehrere **return()**-Anweisungen einsetzen.

Beispiel:

```
float y;
HEADER hd;

/* zweite Aufrufebene void calc1(PAR,
floaty) */
{
   ....
   return;
}

/* erste Aufrufebene void berechne(PAR,
float y) */
{
   ...
   if (y < 0) return;
   calc1(p, y);
   ...
}

/* Startfunktion */
void start(PAR)
{
   read(1, y, 1, hd);
   berechne(p, y);
   write(1, y, 1, hd);
   stop;
}
```

In diesem Beispiel wird aus der Hauptroutine die Funktion *berechne* aufgerufen und aus dieser wiederum die Funktion *calc1.* Ist in der

Funktion *berechne* der Wert von y negativ, wird direkt in die Funktion *start* zurückgesprungen.

In der Funktion *start* darf kein *return* verwendet werden, ansonsten wird eine Fehlermeldung ausgegeben.

reverse_array

Syntax:

reverse_array(<-float dst[],
->float src[],
->int start_index,
->int len) ;

Bei dieser Funktion werden *len* Einträge aus *src[]* ab dem *start_index* in umgekehrter Reihenfolge in *dst[]* ab *start_index* kopiert, wobei *src[]* != *dst[]* sein muß.

Beispiel:

```
void start(PAR)
{
   int len = 10;
   int start_index = 0;
   float src[10];
   float dst[10];
   ...
   reverse_array(dst, src, start_index,
   len);
   /* das ganze Feld src wird umgedreht in
   dst abgelegt */
}
```

search_index

Syntax:

search_index(->float array[],
->int start_index,
->int stop_index,
->float value,
->int option,
<-int index) ;

Mit der Funktion **serch_index()** wird nach dem Wert *value* in *array* gesucht. Das Array wird von *start_index* bis *stop_index* durchsucht, wenn das *array[]* rückwärts durchsucht werden soll, muß *start_index*

kleiner *stop_index* sein. *Index* enthält den zugehörigen Index des ersten gefundenen Wertes, für den das Suchkriterium erfüllt ist. Wurde kein Wert gefunden, auf den das Suchkriterium zutrifft, enthält *index* den Wert –1.

In *option* werden die Suchkriterien vorgegeben. Gültige Werte für *optionen* sind 1 oder 2. Wenn der Wert 1 ist, wird der erste Index zurückgegeben, dessen Wert >= *value* ist. Wenn der Wert 2 ist, wird der erste Index zurückgegeben, dessen Wert <= *value* ist.

Beispiel:

```
void start(PAR)
{
   float werte[5], value;
   int start_index, stop_index, option,
   index;
   werte[0] = 1 ;
   werte[1] = 2 ;
   werte[2] = 3 ;
   werte[3] = 4 ;
   werte[4] = 5 ;
   start_index = 0; stop_index = 4;
   /* festlegen der Start- und Stopwerte */
   value = 3; /* Wert, der in dem array
   gesucht werden soll */
   option = 1;
   search_index(werte, start_index,
               stop_index, value, option,
               index);
   /* index enthält nun den Wert 2 */
   option = 2;
   search_index(werte, start_index,
               stop_index, value, option,
               index);
   /* index enthält nun den Wert 0 */
   start_index = 4; stop_index = 0;
   option = 1;
   search_index(werte, start_index,
               stop_index, value, option,
               index);
   /* index enthält nun den Wert 4 */
   option = 2;
   search_index(werte, start_index,
               stop_index, value, option,
               index);
   /* index enthält nun den Wert 2 */
   option = 1;
   value = 6;
   search_index(werte, start_index,
               stop_index, value, option,
               index);
```

```
   /* index enthält nun den Wert -1 */
}
```

start_cmd

Syntax:

start_cmd(){Befehlsliste}

Die Funktion **start_cmd()** stellt eine Befehlsliste zur Konfiguration von HYDRA-Blöcken zusammen. Es sind maximal 40 Befehle pro **start_cmd()**-Anweisung möglich. Die Befehle müssen mit dem Semikolon getrennt werden. Anschließend können Sie die Befehlsliste mit der **write_cmd()**-Funktion ausgeben.

Jeder dieser Befehle entspricht einem Menüpunkt in der Eingabedialogbox des jeweiligen Funktionsblockes. So hat z.B. der Befehl TRIGGER_STOP=PEGEL seine Entsprechung in der jeweiligen A/D Wandler Dialogbox bei "Trigger Stop" im Bereich "Triggerfunktionen" bei der Option "Reaktion bei: Pegel".

Beispiel:

```
float x, y = 2;
void start(PAR)
{
   x = 5;
   y = 1;
   /* Initialisierungs-Kommando */
   start_cmd()
      {
         ABTASTRATE = x;
         TRIGGER_START = FLANKE;
         TRIGGER_START_FLANKE = STEIGEND;
         TRIGGER_START_PEGEL_WERT = y + 0.2;
         START;
      }
   /*Sende Kommando über Ausgang 1 */
   write_cmd(1);
   stop;
}
```

stop

Syntax:

stop;

stop beendet die Ausführung des **HYDRA Sequenzer** Programms. Das **HYDRA Sequenzer** Programm wird erst wieder nach erneutem Start-Befehl ausgeführt, und zwar ab der Startfunktion.

Beispiel:

```
float x;
HEADER hd;

void start(PAR);
{
   while(1)
   {
      /* lese x */
      read(1, x, 1, hd);
      /* Programmausstieg */
         if (x < 0) stop;
      /* schreibe x */
            write(1, x, 1, hd);
   }
}
```

In obigem Beispiel werden Werte gelesen und ausgegeben, solange *x* nicht negativ ist. Bei negativen x-Werten wird das **Sequenzer** Programm beendet.

write_ini_float

Syntax:

write_ini_float(-> const char *str_application,
-> const char *str_key,
-> float y_value,
-> const char *str_filename);

Diese Verwaltungsfunktion konvertiert einen *float*-Wert in eine Zeichenkette und schreibt diese in eine INI-Datei. Der Name des Abschnitts der INI-Datei ist *str_application*. Der Name des Abschnitts ist *str_key*. Die zu schreibenede Variable ist *y_value.* Der Name der INI-Datei ist *str_filename*. Sie können auch ein Laufwerk und einen Pfad angeben. Wird kein Laufwerk und kein Pfad angegeben, so wird im aktuellen Laufwerk und Verzeichnis nach der angegebenen Datei gesucht. Existiert im Suchpfad keine INI-Datei mit angegebenem Namen, so wird diese neu angelegt. Besteht in der INI-Datei bereits ein Eintrag für diese Variable, wird diese überschrieben. Bei Pfadangaben ist zu beachten, daß das Zeichen „\“ mit Doppel-Backslash („\\“) geschrieben werden muß.

Beispiel:

```
float y_value;
...
y_value = 3.05;
```

write_ini_float(„EINTRAG“,“WERT“,y_value,“C:\\test\\test.ini“);

...

Nach dem Ausführen dieses Befehls steht in der Datei test.ini im Verzeichnis c:\test folgender Eintrag:

[EINTRAG]
WERT = 3.05

Zeitfunktionen

Mit den Standardfunktionen, die in diesem Kapitel erklärt werden, steuern Sie das Zeitverhalten Ihres **HYDRA Sequenzer** Programms.

get_time

Syntax:

float **get_time**(**->** float x);

Die Funktion **get_time()** gibt als Ergebnis die aktuelle Prozessorzeit in Millisekunden zurück, erhöht um den Wert von *x*. Der Wert von *x* muß in Millisekunden angegeben werden. Das Ergebnis dieser Funktion darf nur für die Funktionen **time_diff()** und **time_plus()** verwendet werden.

Beispiel:

```
float y;
...
y = get_time(1000);
/* y wird die, um 1000 Millisekunden
erhöhte,
aktuelle Prozessorzeit in Millisekunden
zugewiesen */
...
```

time_diff

Syntax:

time_diff(**->** float time_1,
-> float time_2,
<- float time_delta);

Diese Funktion subtrahiert vom Wert *time_1* den Wert *time_2*. Das Ergebnis wird der Variablen *time_delta* in Millisekunden zugewiesen. Der Wert für *time_1* und *time_2* muß mit **get_time()** oder **time_plus()** bestimmt worden sein.

Beispiel:

```
float time_start;
float time_delta;
...
time_start = get_time(0);
...
time_diff(get_time(0), time_start,
time_delta);
...
```

time_plus

Syntax:

time_plus(**<->** float time,
-> float time_delta);

Diese Funktion addiert die Werte von *time* und *time_delta*. Die Variable *time* muß mit **get_time()** bzw. mit **time_plus()** bestimmt worden sein. *time_delta* ist in Millisekunden anzugeben. Das Ergebnis wird der Variablen *time* zugewiesen.

Beispiel: siehe **wait_until().**

wait

Syntax:

wait(**->** float x);

Die Funktion **wait()** unterbricht die Programmbearbeitung für die in *x* angegebene Anzahl von Millisekunden. Der Übergabewert *x* bleibt unverändert.

Beispiel:

```
wait(500);
/* warte eine halbe Sekunde */
```

wait_until

Syntax:

wait_until(**->** float value);

Diese Funktion wartet auf die in *value* angegebene absolute Prozessorzeit. Der Wert für *value* muß zuvor mit **time_plus()** oder **get_time()** berechnet worden sein.

Beispiel:

```
float time;
float time_delta;

...
/* aktuelle Prozessorzeit holen time = */
get_time(0);
...
time_delta = 500;
/* zur aktuellen Prozessorzeit  0,5 s
hinzuaddieren */
time_plus(time, time_delta);
/* warten bis die vorgegebene Prozessorzeit
erreicht ist */
wait_until(time);
...
```

Funktionen für den Zugriff auf den HEADER

Eine weitere Kategorie bilden die Standardfunktionen, die den Zugriff auf den Datenheader regeln. Im Datenheader finden sich Informationen für die Interpretation der Daten .

Die zur Verfügung stehenden Standardfunktionen für den Datenheader werden in den folgenden Kapiteln vorgestellt.

init_header

Syntax:

init_header(**<-** HEADER h_header);

Die Funktion **init_header()** initialisiert die Variable *h_header*. Diese Variable enthält den Datenheader und muß vom Typ HEADER sein. Folgende Einstellungen werden bei der Initialisierung verwendet:

- Länge des Datenpaketes = 1;
- Einheit der x-Achse = MILLISEKUNDEN
- Einheit der y-Achse = VOLT
- Erster Wert auf der x-Achse = 0
- Abstand der Werte auf der x-Achse = 1
- Kleinster y-Wert = -10
- Bereich der y-Achse = 20
- Anzahl der Kanäle = 1
- Meßende-Flag = FALSE (FALSE = 0, Meßende-Flag ist nicht gesetzt).

Beispiel:

```
HEADER h_header;
...
void start(PAR)
{
   ...
   init_header(h_header);
   ...
}
```

get_channel_count

Syntax:

get_channel_count(**->** HEADER hd,
<- int kanalzahl);

Die Funktion **get_channel_count()** ermittelt die im Datenheader *hd* eingestellte Kanalzahl.Das Ergebnis wird in die Variable *kanalzahl* eingetragen. Mit der Kanalzahl können vielkanalige Daten verarbeitet werden, und zwar mit Hilfe der Funktionen **copy_channel(), move_channel()** und **insert_channel()**.

Mit der Funktion **set_channel_count()** läßt sich die Anzahl der Kanäle im Datenheader einstellen.

Beispiel:

```
HEADER hd;

void start(PAR)
{
   int len = 40, ch_count;
   float src[400];

   ...
   read(5, src, 400, hd);
   /* ermittelt die Anzahl der Kanäle im
   Datenstrom */
   get_channel_count(hd, ch_count);
   write(5, src, len * ch_count, hd);
}
```

get_x0

Syntax:

get_x0(**->** HEADER hd,
<- float x0);

Die Funktion **get_x0()** ermittelt den Wert *x0* im Datenheader *hd*. Der Wert *x0* gibt die Position des ersten Wertes auf der x-Achse an. Er kann mit der Funktion **set_x0()** gesetzt werden.

Beispiel:

```
float z[100];
HEADER hd;
float x0;
```

```
void start(PAR)
{
   /*die x-Achse beginnt bei -10.0 */
   set_x0(hd, -10.0);
   /* ermittle x0 */
   get_x0(hd, x0);
   ...
}
```

get_xdelta

Syntax:

get_xdelta(**->** HEADER hd,
<- float xdelta);

Die Funktion **get_xdelta()** ermittelt die Inkrementwerte *xdelta* für die x-Achse im Datenheader *hd*. Der Wert *xdelta* beschreibt den Abstand zweier Werte auf der x-Achse. Er kann mit der Funktion **set_xdelta()** gesetzt werden.

Beispiel:

```
HEADER hd;
float delta;

void start(PAR)
{
   /* Abstand zweier Werte auf der  x-Achse
   */
   set_xdelta(hd, 0.1);
   /* ermittelt xdelta */
   get_xdelta(hd, delta);
   ...
}
```

set_channel_count

Syntax:

set_channel_count(**<-** HEADER hd,
-> int kanalzahl);

Die Funktion **set_channel_count()** setzt die Kanalzahl *kanalzahl* im Datenheader *hd* für die Vielkanalübertragung. Mit der Vielkanalübertragung können vielkanalige Daten verarbeitet werden, und zwar mit Hilfe der Funktionen **copy_channel(), move_channel()** und **insert_channel()**.

Beispiel:

```
HEADER hd;

void start(PAR)
{
   int len = 40, src_ch_count, src_channel;
   float src[400];
   float dst[40];

   /* 10 Kanäle in src[] */
   src_ch_count = 10;
   /* Kanal 5 wird selektiert */
   src_channel = 5;
   ...
   /* kopiert Kanal 5 aus src[] in dst[] */
   copy_channel(dst, src, len,
   src_ch_count,src_channel);
   /* in hd Anzahl der Kanäle  auf 1 setzen
   */
   set_channel_count(hd, 1);
   write(1, dst, len, hd);
   ...
}
```

set_lastblock

Syntax:

set_lastblock(**<-** HEADER hd,
-> float status);

Die Funktion **set_lastblock()** dient der Initialisierung des Meßende-Bits im Header *hd*. Hat die Variable *status* den Wert TRUE, wird das Meßende-Bit gesetzt, für den Wert FALSE wird das Meßende-Bit gelöscht. Das Meßende-Bit kennzeichnet den Abschluß eines zeitkontinuierlichen Datenstroms. Es wird im Datenheader gesendet.

Je nach Applikation ist es notwendig, das Ende einer Messung zu kennzeichnen (z.B. HYDRA Statistik-Block: Mitteln einer Messung über mehrere Datenblöcke hinweg). Mit der Funktion **test_lastblock()** kann der Zustand des Meßende-Bits überprüft werden.

Beispiel:

```
HEADER hd;

void start(PAR)
{
   ...
   /*setzt Meßende-Bit */
   set_lastblock(hd, TRUE);
   ...
}
```

set_x0, set_y0

Syntax:

set_x0(**<-** HEADER hd,
-> float x0);

set_y0(**<-** HEADER hd,
-> float y0);

Diese Funktionen setzen bei der Achsenskalierung die kleinsten Werte für die x- bzw. y-Achse für die in *hd* angegebene HEADER Variable.

Beispiel:

```
float z[100];
HEADER hd;

void start(PAR)
{
   ...
   /*die x-Achse beginnt bei -10.0 */
   set_x0(hd, -10.0);
   /*die y-Achse beginnt bei -20.0 */
   set_y0(hd, -20.0);
   write(1, z, 100, hd);
   ...
}
```

set_xdelta

Syntax:

set_xdelta(**<-** HEADER hd,
<- float x_delta);

set_xdelta() setzt bei der Achsenskalierung die Inkrementwerte *x_delta* für die x-Achse der HEADER Variablen *hd*.

Der Wert *x_delta* beschreibt den Abstand zweier Werte auf der x-Achse. Er kann mit der Funktion **get_xdelta()** abgefragt werden.

Beispiel:

```
float z[100];
HEADER hd;

void start(PAR)
{
   /* die x-Achse beginnt bei -10.0 */
   set_x0(hd, -10.0);
   /* für jeden Wert wird x um 2.0 erhöht */
   set_xdelta(hd, 2.0);
   write(1, z, 100, hd);
   /* die x-Achse am Oszilloskop zeigt nun
   */ Werte von -10.0 bis 190.0 an */

}
```

set_xtype, set_ytype

Syntax:

set_xtype(**<-** HEADER hd,
-> TYPE);
set_ytype(**<-** HEADER hd,
-> TYPE);

Bei der Achsenskalierung werden nach Aufruf dieser Funktionen die Achseneinheiten für die x-Achse bzw. die y-Achse der HEADER Variablen *hd* gesetzt.

Für den Datentyp TYPE sind folgende Werte zulässig:

AMPERE	DEZIBEL	HERTZ
DAVOLT	GRAMM	JOULE
GRADCELSIUS	INT	KILONEWTON
HISTO-GRAMM	KILOHERTZ	LOGIK
KILOGRAMM	KILOWATT	MIKROFARAD
KILOOHM	METER	MILLIBAR
LONG	MILLI-AMPERE	MILLISEKUNDEN
MIKROMETER	MILLIMETER	MEGA-OHM
MILLIGRAMM	MINUTEN	NEWTONMETER
MILLIVOLT	NEWTON	POWERSPEKTRUM
NANOFARAD	PIKOFARAD	SQR
OHM	SEKUNDEN	USEREDIT1
PROZENT	UHRZEIT	USEREDIT2
SQRT STATISTIK	WATT	USEREDIT3
VOLT	ZENTIMETER.	USEREDIT4
WORD	BAR	WINKELGRADE
AMPSPEKTRUM	DWORD	G

Beispiel:

```
float z[100];
HEADER hd;
void start(PAR)
{
   /* die x-Achse beginnt bei -10.0 */
   set_x0(hd, -10.0);

   /*für jeden z-Wert wird x um 2.0  erhöht
   */
   set_xdelta(hd, 2.0);

   /* x-Einheit Sekunden */
   set_xtype(hd, SEKUNDEN);

   /* y-Einheit ist BAR */
   set_ytype(hd, BAR);

   write(1, z, 100, hd);

   /* die x-Achse am Oszilloskop  zeigt nun
   Werte von -10.0 bis 190.0 Sekunden an.
   Die y-Achse ist in BAR skaliert. */
```

```
}
```

set_yrange

Syntax:

set_yrange(**<-** HEADER hd,
<- float y_range);

set_yrange() setzt bei der Achsenskalierung den Bereich *y_range* für die y-Achse der HEADER Variablen *hd*.

Beispiel:

```
float z[100];
HEADER hd;

void start(PAR)
{
   /* die y-Achse beginnt bei -10.0 */
   set_y0(hd, -10.0);

   /* der Bereich der y-Achse ist  20.0 */
   set_yrange(hd, 20.0);

   write(1, z, 100, hd);
   /* die y-Achse am Oszilloskop zeigt nun
   Werte von -10.0 bis 10.0 an */
}
```

test_lastblock

Syntax:

test_lastblock(**->** HEADER hd,
<- float status);

Die Funktion **test_lastblock()** dient der Überprüfung des Meßende-Bits im Header *hd*. Hat der Rückgabewert *status* nach dem Aufruf den Wert TRUE, so ist im Header *hd* das Meßende-Bit gesetzt, hat *status* den Wert FALSE, ist das Meßende-Bit gelöscht. Das Meßende-Bit kennzeichnet den Abschluß eines zeitkontinuierlichen Datenstromes. Es wird im Datenheader gesendet.

Je nach Applikation ist es notwendig, das Ende einer Messung zu kennzeichnen (z.B. HYDRA Statistik-Block: Mitteln einer Messung über mehrere Datenblöcke hinweg). Das Meßende-Bit kann mit der Funktion **set_lastblock()** gesetzt oder gelöscht werden.

Beispiel:

```
void start(PAR)
{
   HEADER hd;
   float status;

   ...
   /* testet Meßende */
   test_lastblock(hd, status);
   if (status == FALSE)
   {
      ...
   }
}
```

Ein-/Ausgabe-Funktionen

Dieses Kapitel stellt Ihnen die zur Verfügung stehenden Ein-/Ausgabe-Funktionen des **Sequenzers** vor. Mittels dieser Funktionen können Eingangsdaten von vorgeschalteten Funktionsblöcken abgeholt und Ausgangsdaten an nachfolgende Funktionsblöcke weitergesendet werden.

Die Zählung der 1 bis 16 Ein- und Ausgangsbuttons läuft von 1-16.

init_read_par

Syntax:

init_read_par(**->** int Eingangsbutton,
-> int mode);

Die Funktion **init_read_par()** initialisiert die parallele Leseroutine für den Eingang *input.* Die Funktion muß für jeden Eingang, von dem parallel gelesen werden soll, einmal aufgerufen werden. Sie darf jedoch je Eingang nur einmal im Sequenzer – Programm aufgerufen werden. Die parallele Leseroutine entkoppelt den Datenstrom von der Programmverarbeitung. Es entsteht eine „intelligente“ Eingabeschnittstelle, die ein Datenpaket zwischenspeichern kann.

Die Betriebsart stellen Sie mit der Konstanten *mode* ein. Es werden zwei Betriebsarten unterschieden:

- *mode = 0*,
  Betriebsart 0 initialisiert das synchronisierte Lesen vom jeweiligen Eingang.
- *mode = 1*,
  Betriebsart 1 initialisiert das kontinuierliche (asynchrone) Lesen vom jeweiligen Eingang.

Betriebsart 0

liefert der Einleseroutine erst dann neue Werte, wenn die vorherigen Daten abgenommen wurden und neue Daten auch tatsächlich angekommen sind. Damit wird sichergestellt, daß alle Daten zur Verarbeitung gelangen bzw. die Datenverarbeitung synchronisiert zum Datenstrom abläuft.

Betriebsart 1

liefert der Einleseroutine, im Unterschied zur Betriebsart 0, immer die aktuellsten Eingangsdaten, d.h. nichtverarbeitete Daten werden verworfen. Damit können Zustände auf ihren aktuellen Wert hin überprüft werden, das heißt, es werden immer die aktuellsten Werte verarbeitet

.

Mit der Funktion **test_read_par()** können Sie den Status der Einlesefunktion überprüfen.

Im Detail: Die Funktion **init_read_par()** wirkt sich auf die vier Einleseroutinen **read(), read_var(), read_par()** und **read_par_var()** aus. Jeweils zwei der vier Routinen unterscheiden sich in der Datenübernahme. Das heißt, zwei Routinen warten auf ankommende Daten, die anderen beiden springen ohne Datenübernahme sofort zum **Sequenzer** Programm zurück (terminieren), auch wenn keine neuen Daten angekommen sind.

Die folgende Tabelle stellt die Unterschiede der Einleseroutinen in Zusammenhang mit der Funktion **init_read_par()** dar. Detaillierte Angaben über die **Sequenzer** Standardfunktionen (Einleseroutinen) finden Sie in den folgenden Unterkapiteln, bei der Beschreibung der Funktionen selbst.

Einleseroutine	Read() / read_var()			read_par() / read_par_var()	
Initialisierung des Eingangs, init_read_par()	keine	synchron (mode=0)	asynchron (mode=1)	Synchron (mode=0)	asynchron (mode=1)
Zwischenpufferung eines Eingangs-datenpakets		X	X	X	X
Einleseroutine terminiert (springt zum Sequenzer Programm zurück), wenn keine Daten vorliegen				X	X
nicht rechtzeitig abgeholte Daten werden verworfen			X		X

Aus der Tabelle geht hervor, welche Fälle zu unterscheiden sind:

Haben Sie die Initialisierungsroutine **init_read_par()** aufgerufen, so kann ein Eingangsdatenpaket zwischengepuffert werden. Dies gilt für alle vier Einleseroutinen und unabhängig von der gewählten Betriebsart.

1. Fall:

Betrachten Sie nun die Einleseroutinen **read_par()** und **read_par_var()** (rechte Spalte). Im Gegensatz zu den beiden anderen Einleseroutinen springen diese Routinen sofort zum aufrufenden **Sequenzer** Programm zurück, wenn keine Daten zum Einlesen bereitstehen, das heißt sie werden beendet/terminiert. Das **Sequenzer** Programm bearbeitet also unverzüglich den Befehl, der unmittelbar auf die Einleseroutine folgt.

Die „Verhaltensweise“ der parallelen Leseroutine hängt von der Betriebsart ab. Haben Sie synchron (mode=0) ausgewählt, wartet die Leseroutine, bis sie Eingangsdaten an das Sequenzerprogramm „loswerden“ kann, bevor sie die nächsten Eingangsdaten entgegennimmt. Haben Sie dagegen kontinuierlich (mode=1) eingegeben, werden die nicht rechtzeitig abgeholten Eingangsdaten verworfen.

Der Unterschied zwischen **read_par()** und **read_par_var()** liegt in der Anzahl der Daten, die eingelesen werden. Bei **read_par()** ist es ein Datenpaket, bei **read_par_var()** ist es immer nur ein Datenwert.

2. Fall:

Im Gegesatz zu Fall 1 warten die Einleseroutinen **read()** und **read_var()** (mittlere Spalte) auf ankommende Eingangsdaten. Das Verhalten der Leseroutine hängt wiederum von der Betriebsart ab. Bei mode=0 wartet wieder die Leseroutine, bei mode=1 überschreibt sie die anstehenden Eingabewerte.

Der Unterschied zwischen **read()** und **read_var()** liegt wiederum in der Anzahl der Daten, die eingelesen werden. Bei **read()** ist es ein Datenpaket, bei **read_var()** ist es immer nur ein Datenwert.

Das folgende Beispiel zeigt Einsatzmöglichkeiten der Einleseroutinen.

Beispiel*:*

```
/* Wenn von Eingang 2 ein Wert >= 1 ankommt, dann
werden die von Eingang 1 gelesenen Daten
ausgegeben. */

HEADER h;
HEADER h_x;
int len;
float z[30];
float x,y = 2;

void start(PAR)
{
   int i;
   int show_data = 0;
   /* parallele Leseroutine aktivieren */
   init_read_par(1, 1); /* asynchron */

   /* parallele Leseroutine aktivieren */
   init_read_par(2, 0); /* synchron */
   while(1)
   {
      while(1)
      {
         /* überprüfen, ob weiterhin Daten
         ausgegeben werden sollen */

         read_par_var(2, show_data, h);
         len = 30;
         /* feststellen, ob Daten am Eingang
         1 angekommen sind */

         test_read_par(1, len);
         if(len == 0) continue;
```

```
            /* Daten sind angekommen, deshalb
            Daten jetzt lesen */

            read_par(1, z, len, h_x);
            if(len != 0) break;
         }
         if(show_data)
         {
            show_header(h_x);
            /*Ausgabe der gelesenen Werte auf
            dem Bildschirm*/
            loop(i,0,len)
            {
               /* Daten ausgeben */
               printf("z[%2d] = %4.7g  ", i,
               z[i]);
               if((i+1)%5 == 0)  puts("");
            }
         }
      }
   stop;
   }
```

init_write_par

Syntax:

init_write_par(**->** int output);

Diese Funktion initialisiert die parallele Schreibroutine für den Ausgang *output*. Die Funktion muß für jeden Ausgang, auf den parallel geschrieben werden soll, einmal aufgerufen werden. Die parallele Schreibroutine entkoppelt den Datenstrom von der Programmverarbeitung. Es ensteht eine „intelligente" Ausgabeschnittstelle, die ein Datenpaket zwischenspeichern kann.

Die Ausgabe der Daten läuft parallel zum **Sequenzer** Programm ab. Mit der Funktion **test_write_par()** kann der Zustand der laufenden Ausgabe überprüft werden.

Im Detail: Die Funktion **init_write_par()** wirkt sich auf die vier Ausgaberoutinen **write(), write_var(), write_par()** und **write_par_var()** aus. Diese vier Routinen unterscheiden sich in der Datenübergabe. Das heißt, jeweils zwei der Routinen mit dem Schreiben warten, bis die zuvor geschriebenen Ausgangsdaten an den nächsten Funktionsblock weitergegeben worden sind.

Die folgende Tabelle stellt die Unterschiede der Ausgaberoutinen in Zusammenhang mit der Funktion **init_write_par()** dar. Detaillierte Angaben über die **Sequenzer** Standardfunktionen

(Ausgaberoutinen) finden Sie in den folgenden Unterkapiteln, bei der Beschreibung der Funktionen selbst.

Schreibroutine	**Write() / write_var()**		**write_par() / write_par_var()**
Initialisierung des Ausgangs, init_write_par()	keine	initialisiert	initialisiert
Zwischenpufferung eines Ausgangs-datenpakets		X	X
Schreibroutine ter-miniert auch (springt zum Squenzer Programm zurück), wenn Daten nicht ausgegeben		X[1)]	X
zu früh gesendete Daten werden verworfen			X

1) wenn zuvor gesendete Daten ausgegeben sind.

(Da ein Datenpaket zwischengepuffert werden kann)

Aus der Tabelle geht hervor, daß folgende Fälle zu unterscheiden sind:

Haben Sie die Initialisierungsroutine **init_write_par()** aufgerufen, so kann ein Ausgangsdatenpaket zwischengepuffert werden. Dies gilt für alle vier Ausgaberoutinen gleichermaßen.

1. Fall:

Betrachten Sie nun die Ausgaberoutinen **write_par()** und **write_par_var()** (rechte Spalte). Im Gegensatz zu den beiden anderen Ausgaberoutinen springen diese Routinen sofort zum aufrufenden **Sequenzer** Programm zurück, wenn keine Daten ausgegeben werden können, daß heißt sie werden terminiert/beendet. Das **Sequenzer** Programm bearbeitet also unverzüglich den Befehl, der unmittelbar auf die Ausgaberoutine folgt. Außerdem werden zu früh gesendete Daten verworfen. Das heißt, die neuesten (zuletzt eingetroffenen) Daten werden verworfen, wenn die zuvor gesendeten Daten nicht ausgegeben werden können.

Der Unterschied zwischen **write_par()** und **write_par_var()** liegt in der Anzahl der Daten, die ausgegeben werden. Bei **write_par()** ist

es ein Datenpaket, bei **write_par_var** ist es immer nur ein Datenwert

2. Fall:

Im Gegesatz zu Fall 1 warten die Ausgaberoutinen **write()** und **write_var()** (mittlere Spalte) mit dem Senden der Daten, bis die „intelligente“ Ausgabeschnittstellen neue Daten aufnehmen kann, d.h. die zuvor gesendeten Daten ausgegeben worden sind. In diesem Fall werden keine Daten verworfen.

Der Unterschied zwischen **write()** und **write_var()** liegt wiederum in der Anzahl der Daten, die ausgegeben werden. Bei **write()** ist es ein Datenpaket, bei **write_var** ist es immer nur ein Datenwert

Das folgende Beispiel zeigt Einsatzmöglichkeiten der Ausgaberoutinen.

Beispiel*:*

```
/*Sequenzer-Programm zur Demonstration der
parallelen Schreibroutinen. Das Programm
protokolliert die Anzahl der erfolgreichen
und nicht erfolg-reichen Schreibversuche im
Array z[] und schickt das Ergebnis an den
Ausgang 1*/

HEADER h_x;
float z[2];

void start(PAR)
{
   int i;
   int can_write;
   float range = 1;

   /* Wertzuweisung Datenfeld Datenfeld mit
   0 initialisieren */

   loop(i, 0, 2)
   {
      z[i] = 0;
   }

   /* Datenheader einstellen */
   set_y0(h_x, 0);
   set_xtype(h_x, NIX);
   set_ytype(h_x, HISTOGRAMM);
   set_channel_count(h_x, 2);

   /* paralleles Schreiben initialisieren */
   init_write_par(1);
   while(1)
```

```
   {
      /* nicht zu schnell */
      wait(30);

      // überprüfen, ob das letzte Paket
      schon ausgegeben ist */

      test_write_par(1, can_write);
      if(can_write)
      {
         /* Anzahl der Erfolge beim Schreiben
         */
         z[0] = z[0] + 1;
         if(range < z[0]) range = z[0];
         /* y-Achsenskalierung anpassen */
         set_yrange(h_x, range * 1.1 + 1);
         /* Datenfeld z an Ausgang 1 ausgeben
         */
         write(1, z, 2, h_x);
      }
      else
      {
         /* Anzahl der Mißerfolge beim
         Schreiben */
         z[1] = z[1] + 1;
         if(range < z[1]) range = z[1];
      }
   }
   stop;
}
```

read

Syntax:

read(**->** int Eingangsbutton,
<- float yin[],
<-> int count,
<- HEADER hd);

Die Funktion **read()** liest vom Eingang mit der Nummer *Eingangsbutton* die **Anzahl *count* Daten** und trägt sie in die Feldvariable *yin*[] ein. Ist der Eingnag für asynchrones Lesen initialisiert worden, können Eingangsdaten durch Überschreiben verlorengehen!

Siehe auch **init_read_par().**

Die Funktion **read()** liest:

- höchstens die in *count* angegebene Anzahl von Werten. Kommen mehr Daten an, so werden diese verworfen. Der Wert von *cont* bleibt in diesem Fall unverändert.
- Kommen weniger Daten an, so werden nur die eingetroffenen Daten gelesen und die tatsächliche Anzahl der gelesenen Eingangsdaten wird in *count* zurückgegeben. Möchten Sie immer eine konstante Anzahl von Werten einlesen, müssen Sie in diesem Fall *count* nach dem Einlesen der Daten wieder auf die gewünschte Anzahl hochsetzen.
- Sind keine neuen Eingangsdaten angekommen, wartet die Funktion **read()** auf das Eintreffen neuer Eingangsdaten.

In der HEADER Variablen *hd* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen von den empfangenen Daten übernommen. Sollte nach einer Umrechnung eine Umskalierung notwendig werden, muß diese nach jedem Aufruf von **read()** mit den entsprechenden Befehlen ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. vorgenommen werden.

Beispiel:

```
float x, y[100];
int anzahl, Eingangsbutton;
HEADER hdx, hdy;

void berechne(PAR ,float x, float y[])
{
   ...
}
void start(PAR)
{
   Eingangsbutton = 2;
   anzahl = 100;
   /* lese einen x Wert */
   read_var(1, x, hdx);
   /* lese 100 Werte y von */Eingangsbutton
   2
   read(Eingangsbutton, y, anzahl, hdy);
   /* zu wenig Daten */
   if(anzahl < 100)
   stop;
   berechne(p, x, y);
   /* schreibe x */
   write(1, x, 1, hdx);
   Eingangsbutton = 3;
   /* schreibe y[] */
   write(Eingangsbutton, y, anzahl, hdy);
   stop;
}
```

In obigem Beispiel wird in die Variable *x* ein Wert gelesen, in das Datenfeld *y*[] werden 100 Werte gelesen. Sind nicht ausreichend Werte vorhanden, wird das **Sequenzer** Programm beendet, ansonsten wird eine Berechnungsfunktion aufgerufen.

read_par

Syntax:

read_par(**->** int Eingangsbutton,
<- float yin[],
<-> int count,
<- HEADER hd);

Die Funktion **read_par()** liest vom Eingang mit der Nummer *Eingangsbutton* die **Anzahl *count* Daten** in die Feldvariable *yin*[] über die parallele Leseroutine ein. Ist der Eingang für asynchrones Lesen initialisiert worden, können Eingangsdaten durch Überschreiben verlorengehen! Siehe auch **init_read_par().**

Die Funktion **read_par()** liest:

- höchstens die in *count* angegebene Anzahl von Werten. Kommen mehr Daten an, so werden diese verworfen. Der Wert von *count* bleibt in diesem Fall unverändert.
- Kommen weniger Daten an, so werden nur die eingetroffenen Daten gelesen und die tatsächliche Anzahl der gelesenen Eingangsdaten wird in *cont* zurückgegeben. Möchten Sie immer eine konstante Anzahl von Werten einlesen, müssen Sie in diesem Fall *count* nach dem Einlesen der Daten wieder auf die gewünschte Anzahl hochsetzen.
- Sind keine neuen Eingangsdaten angekommen, so wird der Wert von *count* auf 0 gesetzt, *yin*[] bleibt unverändert und die Einleseroutine terminiert.
  Es ist darauf zu achten, daß *count* vor dem Aufruf der Funktion **read_par()** einen gültigen Wert enthält. *count* muß vor dem Aufruf größer 0 und kleiner oder gleich der Größe des Datenfeldes *yin*[] sein
  (0 < *count* <= Größe von *yin*[]). Standen beim letzten Aufruf der Funktion keine Daten zur Verfügung, wurde *count* auf 0 gesetzt und muß deshalb vor einem erneuten Aufruf wieder auf den gewünschten Wert gesetzt werden.

Die parallele Leseroutine **read_par()** muß mit der Funktion **init_read_par()** initialisiert worden sein, damit der Aufruf wirksam wird. Andernfalls liefert **read_par()** keine Daten, d.h. der Wert von *count* wird auf 0 gesetzt.

In der HEADER Variable *hd* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen von den empfangenen Daten übernommen. Sollte nach einer Umrechnung eine Umskalierung notwendig werden, muß diese nach jedem Aufruf von **read_par()** mit den entsprechenden Befehlen - ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. - vorgenommen werden.

Beispiel: siehe Funktion **init_read_par().**

read_par_var

Syntax:

read_par_var(**->** int Eingangsbutton,
<- float yin,
<- HEADER hd);

Diese Funktion liest vom Eingang mit der Nummer *Eingangsbutton* genau **ein Datum** in die Variable *yin* über die parallele Leseroutine ein. Ist der Eingang für asynchrones Lesen initialisiert worden, können Eingangsdaten durch Überschreiben verlorengehen!

Siehe auch **init_read_par().**

Steht **kein neues** Datum zur Verfügung, bleibt der Wert von *yin* unverändert, und die Einleseroutine terminiert. Steht bereits ein Datum bereit, so wird dieses unmittelbar übernommen.

Die parallele Leseroutine **read_par_var()** muß mit der Funktion **init_read_par()** initialisiert worden sein, damit der Aufruf wirksam ist. Andernfalls liefert **read_par_var()** keine Daten, d.h. der Wert von *yin* wird auf 0 gesetzt.

In der HEADER Variablen *hd* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen von den empfangenen Daten übernommen. Sollte nach einer Umrechnung eine Umskalierung notwendig werden, muß diese nach jedem Aufruf von **read_par_var()** mit den entsprechenden Befehlen - ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. - vorgenommen werden.

Beispiel: siehe Funktion **init_read_par()**.

read_var

Syntax:

read_var(**->** int Eingangsbutton,
<- float yin,
<- HEADER hd);

Die Funktion **read_var()** liest vom Eingang mit der Nummer *Eingangsbutton* genau **ein Datum** in die Variable *yin* ein. Ist der Eingang für asynchrones Lesen initialisiert worden, können Eingangsdaten durch Überschreiben verlorengehen! Siehe auch **init_read_par().**

Stehen noch keine neuen Daten zur Verfügung, wird auf die Ankunft neuer Daten gewartet.

In der HEADER Variablen *hd* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen von den empfangenen Daten übernommen. Sollte nach einer Umrechnung eine Umskalierung notwendig werden, muß diese nach jedem Aufruf von **read_var()** mit den entsprechenden Befehlen - ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc.- vorgenommen werden.

Beispiel: siehe Funktion **read().**

test_read_par

Syntax:

test_read_par(**->** int Eingangsbutton,
<- int len);

Die Funktion **test_read_par()** überprüft, ob am Eingang mit der Nummer *Eingangsbutton* Daten zur Verarbeitung anliegen. Der Wert von *len* zeigt an, wieviele Daten zur Weiterverarbeitung bereitstehen. Sind keine Daten vorhanden, hat *len* den Wert 0. Voraussetzung für die Funktion **test_read_par()** ist, daß die Funktion **init_read_par()** zuvor für den Eingangskanal *kanal* aufgerufen wurde.

Die Funktion **test_read_par()** sollte nur in der synchronen Betriebsart verwendet werden, da in der asynchronen Betriebsart nach erfolgtem **test_read_par()** bereits neue Daten mit anderer Datenzahl anstehen können.

Beispiel: siehe Funktion **init_read_par()**.

wait_read_par_list

Syntax:

wait_read_par_list(**<->** int eingang,
-> int eingang_1,
-> int eingang_2,...,
-> int eingang_n,

-> int modus);

Die Funktion liefert aus einer Reihe von Eingängen denjenigen Eingang zurück, an dem Daten angekommen sind. Zu diesem Zweck wird der Funktion eine Parameterliste übergeben (*eingang_1, eingang_2, ..., eingang_n*), in der die zu überprüfenden Eingänge stehen.

Bei der Funktion **wait_read_par_list()** muß jeder Eingang zuvor mit **init_read_par()** initialisiert worden sein.

Im Parameter *eingang* wird die Nummer des Eingangs zurückgegeben, an dem Daten angekommen sind. Die Funktion kann in einem von zwei Modi verwendet werden, der durch den Parameter *modus* vorgegeben wird.

Sind an keinem der angegebenen Eingänge Daten empfangen worden, so wird mit:

- -1 als letzter Parameter auf die Ankunft von Daten gewartet,
- 0xFF als letzter Parameter sofort zurückgesprungen und als Ergebnis -1 zurückgegeben.

Die Eingänge werden in der Reihenfolge (Priorität) auf die Ankunft von Daten überprüft, in der sie in der Parameterliste stehen. Mit dem Übergabeparameter *eingang* kann angegeben werden, ab welchem Eingang die Liste der Eingänge abgefragt werden soll. Sind bereits an einem oder mehreren Eingängen Daten angekommen, so wird die Nummer des ersten Eingangs aus der Prioritätsliste in der Variablen *eingang* zurüchgegeben.

Wird die Variable *eingang* vor erneutem Aufruf der Funktion **wait_read_par_list()** nicht verändert, steht der dem zuletzt zurückgegebenen Eingang folgende Eingang an erster Stelle in der Prioritätsliste.

Dadurch ist gewährleistet, daß die Nummer von jedem Eingang zurückgegeben wird, sofern Daten angekommen sind („rotierende Priorität“).

Soll die Prioritätsliste stets die gleiche bleiben, so muß vor jedem Aufruf der Funktion **wait_read_par_list()** die Variable *eingang* auf den gleichen Wert gesetzt werden.

Beispiel:

```
float eingang = 1;
...
wait_read_par_list(eingang,  1, 2,  5, -1);
...
```

In diesem Beispiel wird auf die Ankunft von Daten von Eingang 1, 2 oder 5 gewartet. Die Nummer des Eingangs an dem Daten angekommen sind, wird in *eingang* zurückgegeben. Die Funktion wartet, bis von einem der angegebenen Eingänge Daten empfangen worden sind.

write

Syntax:

write(**->** int Ausgangsbutton,
-> float data[],
-> int len,
-> HEADER hdw);

Die Funktion **write()** dient der Ausgabe von Daten über den Ausgangskanal *Ausgangsbutton.* Die Daten werden im Daten-Feld *data*[] übergeben. Die Anzahl der Werte, die geschrieben werden sollen, wird in *len* übergeben. Das **Sequenzer** Programm wird erst dann fortgesetzt, wenn die Ausgabe beendet ist.

Ist der Ausgang für paralleles Schreiben mit der Funktion **init_write_par()** initialisiert worden, kann der Ausgang ein Datenpaket zwischenpuffern (siehe **init_write_par()**).

In diesem Fall terminiert die Ausgaberoutine, wenn die zuvor gesendeten Daten ausgegeben worden sind.

In der HEADER Variablen *hdw* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen der zu sendenden Daten übergeben. Eine Umskalierung muß mit den entsprechenden Befehlen ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. vorgenommen werden.

Beispiel:

```
float x;
HEADER h_x;
float z[20];

void start(PAR)
{
   int i;

   /* Wertzuweisung Datenfeld */
   loop(i, 0, 20)
   {
      z[i] = i * 2;
   }
   x = 5.43;
   /* Variable x an Ausgang 1 ausgeben */
   write_var(1, x, h_x);
   /* Abstand der Werte auf der  x-Achse */
   set_xdelta(h_x, 0.1);
   /* Datenfeld z an Ausgang 1 ausgeben */
   write(1, z, 10, h_x);
   /* 10 Daten */
```

```
   stop;
}
```

test_write_par

Syntax:

test_write_par(**->** int Ausgangsbutton,
<- int status);

Die Funktion **test_write_par()** überprüft, ob die parallele Schreibroutine am Ausgang *Ausgangsbutton* die Daten bereits ausgegeben hat. Der Rückgabewert von *status* zeigt den Zustand der Ausgabe an:

- *status = TRUE* - die parallele Schreibroutine hat die Daten bereits ausgegeben.
- *status = FALSE* - Daten sind noch nicht ausgegeben.

Voraussetzung für die Funktion **test_write_par()** ist, daß die Funktion **init_write_par()** zuvor für den Ausgangs-kanal *kanal* aufgerufen wurde.

Beispiel: siehe Funktion **init_write_par()**.

write_cmd

Syntax:

write_cmd(**->** int Ausgangsbutton);

write_cmd() gibt am Ausgang mit der Nummer *Ausgangsbutton* die zuvor mit **start_cmd()** definierte Blockbefehlsliste aus. Diese Funktion dient der Konfiguration verschiedener HYDRA-Blöcke zur Laufzeit, d.h. nach dem Start-Befehl. Die Befehle müssen an die entsprechenden **St**-Eingänge der HYDRA-Blöcke gesendet werden.

Die Voreinstellung der Blockbefehle geschieht in der Eingabedialogbox. Werden die Werte und Einstellungen der Eingabedialogbox nicht durch **write_cmd()** bzw. **start_cmd()** manipuliert oder außer Kraft gesetzt, so behalten sie ihren Wert und ihre Gültigkeit (siehe auch **start_cmd()**).

Beispiel:

```
float x, y = 2;
void start(PAR)
{
   x = 5;
   y = 1;
   /* Initialisierung */
   start_cmd()
```

```
   {
      ABTASTRATE = x;
      TRIGGER_START = FLANKE;
      TRIGGER_START_FLANKE = STEIGEND;
      TRIGGER_START_PEGEL_WERT = y + 0.2;
      START;
   }
   /* Sende Kommando über Ausgang 1 */
   write_cmd(1);
   stop;
}
```

Nicht jeder Blockbefehl hat für jeden HYDRA-Block Gültigkeit. Generell ist jeder HYDRA-Block mit einem **St**-Eingang in der Lage, zumindest den START- und den STOP-Befehl zu empfangen. Für besondere Implementationen der **Sequenzer** Befehle ist die Dokumentation des jeweiligen HYDRA-Blocks heranzuziehen (siehe auch Blockbefehle).

write_par

Syntax:

write_par(**->** int Ausgangsbutton,
-> float data[],
-> int len,
-> HEADER hdw);

Die Funktion **write_par()** dient der Ausgabe von Daten mit der parallelen Schreibroutine über den Ausgangskanal *Ausgangsbutton*. Die Daten werden im Datenfeld *data*[] übergeben. Die Anzahl der Werte, die geschrieben werden sollen, wird in *len* festgelegt.

Sind die Daten des vorangegangenen **write_par()** Befehls noch nicht vollständig ausgegeben worden, so werden die neuen Daten verworfen, d.h. sie gelangen nicht zur Ausgabe, und die Ausgaberoutine terminiert.

Voraussetzung für die parallele Ausgabe ist, daß die Funktion **init_write_par()** für den Ausgabekanal *Ausgangsbutton* bereits einmal aufgerufen wurde.

In der HEADER Variablen *hdw* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen der zu sendenden Daten übergeben. Eine Umskalierung muß mit den entsprechenden Befehlen ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. vorgenommen werden.

Beispiel: siehe Funktion **init_write_par()**.

write_par_var

Syntax:

write_par_var(-> int Ausgangsbutton,
-> float data,
-> HEADER hdw);

Die Funktion **write_par_var()** gibt genau **ein Datum** mit der parallelen Schreibroutine über den Ausgangskanal *Ausgangsbutton* aus. Das Datum wird in der Variable *data* übergeben.

Sind die Daten des vorangegangenen **write_par()/ write_par_var()** Befehls noch nicht vollständig ausgegeben worden, so wird das neue Datum verworfen, d.h. es gelangt nicht zur Ausgabe, und die Ausgaberoutine terminiert.

Voraussetzung für die parallele Ausgabe ist, daß die Funktion **init_write_par()** für den Ausgabekanal *Ausgangsbutton* bereits einmal aufgerufen wurde.

In der HEADER Variablen *hdw* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen der zu sendenden Daten übergeben. Eine Umskalierung muß mit den entsprechenden Befehlen ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. vorgenommen werden.

Beispiel: siehe Funktion **init_write_par().**

write_var

Syntax:

write_var(**->** int Ausgangsbutton,
-> float data,
-> HEADER hdw);

Die Funktion **write_var()** dient der Ausgabe **eines Datums** über den Ausgangskanal *Ausgangsbutton*. Das Datum wird in der Variable *data* übergeben. Das **Sequenzer** Programm wird erst dann fortgesetzt, wenn die Ausgabe beendet ist.

Ist der Ausgang für paralleles Schreiben mit der Funktion **init_write_par()** initialisiert worden, kann der Ausgang ein Datum zwischengepuffert (siehe auch **init_write_par()**).

In diesem Fall terminiert die Ausgaberoutine wenn die zuvor gesendeten Daten ausgegeben worden sind.

In der HEADER Variablen *hdw* wird die Skalierung sowie die Einheiten der x- bzw. y-Achsen der zu sendenden Daten übergeben. Eine Umskalierung muß mit den entsprechenden Befehlen ***set_x0(), set_y0(), set_xtype(), set_ytype()*** etc. vorgenommen werden.

Beispiel: siehe Funktion **write()**.

STD-IO-Funktionen

Mit den folgenden Funktionen können Sie Meldungen, Datenwerte und Kommentare im Fenster **STD-IO** ausgeben lassen.

Dazu müssen Sie in der **Sequenzer** Dialogbox die Option **Standart Ausgabe aktivieren** angewählt haben
(s. Kapitel 6.1).

Das **Hydra Control Programm** wird in Kapitel 7 ausführlich beschrieben.

array_puts

Syntax:

array_puts(**->** float xarray[],
-> int len);

Diese Funktion gibt die im Feld *xarray*[] gespeicherten ASCII-Werte im Fenster **STD_IO** des **HYDRA Control Programms** aus. Jeder *float*-Wert wird als ein ASCII-Zeichen interpretiert. Der Wert von *len* gibt die Anzahl der ASCII-Zeichen im Feld *xarray*[] an.

Beispiel:

```
void start(PAR)
{
   int len = 40;
   float str[40];
   float x = 1.23;
   float y = 4.56;

   /* ASCII-Zeichen in Feld schreiben */
   float_printf(str, len, "%.7g : %.7g", x,
   y);
   /* Ausgabe in STD-IO */
   puts("x : y =");
   /* „x␣:␣Y␣=“ */
   array_puts(str, len);
   /* „1.23␣:␣4.56“ */
}
```

debug

Syntax:

debug(-> int time);

debug() schaltet den Debugmodus des **HYDRA Sequenzer** Programms an oder aus. Ist der Wert der Variablen *time > 0*, wird der Debugmodus aktiviert, ansonsten deaktiviert. Mit dem Wert von *time* läßt sich das Zeitintervall in Millisekunden bestimmen, in dem der **Sequenzer** jeden Befehl ausführt.

Im Debugmodus gibt der **HYDRA Sequenzer** Block über das **HYDRA Control Programm** im Fenster **STD-IO** die Nummer der gerade bearbeiteten Programmzeile aus. Damit lassen sich etwaige Programmfehler, wie Unendlichschleifen oder stockende Ein-/Ausgaben, aufspüren.

Beispiel*:*

```
/* Deklarationen */
float x, y;
HEADER hd;

/* hier beginnt das Unterprogramm */
void Addiere(PAR, float x, float y)
{
   y = x + y;
}

void start(PAR)
{
   x = 3.5;
   y = 4.5;
   /* Debugmodus eingeschaltet 10 ms */
   debug(10);
   Addiere(p, x, y);

   /* Debugmodus ausgeschaltet */
   debug(0);

   /* y hat nun den Wert 8.0 */
   write(1, y, 1, hd);

   /* Terminierung des Sequenzer-Programms
   */
   stop;
}
```

err_printf

Syntax:

err_printf(**<-** const char *format[,
-> arg1,
-> arg2,
...]);

Diese Funktion entspricht der Funktion **printf()**. Der einzige Unterschied zwischen diesen beiden Funktionen besteht darin, daß **err_printf()** in der Dialogbox bei **Standart Ausgabe aktivieren** nicht abgeschaltet werden kann (s. Kapitel 6, Konfiguration).

Beispiel: siehe die Funktion **printf()**.

err_puts

Syntax:

err_puts(**->** const char *string);

Diese Funktion entspricht der Funktion **puts()**. Der einzige Unterschied zwischen diesen beiden Funktionen besteht darin, daß **err_puts()** in der Dialogbox bei **Standart Ausgabe aktivieren** nicht abgeschaltet werden kann
(s. Kapitel 6, Konfiguration).

Beispiel: siehe die Funktion **puts()**.

printf

Syntax:

printf(**<-** const char *format[,
-> arg1,
-> arg2,
...]);

Die Funktion **printf()** erzeugt eine formatierte Ausgabe im **STD-IO** Fenster des **HYDRA Control Programms**. Sie dient dazu, numerische Werte in Zeichendarstellung umzuwandeln, also zur Ausgabe von float- und integer-Werten. Die Ausgabe mit **printf()** ist

in der Dialogbox des **Sequenzers** abschaltbar (s. Kapitel 6.1, Konfiguration).

Die Zeichenkette *format* enthält zwei Arten von Objekten:

- gewöhnliche ASCII-Zeichen, die einfach ausgegeben werden,
- Format-Elemente, die jeweils die Umwandlung und Ausgabe des folgenden Arguments *argx* veranlassen. Jedes Format-Element beginnt mit dem Zeichen „**%**“. Anschließend folgen die Formatanweisungen.

Es stehen folgende Formatanweisungen zur Verfügung:

- **%md** gibt das *int*-Argument auf einer Feldbreite mit **m** Stellen aus. Beispiel: Die Zahl 53 ergibt mit %6d die Ausgabe ' 53'.
- **%m.ne** gibt das *float*-Argument in der Exponentenschreibweise auf einer Feldbreite mit **m** Stellen und **n** Nachkommastellen aus. Die Zahl 53 ergibt mit %8.2e die Ausgabe '0.53+E02'
- **%m.nf** gibt das *float*-Argument mit **n** Nachkommastellen auf einer Feldbreite mit **m** Stellen aus. Die Zahl 53 ergibt mit %4.1f die Ausgabe '53.0'
- **%m.ng** gibt das *float*-Argument entweder im %f-Format oder im %e-Format aus, je nachdem welches kürzer ist. Nicht signifikante Nullziffern werden dabei nicht ausgegeben.

Ist in der Formatzeichenkette der Wert für **m** negativ, so wird das Argument im Feld links ausgerichtet. Die Zeichenkette *format* kann zudem noch Formatierungs-parameter zur Zeilenformatierung beinhalten. Diese werden mit dem Zeichen **** eingeleitet. Es sind folgende Formatbefehle möglich:

- **\a** - Klingelzeichen (bell),
- **\b** - Rückzeichen (backspace),
- **\f -** Seitenvorschub (form feed),
- **\n** - Zeilenvorschub (line feed),
- **\r -** Zeilenrücklauf (carriage return),
- **\t -** horizontaler Tabulator (htab),
- **\v -** vertikaler Tabulator (vtab),
- **\\ -** das \ Zeichen (back slash),
- **\xhh** - hexadezimales Zeichen mit Wert hh (z.B. \x0d).

Beispiel:

```
printf("Eine Zeile\n");
printf("Eine Seite\f");
printf("Tab1\tTab2\tTab3\n");
```

```
printf("Wert für x, y: %5.2f, %5.2f\n", x,
y);
```

puts

Syntax:

puts(**->** const char *string);

Die Funktion **puts()** gibt im **STD-IO** Fenster des **HYDRA Control Programms** die Zeichenkette *string* aus. Die Zeichenkette muß mit zwei Hochkommazeichen (z.B: "das ist eine Zeichenkette") eingeschlossen werden. Die Zeilenformatierung geschieht mit den in der Funktion **printf()** beschriebenen Zeichen zur Zeilenformatierung.

Die Ausgabe mit der Funktion **puts()** ist in der Dialogbox des **Sequenzers** abschaltbar (s. Kapitel 6.1, Konfiguration).

Beispiel:

```
puts("Eine Zeile\n");
puts("Eine Seite\f");
puts("Tab1\tTab2\tTab3\n");
```

show_float, show_hex

Syntax:

show_float(**->** float value);

show_hex(**->** float value);

Diese Funktionen geben im **STD-IO** Fenster des **HYDRA Control Programms** die Werte von *value* aus. Die Funktion **show_float()** formatiert den Wert von *value* ins Gleitkommaformat, die Funktion **show_hex()** in das hexadezimale Format.

Beispiel:

```
void start(PAR)
{
   float x, y;
   ...
   x = 23.5;
   y = 0x3F;
   show_float(x);
   /* ergibt '23.5' */
   show_hex(y);
   /* ergibt '3F' */
}
```

show_header

Syntax:

show_header(**->** HEADER hd);

show_header() gibt im **STD-IO** Fenster des **HYDRA Control Programms** den Datenheader *hd* aus.

Beispiel:

```
HEADER hd;

void start(PAR)
{
   ...
   init_header(hd);
   ....
   /* gibt Datenheader aus */
   show_header(hd);
   ...
}
```

Als Ergebnis erhalten Sie folgende Werte:

Header <hd>

Anzahl = 1
Kanaele= 1
Messende = FALSE
xdelta = 1
x0 = 0
xtype = MILLISEKUNDEN
ytype = VOLT.

Blockbefehle

Für die Steuerung zahlreicher Hydra-Blöcke mit den Hydra Sequenzer Funktionen **start_cmd()** und **write_cmd()** stehen Ihnen die in diesem Kapitel beschriebenen Blockbefehle zur Verfügung. Sie werden in alphabetischer Reihenfolge vorgestellt.

Genauere Erläuterung der beiden Funktionen **start_cmd()** und **write_cmd()** entnehmen Sie bitte dem Kapitel Standartfunktionen.

Nach dem Start gelten zunächst generell die Eingaben und Einstellungen in den Dialogboxen der Blöcke. Erhalten Blöcke über den ST-Button neue Parameterwerte oder neue Einstellungen, so sind die entsprechenden Eingaben der Dialogboxen ohne Bedeutung.

Allgemein gilt, daß neue, über den ST-Button gesendete Eingaben erst nach einem erneuten START-Befehl gültig werden. (Ausnahme: REGLER)

Die Blockbefehle teilen sich auf in Parametrier- und Steuerbefehle.

ABTASTLUECKEN

Syntax:

ABTASTLUECKEN = EIN/AUS;

Gültig für:

ADC S002, IO PORT H109/H110, ADC H004, ADC CARD

Mit diesem Befehl stellen Sie Abtastlücken zwischen Messungen zulassen in der Dialogbox Optionen ein bzw. aus. Sie steuern also ob zeitkontinuierlich zwischen zwei Messungen abgetastet wird oder nicht. Wählen Sie den Befehl ABTASTLUECKEN EIN, so sind zeitliche Sprünge zwischen zwei Messungen möglich, andernfalls nicht.

Beispiel: siehe **ANZAHL_MESSUNGEN**

ABTASTRATE

Syntax:

ABTASTRATE = (float)t;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, Drehgeber H409, SK PORT S100, SK PORT S102, Zähler CARD

Verwenden Sie diesen Blockbefehl zur Einstellung der Abtastrate des angeschlossenen ADC-Blocks. Der Wert für t ist in Millisekunden anzugeben. Die Einstellung wird nur dann akzeptiert, wenn der Block vorher mit einem **STOP**-Befehl deaktiviert wurde

Beispiel:

```
/* hält den ADC an, stellt die Abtastrate auf
5.5 ms ein und startet den ADC wieder */
adc_h109 = 3
/* Ausgang 3 des Sequenzerblocks ist mit dem
ST-Button des ADC H109 verbunden */

start_cmd()
{
   STOP;
   ABTASTRATE = 5.5;
   START;
}
write_cmd(adc_h109);
```

ANZAHL_MESSUNGEN

Syntax:

ANZAHL_MESSUNGEN = (int)n;

Gültig für:

ADC S002, ADC H109/H110, ADC H004, ADC CARD

Mit diesem Blockbefehl stellen Sie die Anzahl der Messungen ein. Der Wert von *n* gibt an, wie oft auf einen Triggerstart gewartet werden soll. Eine Messung beinhaltet alle Daten zwischen einem Triggerstart- und einem Triggerstopereignis. Hat *n* den Wert 0, so ist die Anzahl der Messungen unbegrenzt. Der mit diesem Blockbefehl

eingestellte Wert entspricht mit dem Menüpunkt **Anzahl der Messungen** im Auswahlfeld **Trigger/Start** des angeschlossenen A/D Wandlerblockes.

Beispiel:

```
/* läßt für die folgenden Messungen Abtastlücken
zu */
adc_h109 = 3;
start_cmd()
{
     STOP;
     ABTASTLUECKEN=EIN;
     /* Abtastrate 5.5 msec */
     ABTASTRATE=5.5;
     /* 10 Messungen START; */
     ANZAHL_MESSUNGEN=10;
}
/* Übertrage Befehle */
write_cmd(adc_h109);
```

AUSGABE_ISTWERT

Syntax:

AUSGABE_ISTWERT

Gültig für:

REGLER, GRENZWERT

REGLER*:*

Dieser Befehl bewirkt die Ausgabe der aktuellen Blockgrößen am Ausgang Ii. Ausgegeben wird ein Datenfeld, welches die folgenden 6 Größen enthält:

data[0] **:** Aktueller Istwert des äußeren REGLER

data[1] **:** Aktueller Istwert des inneren REGLER

data[2] **:** Aktueller Sollwert des äußeren REGLER. Bei **SEQ_HALTE_SOLLWERT = AN** ist dieser Wert der eingefrorene Sollwert.

Data[3] **:** Aktueller Sollwert des inneren REGLER

data[4] **:** Aktueller Wert der Ausgangsgröße am Ausgang **A** des REGLER

data[5] **:** Aktueller Sollwert des äußeren REGLER. Bei **SEQ_HALTE_SOLLWERT = AN** ist dieser Wert der externe am Eingang **So** vorgegebene und

verworfene Sollwert. Sind noch keine ADC-Daten am REGLER angekommen, ist dieser Wert 10^{20}. Damit kann festgestellt werden, ob der ADC-Block bereits Daten an den REGLER übertragen hat.

Grenzwertblock:

Dieser Blockbefehl bewirkt die Ausgabe des aktuellen Zustandes der Ausgänge am Ausgang **Ii**. das ausgegebene Datenfeld umfasst folgende 32 Größen:

data[0] :	1: 1. Grenzwert überschritten, 0: nicht überschritten
data[15] :	1: 16. Grenzwert überschritten 0: nicht überschritten
data[16] :	Zuletzt ausgegebener Wert (Wert der Reaktion) an Ausgang R1
data[31] :	Zuletzt ausgegebener Wert (Wert der Reaktion) an Ausgang R16

Sind noch keine Grenzwertüberschreitungen aufgetreten, dann haben data [0] bis data[31] den Wert 0.

AUSGABERATE

Syntax:

AUSGABERATE = (float)t;

Gültig für:

DAC 101, DAC H004

Durch Aufruf dieses Blockbefehls stellen Sie die Datenrate des DAC Blocks ein. Der Wert für *t* ist in Millisekunden anzugeben. Die Einstellung wird nur dann akzeptiert, wenn der Block vorher mit einem **STOP**-Befehl deaktiviert wurde. Dieser Wert ist mit dem Eingabefeld **Ausgaberate** in der Dialogbox identisch.

BLOCKLAENGE

Syntax:

BLOCKLAENGE = (int)b;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002,IO PORT H407, Drehgeber H409, H409 Pulsbreite, SK PORT S100, SK PORT S102, Zähler CARD

Mit dem Befehl **BLOCKLAENGE** stellen Sie die Datenpaketlänge des HYDRA-Blocks ein. Der Wert für *b* bestimmt die Anzahl der Daten, die bei einem Datentransfer auf einmal übertragen werden. Für nachfolgende FFT-Blöcke sollte dieser Wert der Auflösung der FFT entsprechen. Diese Größe können dem Wert im Eingabefeld **Maximale Datenblocklänge bei der Übertragung** in der Eingabemaske **Optionen** der angeschlossenen A/D-Wandlerblöcke eingeben.

DATENZAHL

Syntax:

DATENZAHL = (int)n;

Gültig für:

ADC H109/H110, ADC S002, ADC H004, ADC CARD

Mit diesem Befehl stellen Sie die Anzahl der Werte ein, die bis zum Ende einer Messung aufgenommen werden sollen. Die Datenzahl wird wirksam, wenn die Bedingung für den Triggerstop auf **TRIGGER_STOP** = **DATENZAHL** gesetzt ist (siehe Beispiel). Dieser Wert entspricht der **DATENZAHL** die Sie auch bei **TRIGGER/STOP** in der Dialogbox einstellen können. Als Synonym können Sie auch den Befehl **TRIGGER_STOP_WERTE** verwenden.

Beispiel:

```
/* setzt die Bedingung für Trigger Stop auf
DATENZAHL, die Datenzahl auf 200, d.h. das
Messende nach 200 Abtast-werten, den Pretrigger
auf 0 Werte */
```

```
start_cmd()
{
   TRIGGER_STOP = DATENZAHL;
   DATENZAHL = 200;
   PRETRIGGER = 0;
}
write_cmd(1);
```

DITHER

Syntax:

DITHER = EIN/AUS;

Gültig für:

REGLER

Mit diesem Befehl schalten Sie das Dither-Signal ein bzw. aus. Im Beispiel wird die Ditheramplitude auf 1V gesetzt. Der Vorzeichenwechsel erfolgt nach 4 **REGLER**. Das ergibt bei einer **REGLER** von 1 kHz eine Ditherfrequenz von 125 Hz.

Beispiel: **start_cmd()**

```
{
   DITHER_AMPLITUDE = 1;
   DITHER_N = 4;
   DITHER = EIN;
{
write_cmd(2)
```

DITHER_AMPLITUDE

Syntax:

DITHER_AMPLITUDE = (float)a

Gültig für:

REGLER

Legen Sie durch Aufruf dieses Befehls die Amplitude des Dither-Signals fest. Der Wert von a wird in Volt angegeben. Das Dithersignal wird zur begrenzten Stellgröße hinzuaddiert. Die Stellgröße wird auf den Bereich zwischen **STELLGRÖSSE_MIN** und **STELLGRÖSSE_MAX** begrenzt. Danach wird das Dithersignal

zur Stellgröße addiert. Es ist daher sinnvoll die Werte für **STELLGRÖSSE_MIN** und **STELLGRÖSSE_MAX** und die **DITHER_AMPLITUDE** so zu wählen, daß die Summe nicht über den Ausgabebereich (± 10V) des DA-Wandlers hinausgeht.

Der Wert der **DITHER_AMPLITUDE** entspricht dem Wert im Eingabefeld AMPLITUDE in der Dialogbox Dither.

Beispiel: Siehe **DITHER**

DITHER_N

Syntax:

DITHER_N = (int)n

Gültig für:

REGLER

Bestimmen Sie mit diesem Befehl die Taktrate des Dither-Signals. Das Vorzeichen des Dithersignals wechselt nach n Ausgaben (n REGLER). Dieser Wert entspricht der Einstellung n in der Dialogbox Dither.

Beispiel: Siehe **DITHER**

FAKTOR

Syntax:

FAKTOR (float)f;

Gültig für:

ADC H004, H407, DAC H207,REGLER

Bestimmen Sie mit diesem Befehl den Gewichtungsfaktor des folgenden Kanals. Verwenden Sie diesen Befehl um die Gewichtungsfaktoren zu Bestimmung des Istwertes aus mwehreren Eingangsgrößen einzustellen. Sämmtliche Gewichtungsfaktoren müssen mit einem write_cmd()-Befehl angegeben werden. Die Werte für **FAKTOR** entsprechen den Eingabefeldern a1 bis a4 der Dialogbox Istwert_Arithmetik.

Für REGLER:

Der Istwert berechnet sich aus:

Istwert = Offset + Faktor 1 * Kanal [Kanalnummer 1] +
Istwert = Offset + Faktor 2 * Kanal [Kanalnummer 2] +
Istwert = Offset + Faktor 3 * Kanal [Kanalnummer 3] +
Istwert = Offset + Faktor 4 * Kanal [Kanalnummer 4]

Der Istwert kann also aus maximal 4 Eingangsgrößen berechnet werden. Der erste Faktor eines write_cmd ()-Befehls ist a1, der zweite a2 usw.

Beispiel: Vom Eingang E werden 3 Kanäle empfangen. Davon sollen der Kanal 1 und 3 nach folgender Formel zur Istwertbestimmung verwendet werden:

Istwert = 2.5 + 1.3 * Kanal [1] + 2.5 * Kanal [3]

```
start_cmd()
{
   OFFSET = 2.5;
   KANAL_NUMMER = 1 ;
   FAKTOR = 1.3;
   KANAL_NUMMER = 3;
   FAKTOR = 2.5 ;
}
write_cmd (1);
```

Für ADC H004:

Mit diesem Befehl können Sie den Gewichtungsfktor des mit **KANAL_NUMMER** zuvor ausgewählten Kanals einstellen. Bis zu 16 Kanälen, bei differentieller Messung könne eingestellt werden.

Die Werte für f entsprechen den Eingaben in der Dialogbox.

Beispiel:

```
start_cmd ()
{
   KANAL_NUMMER       = 2;
   OFFSET             = 3;
   FAKTOR             = 1.5;
   KANAL_NUMMER       = 5;
   OFFSET             = 0.8;
   FAKTOR             = 2;
   START;
}
write_cmd (3)
```

FEHLER

Syntax:

FEHLER = EIN/AUS;

Gültig für:

REGLER

Starten bzw. stoppen Sie mit diesem Befehl die Fehlerbehandlungsroutine. Die Einstellung der gewünschten Reaktion im Fehlerfall erfolgt in der Dialogbox. Zur Fehlerbehandlungsroutine siehe Bescfhreibung des REGLER.

FEHLER_DATENSATZLUECKE

Syntax:

FEHLER_DATENSATZLUECKE = EIN/AUS;

Gültig für:

DAC H004

Mit diesem Befehl aktivieren bzw. deaktivieren Sie die Fehlermeldung *"Ausgabelücke zwischen den Datenpaketen"*. Ist diese Fehlermeldung aktiviert, erfolgt automatisch ein Deaktivieren der Fehlermeldung *"Zu früher Empfang des START-Befehles"*. Grund: Die **START** Befehle für weitere Datensätze müssen erfolgen, bevor die Ausgabe des letzten Datenpaketes beendet wurde, damit ein lückenloses Aneinanderfügen der Daten möglich ist. Das wiederum würde die Fehlermeldung "Zu früher Empfang des START_Befehles" auslösen. Deshalb wird diese Fehlermeldung deaktiviert.

Der Befehl entspricht der Einstellung „Abtastlücken zulasen“ im Dialogfeld „Optionen“.

FEHLER_FRUEHSTART

Syntax:

FEHLER_FRUEHSTART = EIN/AUS;

Gültig für:

DAC H004, dort auskommentiert mm 13.6.97

Mit diesem Blockbefehl aktivieren bzw. deaktivieren Sie die Fehlermeldung "Zu früher Empfang des START_Befehls" Siehe auch **FEHLER_DATENSATZLUECKE**.

FILTER

Syntax:

FILTER = EIN/AUS;

Gültig für:

ADC H109/H110

Dieser Blockbefehl aktiviert bzw. deaktiviert das Anti-Aliasing Filter des Ausgewählten A/D-Wandler. Sie müssen den ADC-Kanal, an dem der Befehl wirksam wird, vorher mit dem Befehl **KANAL_NUMMER** setzen (siehe Beispiel).

Dieser Befehl entspricht der Einstellung in der Dialogbox.

Beispiel:

```
/* setzt den ADC-Kanal für die folgenden Befehle
auf 3, schaltet den Kanal 3 an, aktiviert Anti-
Aliasing Filter auf Kanal 3 */
start_cmd()
{
   KANAL_NUMMER = 3;
   KANAL = EIN;
   FILTER = EIN;
}
write_cmd(1);
```

FILTER_ECKFREQUENZ

Syntax:

FILTER_ECKFREQUENZ = (float)f;

Gültig für:

ADC H109/H110

Mit diesem Befehl stellen Sie die Frequenz des Anti-Aliasing Filters der angeschlossenen A/D-Wandler ein. Die Frequenz *f* ist in Hertz anzugeben. Die **FILTER_ECKFREQUENZ** ist für alle ADC-Kanäle gleich (siehe Beispiel).

Beispiel:

```
/* setzt den ADC-Kanal für die folgenden Befehle
auf 2, schaltet den Kanal 2 an, aktiviert den
```

```
Anti-Aliasing Filter auf Kanal 2 und stellt die
Filtereckfrequenz aller Kanäle  auf 300 Hertz */

start_cmd()
{
   KANAL_NUMMER = 2;
   KANAL = EIN;
   FILTER = EIN;
   FILTER_ECKFREQUENZ = 300; /* für alle
   Kanäle */
}
write_cmd(3);
```

HALTE_SOLLWERT

Syntax:

HALTE_SOLLWERT = EIN/AUS;

Gültig für:

REGLER.

Der aktuelle Sollwert wird durch Aufruf dieses Befehls eingefroren.Es werden weiterhin Daten am Eingang SO abgenommen. Diese Daten werden verworfen. Wird ein Sollwert >= 10E20 empfangen, wird das Halten des Sollwerts automatisch deaktiviert.

KANAL

Syntax:

KANAL = EIN/AUS;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, DAC H004, IO PORT H407, H409 Pulsbreite, SK PORT S100, SK PORT S102

Aktiviert bzw. deaktivieren Sie einen Eingangskanal des angeschlossenen A/D-Wandlers durch Aufruf dieses Blockbefehl. Sie müssen den ADC-Kanal, an welchem der Befehl wirksam wird, zuvor mit dem Befehl **KANAL_NUMMER** setzen (siehe Beispiel). Um mehrere Kanäle gleichzeitig ein-oder auszuschalten, könne Sie den Befehl **KANALMUSTER** verwenden.

Beispiel*:*

```
/* setzt den ADC-Kanal für die folgenden
Befehle auf 3, schaltet den Kanal 3 ein, */

start_cmd()
{
   KANAL_NUMMER = 3;
   KANAL = EIN;
   write_cmd(2);
}
```

KANALMUSTER

Syntax:

KANALMUSTER = (int)m;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, DAC H004, IO PORT H407, H409 Pulsbreite, SK PORT S100, SK PORT S102

Mit diesem Blockbefehl wählen Sie die Eingangskanäle des angeschlossenen ADC Blockes aus. Der Wert von *m* ist als binäres Muster anzusehen, wobei für jede Stelle ein Eingangskanal steht, welche die Werte "0" und "1" annehmen kann. Die niederwertigste Stelle ist immer Kanal 1. Die Anzahl der Stellen muß der Anzahl der

Eingangskanäle entsprechen. Eine "1" aktiviert den zugehörigen Eingangskanal, eine "0" deaktiviert ihn. Die Einstellung wird nur dann akzeptiert, wenn Sie den Block vorher mit einem **STOP**-Befehl deaktiviert haben. Um nur einen Kanal auf einmal ein- oder auszuschalten können Sie auch den Befehl Kanal verwenden.

Beispiel:

```
start_cmd()
{
   /* Taste Kanal 1+2 ab, Kanal 3+4 */
   STOP;
   /* deaktiviert */

   KANALMUSTER = 0011;
   START;
}
write_cmd(2);

start_cmd()
{
   /* taste Kanal 3+4 ab,Kanal 1+2 */
   STOP ;
   /* deaktiviert */

   KANALMUSTER = 1100;
   START;
}
write_cmd(2);
```

KANALNUMMER

Syntax:

KANALNUMMER = (int) n;

Gültig für

ADC CARD, ADC H004, ADC H109, ADC S002, DAC H004, DAC H207, IO PORT H407, H409 Pulsbreite, REGLER, SK PORT S100, SK PORT S102

Mit diesem Befehl legen Sie die Kanalnummer für die folgenden Befehle fest. Alle nachfolgenden kanalabhängigen Befehle (z.B. **KANAL** = EIN/AUS, **FILTER** EIN/AUS) wirken dann nur auf diesen Kanal.

LETZTER_PARAMETER

Syntax:

LETZTER_PARAMETER;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, DAC H004, IO PORT H407, Drehgeber H409, H409 Pulsbreite, SK PORT S100, SK PORT S102, Zähler CARD

Dieser Befehl bewirkt die Initialisierung der Ausgabe mit den zuvor angegebenen Parametern. Weitere *write_cmd()* Befehle werden erst dann gelesen, wenn der D/A-Block den **START** Befehl sofort ausführen kann.

Der Blockbefehl dient zum einen der Synchronisation des Sequenzers mit dem DA-Wandler und zum anderen der Synchronisation mehrerer Blockbefehle innerhalb einer Übertragung von Blockbefehlen.

Der Blockbefehl **LETZTER_PARAMETER** bewirkt, daß der nachfolgende Blockbefehl erst dann ausgeführt wird, wenn die mit dem vorangegangenen **START**Befehl gestartete Verabreitung von daten beendet ist.

Beispiel: Drei Datensätze sollen lückenlos aneinandergefügt werden.

Datensatz 1: Sinus, periodischer Ausgabe 100 mal

Datensatz 2: Rechteck, periodischer Ausgabe 50 mal

Datensatz 3: Dreieck, periodischer Ausgabe 150 mal

Beispiel 1:

```
start_cmd()
{
   PERIODISCH = EIN; /* periodische Ausgabe
   aktivieren */
   PERIODEN_ZAHL = 100;  /* Periodenzahl für
   datensatz */
   LETZTER_PARAMETER;
}
write _cmd (1)
.......

start _cmd()
{
   START;  /* Satrt für Ausgabe des
   Datensatzes 100 mal */
```

```
}
write_cmd (1);
```

Beispiel 2:

```
start_cmd()
{
   PERIODISCH = EIN; /* periodische Ausgabe
   aktivieren */

   PERIODEN_ANZAHL = 100; /* Periodenzahl
   für ersten Datensatz */
   START; /* Start für Ausgabe ersten
   Datensatz 100 mal */

   PERIODEN_ANZAHL = 50; /* Periodenzahl für
   zweiten Datensatz */
   LETZTER PARAMETER;
   START; /* Start für Ausgabe zweiter
   Datensatz 50 mal */

   PERIODEN_ANZAHL = 150; /* Periodenzahl
   für dritten Datensatz */
   LETZTER _PARAMETER;
   START; /* Start für Ausgabe dritter
   Datensatz 150 mal */
}
write_cmd (1);
```

In **Beispiel 1** erfolgen die Befehle **LETZTER_PARAMETER** und **START** mit zwei start_cmd(), write_cmd()-Aufrufen. Mit dem ersten start_cmd()/ write_cmd() Aufruf wird die Initialisierung des DAC-Blocks durchgeführt. Der Sequenzer kann den **START** Befehl erst dann senden, wenn die Initialisierung des DIA_Blockes abgeschlossen ist. Das heißt, der Sequenzer wird so lange vom DIA-Block angehalten und somit zeitlich synchronisiert. Der im zweiten start_cmd()/ write_cmd()-Anfang gesendete **START** Befehl bewirkt den sofortigen Beginn der Datenausgabe. Damit ist dem Sequenzer bekannt, wann die Datenausgabe beginnt.

Beispiel 2 zeigt wie ein "Blockprogramm" erstellt werden kann. Innerhalb eines start_cmd()/write_cmd()-Aufrufs werden mehrere Initialisierungsbefehle (auch gleiche) getrennt durch **LETZTER_PARAMETER** gesendet.

Die Initialisierung wird bis zum Befehl **LETZTER_PARAMETER** durchgeführt. Danach wird gewartet bis die vorangegangene Verarbeitung von Daten beendet ist und anschließend mit **START** die nächste Verarbeitung von Daten sofort gestartet. Damit können mehrere Datensätze lückenlos aneinandergefügt werden, wenn der Empfang neuer Daten rechtzeitig erfolgt.

Es ist auch eine Kombination beider Beispiele denkbar. Dazu muß allerdings der **START** Befehl durch den zweiten start_cmd() / write_cmd() -Aufruf rechtzeitig erfolgen, damit keine Lücken zwischen den Datensätzen entstehen.

Generell gilt: Der **START** Befehl kann erst dann ausgeführt werden, wenn das erste Datenpaket des auszugebenden Datensatzes gelesen und zur Ausgabe vorbereitet ist. Die Fehlermeldung *"Zu früher Empfang des Start-Befehles"* wird deshalb automatisch deaktiviert.

MESSZEIT

Syntax

MESSZEIT = (int)t;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Durch Aufruf von **MESSZEIT** legen Sie die Meßzeit für den angeschlossenen A/D-Wandlerblock fest. Dieser entspricht im Auswahlfeld **Trigger/Stop** dem Punkt **Messzeit**. Der Wert für *t* ist in Millisekunden anzugeben. Die Meßzeit wird nur aktiviert, wenn Sie als Triggerstopfunktion **TRIGGER_STOP** = **MESSZEIT** gesetzt haben. Als Synonym können Sie auch den Befehl **TRIGGER_STOP_MESSZEIT** verwenden, siehe Seite 143.

Beispiel*:*

```
/* der an Sequenzerausgang 5 angeschlossene A/D
Block wird auf die Triggerstopfunktion
MESSZEIT gesetzt, die Meßzeit beträgt 10 ms.
*/
start_cmd()
{
   TRIGGER_STOP = MESSZEIT;
   MESSZEIT = 10;
   START;
}
/* nach dem START wird auf das
Triggerstartereignis gewartet, 10 ms
gemessen und die Messung anschließend
beendet. */
write_cmd(5);
```

NUMMER

Syntax:

NUMMER = (int)n;

Gültig für:

GRENZWERT

Dieser Befehl gibt die Nummer des Grenzwerts an, auf den sich die Befehle **OBERER_PEGEL**, **UNTERER_PEGEL**, **PEGEL**, **WERT** beziehen. Die Grenzwerte sind in der Dialogbox von oben nach unten von 1 bis 16 durchnummeriert. (siehe auch Grenzwertblock)

Beispiel:

```
start_cmd()
/* Bei der Grenzwertüberwachung 1 wird der
Pegel auf 5 gesetzt, bei der
Bereichsüberwachung 2 wird der obere
Grenzwert  auf -1 und der untere Grenzwert
auf -3 gesetzt. */
{
   NUMMER      = 1;
   PEGEL       = 5;
   NUMMER      = 2
   OBERER_PEGEL    = -1;
   UNTERER_PEGEL  = -3;
}
write_cmd (5);
```

OBERER_PEGEL

Syntax:

OBERER_PEGEL = (float) p;

Gültig für:

GRENZWERT

Dieser Befehl setzt den oberen Grenzwert für die Bereichsüberwachung.Mit dem Blockbefehl **NUMMER** muß ausgewählt werden, auf welche Bereichsüberwachung sich der **OBERE_PEGEL** bezieht.

OFFSET

Syntax:

OFFSET;

Gültig für:

ADC H004, DAC H207, REGLER, GRENZWERT

Mit diesem Befehl legen Sie den Gleichanteil bei der Berechnung des Istwertes aus mehreren Eingangsgrößen fest. Mit diesem Befehl legen Sie den Gleichanteil fest.

Für REGLER:

Der Gleichanteil wird als konstanter Wert bei der Berechnung des Istwertes aus einer oder mehreren Eingangsgrößen addiert. Dieser Wert entspricht der Eingabe für Offset in der Dialogbox Istwertorithmetik.

Für ADC H004:

Der Gleichanteil wrd als konstanter Wert bei der Skalierung der Eingangsgröße addiert. Dieser Wert entspricht der Eingabe für Offset in der Dialogbox Skalierung.

Beispiel: siehe Faktor

PERIODISCH

Syntax:

PERIODISCH = EIN/AUS;

Gültig für:

DAC 101, DAC H004

Dieser Blockbefehl aktiviert bzw. deaktiviert bei der Datenausgabe über DAC die periodische Ausgabe im Timerbetrieb. Eine entsprechende Einstellmöglichkeit findet sich auch in der Dialogbox.

Beispiel*:*

```
/* hält den DAC an, stellt periodische
Ausgabe ein und startet den DAC wieder */

dac_h004 = 1;
start_cmd()
```

```
{
   STOP;
   PERIODISCH = EIN;
   START;
}
write_cmd(dac_h004);
```

PERIODEN_ANZAHL

Syntax:

PERIODEN_ANZAHL = n;

Gültig für:

DAC H004

Mit diesem Befehl legen Sie bei periodischer Ausgabe die Anzahl der auszugebenden Perioden fest. Ein entsprechendes Eingabefeld findet sich auch in der Dialogbox.

PRETRIGGER

Syntax:

PRETRIGGER = (int)s;

Synonym: TRIGGER_START_PRETRIGGER

analog zu TRIGGER_START_PRETRIGGER = (int)s;

PUFFER_GROESSE

Syntax:

PUFFER_GROESSE ***= (int) n ;***

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, H409 Pulsbreite, SK PORT S100, SK PORT S102

Mit diesem Befehl geben Sie die Anzahl der maximal im Puffer gehaltenen Meßwerte eines jeden A/D Kanals an. Dieser Wert

entspricht der in der Dialogbox **Optionen** einzustellenden Größe **Zwischenpufferung für *n* Werte**.

RAMPE_STEILHEIT

Syntax:

RAMPE_STEILHEIT = (float)slope;

Gültig für:

DAC 101, DAC H004

Rufen Sie diesen Blockbefehl auf, um die Steilheit der Rampe im DAC-Block einzustellen, mit welcher der vorgegebene Endwert angefahren wird. Zuvor müssen Sie den Kanal, für den die Rampensteilheit eingestellt werden soll, mit **KANAL_NUMMER** auswählen. Der Wert für *slope* wird in der Einheit Volt/Sekunde angegeben. Die Rampensteilheit ist eine Vorzeichenlose Größe.

Beispiel:

```
/* hält den DAC an, stellt die Rampensteilheit auf
100 ein und startet den DAC wieder */
start_cmd()
{
   STOP;
   RAMPE_STEILHEIT = 100;
   START;
}
write_cmd(dac101);
```

RAMPE

Syntax:

RAMPE = EIN/AUS;

Gültig für:

DAC 101, DAC H004

Mit diesem Blockbefehl aktivieren bzw. deaktivieren Sie den Rampenmodus des entsprechenden DAC Blocks. Der Rampenmodus bewirkt, daß zwischen zwei aufeinanderfolgenden Datenpaketen und nach dem Stop-Befehl die jeweiligen Werte nur mit der vorgegebenen Rampensteilheit angefahren werden.

RAMPE_ENDWERT(float)

Syntax:

RAMPE_ENDWERT = (float) endwert;

Gültig für:

DAC 101, DAC H004

Stellen Sie mit **RAMPE_ENDWERT** den Endwert des DAC ein, der nach Erhalt des Kommandos **STOP_RAMPE** oder des Stop-Befehls angefahren wird. Zuvor müssen Sie den Kanal für den der endwert eingestellt werden soll mit **KANL_NUMMER** auswählen. Den Wert für *endwert* müssen Sie in Volt angeben. Im Rampenmodus wird dieser Wert mit der vorgegebenen Rampensteilheit angefahren.

Beispiel*:*

```
/* hält den DAC an, stellt den Endwert von Kanal
1auf 0.5 Volt ein und startet den DAC wieder
*/

start_cmd()
{
   STOP;
   KANAL_NUMMER = 1;
   RAMPE_ENDWERT = 0.5;
   START;
}
write_cmd(dac101);
```

REGLER

Syntax:

REGLER = EIN/AUS;

Gültig für:

REGLER

Durch Aufruf dieses Befehls aktivieren bzw. deaktivieren Sie den **REGLER**. Die Differenzbildung zwischen Sollwert und Istwet des betreffenden **REGLER** wird durch die Deaktivierung nicht außer Kraft gesetzt. Soll auch die Differenzbildung verhindert werden, so muß der Befehl **STELL_GLEICH_SOLL** aufgerufen werden.

REGLER_KR

Syntax:

REGLER_KR = (float)kr;

Gültig für:

REGLER

Rufen Sie diesen Befehl auf, um den P-Anteil des **REGLER** neu einzustellen. Dieser Befehl entspricht dem Eingabefeld der Verstärkung in der Dialogbox.

REGLER_NUMMER

Syntax:

REGLER_NUMMER = (int)n;

Gültig für:

REGLER

Mit diesem Blockbefehlbestimmen Sie beim **REGLER**, ob sich die folgenden Blockbefehle auf den inneren oder äußeren **REGLER** beziehen.

1 = äußerer **REGLER**

2 = innerer **REGLER**

Die **REGLER_NUMMER** ist für folgende Blockbefehle relevant.

REGLER; REGLER_KR; REGLER_TN; REGLER_TV; STELLGROESSE_MIN; STELLGROESSE_MAX; SET_STELLGROESSE; STELL_GLEICH_SOLL; SOLLWERT.

REGLER_TN

Syntax:

REGLER_TN = (float)tn;

Gültig für:

REGLER

Mit diesem Befehl können Sie den I-Anteil des **REGLER** neu einzustellen. Dieser Befehl entspricht dem Eingabefeld des I-Anteils in der Dialogbox.

REGLER_TV

Syntax:

REGLER_TV = (float)tv;

Gültig für:

REGLER

Verwenden Sie diesen Blockbefehl, um den D-Anteil des **REGLER** neu einzustellen. Dieser Befehl entspricht dem Eingabefeld des D-Anteils in der Dialogbox.

RESET_STELLGROESSE

Syntax:

RESET_STELLGROESSE;

Gültig für:

REGLER

Verwenden Sie diesen Befehl, um die interne Rechengröße der Stellgröße auf den angegebenen Bereich zwischen **STELL-GROESSE_MIN** und **STELLGROESSE_MAX** zurückzusetzen. Ist der aktuelle Wert der Stellgröße des inneren oder äußeren Reglers außerhalb der angegeben Grenzen, erfolgt eine Anpassung des Wertes. Bei einem Kaskadenregler wird die Stellgröße des äußeren und inneren Reglers falls erforderlich zurückgesetzt. Dies ist hilfreich, wenn die Stellgröße durch den I-Anteil davongelaufen ist. Der korrigierte Wert wird am Ausgang des Reglers ausgegeben, und zwar unabhängig davon, ob der Regler aktiv ist oder nicht.

SET_STELLGROESSE

Syntax:

SET_STELLGROESSE = (float)s;

Gültig für:

REGLER

Dieser Befehl setzt die Stellgröße auf einen definierten Wert. Bei einem Kaskadenregler müssen Sie den regler mit **REGLER_NUMMER** auswählen, dessen Stellgröße gesetzt werden soll. Zusätzlich werden die Werte der Stellgrößen wie bei dem Befehl **RESET_STELLGROESSE** begrenzt. Der angegebene Wert wird am Ausgang des Reglers ausgegeben, und zwar unabhängig davon, ob der Regler aktiv ist oder nicht.

SOLLWERT

Syntax:

SOLLWERT = (float)s;

Gültig für:

REGLER

Geben Sie mit diesem Befehl einen neuen Sollwert vor (Regler mit **REGLER_NUMMER** angeben).

Die Daten vom Eingang **So** überschreiben diesen Sollwert wieder. Dieser Befehl sollte nur benutzt werden, wenn der Sollwert mit **HALTE_SOLLWERT** = AN eingefroren wurde.

Beispiel:

```
start_cmd()
{
   /* aktuellen Sollwert halten */
   HALTE_SOLLWERT = AN;
   /* Sollwert  1.5 vorgeben */
   SOLLWERT = 1.5;
}
write_cmd(1);
```

SOLLWERT_GLEICH_ISTWERT

Syntax:

SOLLWERT_GLEICH_ISTWERT;

Gültig für:

REGLER

Der Sollwert wird auf den neu berechneten Istwert gesetzt. Damit kann die aktuelle Position eingefroren werden. Dazu muß vorher der Befehl **HALTE_SOLLWERT** = EIN angegeben werden. Ansonsten wird dieser Sollwert durch die am Eingang **So** ankommenden Daten wieder überschrieben. Der Eingabe SO nimmt weiterhin Daten ab.

START

Syntax:

START;

Gültig für:

Alle Blöcke mit **St**-Eingang

Durch Aufruf von START starten Sie die Datenaufnahme bzw. Datenausgabe eines HYDRA-Blocks.

Bei der Datenausgabe über DAC wird das zuvor übersendete Paket genau einmal ausgegeben. Dies gilt nur, wenn die periodische Ausgabe nicht aktiviert ist. Ist die periodische Ausgabe aktiviert, werden die übergebenen Daten periodisch ausgeben. Das nächste Paket wird dann erst nach einem **STOP**-Befehl übernommen.

START_BIS_STOP

Syntax:

START_BIS_STOP;

Gültig für:

DAC 101

Dieser Befehlblock startet die Datenausgabe eines DA-Blocks und übernimmt bis zum **STOP**-Befehl weitere Daten. Sie können diese Betriebsart nur für nicht periodische Ausgabe (D/A) aktivieren. Bei der Datenaufnahme (A/D) bleibt der Befehl unwirksam. Diese Betriebsart ist zur kontinuierlichen Datenausgabe von Datei an einen D/A-Wandler geeignet.

Beispiel:

```
/* Startet den an Kanal dac101
angeschlossenen DAC */

start_cmd()
{
   START_BIS_STOP;
}
write_cmd(dac101);
/* warte 1 Sekunde */
wait(1000);
start_cmd()
```

```
{
/* halte Ausgabe an */
   STOP;
}
write_cmd(dac101);
```

START_RAMPE

Syntax:

START_RAMPE;

Gültig für:

DAC H004

Durch Aufruf dieses Befehls erfolgt nach dem Empfang des ersten Paketes eines neuen Datensatzes erfolgt die sofortige Ausgabe der Rampe. Die Ausgabe des Datenpaketes wird danach mit einem **START**-Befehl gestartet. Dadurch ist bekennt, wann durch Aufruf des **START**-Befehl die Ausgabe des Datenpaketes beginnt. Wird lediglich ein **START**-Befehl (ohne vorherigen **START_RAMPE**) aufgerufen, begunnt die Ausgabe des Datenpaketes sofort nach dem der DA-Ausgang auf den ersten Wert des Datenpaketes gefahren ist. Damit ist der Zeitpunkt, an dem die Ausgabe des Datenpaketes beginnt nicht bekannt. Ist die Rampe deaktiviert, wird der D/A-Wandler direkt auf den ersten Wert des neuen Paketes gesetzt.

Es ist zu beachten, daß bei Synchronisation mit A/D die Ausgabe der Rampe bzw. des ersten Wertes erst dann erfolgt, wenn der A/D-Wandler die Abtastung beginnt.

Beispiel:

```
/* zuerstRampe ausgeben, dann werten, erst
anschließend Datenpaket ausgeben */
start_cmd()
{
   START_RAMPE;
}
write_cmd(2);
wait (5000);
start_cmd()
{
START;
}
write_cmd(2);
```

STELL_GLEICH_SOLL

Syntax:

STELL_GLEICH_SOLL = EIN/AUS;

Gültig für:

REGLER

Mit diesem Blockbefehl kann von einem Kaskadenregler auf einen einfachen Regler umgeschaltet werden, indem der äußere Regler mit **REGLER** = AUS deaktiviert und dessen Stellgröße auf den Sollwert gesetzt wird.

Beispiel:

```
start_cmd()
{
   /* äußeren Regler auswählen */
   REGLER_NUMMER = 1;
   /* äußeren Regler abschalten */
   REGLER = AUS;
   /* Sollwert = Stellgröße des äußeren
   Reglers = Sollwert des inneren Reglers*/
   STELL_GLEICH_SOLL = EIN;
}
write_cmd();
```

STELLGROESSE_MAX

Syntax:

STELLGROESSE_MAX = (float)s;

Gültig für:

REGLER

Mit diesem Blockbefehl geben Sie den oberen Grenzwert für die Stellgröße ein. Bei einem Kaskadenregler müssen Sie mit dem Befehl **REGLER_NUMMER** den Regler auswählen für den der Grenzwert gesetzt werden soll. Der Wert der Stellgröße wird in Volt angegeben.

Beispiel:

```
start_cmd()
{
   REGLER_NUMMER = 2;  /* innerer Regler */
```

```
   STELLGROESSE_MAX = 7;
   STELLGROESSE_MIN = -7;
}
write_cmd (2);
```

STELLGROESSE_MIN

Syntax:

STELLGROESSE_MIN = (float)s;

Gültig für:

REGLER

Mit diesem Blockbefehl geben Sie den unteren Grenzwert für die Stellgröße ein. Bei einem Kaskadenregler müssen Sie mit dem Befehl **REGLER_NUMMER** den Regler auswählen für den der Grenzwert gesetzt werden soll. Der Wert der Stellgröße wird in Volt angegeben.

STOP

Syntax:

STOP;

Gültig für:

Alle Blöcke mit **St**-Eingang

Rufen Sie den **STOP**-Befehl auf, um die Datenaufnahme bzw. Datenausgabe der angeschlossenen HYDRA-Blöcke sofort anzuhalten.

STOP_ENDE_PERIODE

Syntax:

STOP_ENDE_PERIODE;

Gültig für:

DAC H004

Durch Aufrufen dieses Befehls erfolgt bei periodischer Ausgabe der Abbruch erst nach der Ausgabe aller Daten einer Periode. Dadurch wird ein definierter Endzustand erreicht.Werden die daten ncht periodisch ausgegeben, ist der Befehl nicht erlaubt.

STOP_RAMPE

Syntax:

STOP_RAMPE;

Gültig für:

DAC 101, DAC H004

Mit diesem Block halten Sie die Datenausgabe des angeschlossenen HYDRA-DAC Blocks an (die daten im FIFO der DAC-Karte werden noch ausgegeben) und fahren den eingestellten Endwert mit der vorgegebenen Rampe an.

Beispiel*:*

```
/* Startet den an Ausgang dac101 angeschlossenen
DAC Ausgang ist Kanal 3 */

dac101 = 3;
start_cmd()
{
   START_BIS_STOP;
}
write_cmd(dac101);

/* warte 1 Sekunde */
wait(1000);
/* halte Ausgabe sofort an und fahren Rampe
*/
start_cmd()
{
   STOP_RAMPE;
}
write_cmd(dac101);
```

TORZEIT

Syntax:

TORZEIT = (float)t;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, Drehgeber H409, SK PORT S100, SK PORT S102, Zähler CARD

Mit diesem Befehl stellen Sie die **TORZEIT** in Millisekunden für dieFrequenzmessung ein. Diese Eingabe entspricht der **TORZEIT**, die Sie in der Dialogbox im Betriebsmodi Frequenzmessung angeben können.

TRIGGER_START=

Syntax:

TRIGGER_START = trigger;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Befehl setzen Sie die Bedingung für die Triggerstartfunktion, d.h. den Meßbeginn. Die Variable *trigger* kann folgende Werte annehmen:

- **KEIN_TRIGGER** - Messung beginnt sofort nach dem **START**-Befehl.
- **PEGEL** - Messung beginnt bei Erreichen des vorgegebenen Pegels.
- **FLANKE** - Messung beginnt bei Durchschreiten der vorgegebenen Flanke.
- **PEGELBEREICH** - Messung beginnt bei Erreichen des vorgegebenen Pegelbereichs.
- **FLANKENBEREICH** - Messung beginnt, wenn die Flanke den vorgegebenen Bereich durchschreitet.
- **EINGANG_ST** - Meßbeginn unmittelbar bei Empfang eines Startsignals über Eingang **St**. Ist diese Triggerfunktion nicht aktiv, so wird die Meßaufnahme nach Empfang des

Startsignales und dem Erreichen der jeweiligen Triggerstart-Bedingung begonnen.

Generell gilt: Damit die bei angeschlossenem ST_Button, die Messung beginnen kann, muß immer ein **START** befehl gesendet werden.

PEGEL_TRIGGERBUCHSE - Messung beginnt bei Erreichen des vorgegebenen Pegels am externen Triggereingang des jeweiligen IO PORTs. Die Triggerbuchse ist nicht bei allen IO PORTs vorhanden.

FLANKE_TRIGGERBUCHSE - Messung beginnt bei Durchschreiten der vorgegebenen Flanke an der jeweiligen externen Triggerbuchse.

Diese Werte entsprechen den im Auswahlfeld **Trigger/Start** wählbaren Triggerfunktionen.

Beispiel*:*

```
/* aktiviert Kanal 3 */
start_cmd()
{
   KANALMUSTER = 0100;
   /* Abtastrate 0.1 Millisekunden
   */ABTASTRATE = 0.1;
   /* akktiviere Triggerstartfunktion FLANKE
   */
   TRIGGER_START = FLANKE;
   /* Starttrigger bei steigender Flanke */
   TRIGGER_START_FLANKE = STEIGEND;
   /* es wird auf Kanal 3 getriggert */
   TRIGGER_START_CHANNEL = 3;
   KANAL_NUMMER = 3;
   KANAL = EIN;
   TRIGGER_STOP = DATENZAHL;
   DATENZAHL = 200;
   PRETRIGGER = 0;
   START;
}
write_cmd(adc_h003);
```

TRIGGER_START_BEREICH

Syntax:

TRIGGER_START_BEREICH = INNERHALB/AUSSERHALB;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Blockbefehl legen Sie fest, ob die Triggerstartfunktion für die Triggerart **PEGELBEREICH** innerhalb oder außerhalb des spezifizierten Bereichs aktiviert wird.

Beispiel:

```
start_cmd()
{
   /* aktiviere Bereichstrigger */
   TRIGGER_START = PEGELBEREICH;
   /* Bereich ist von -1.0 bis +1.0 Volt */
   TRIGGER_START_UNTERER_PEGEL = -1.0;
   TRIGGER_START_OBERER_PEGEL = 1.0;
   /* Meßbeginn bei Pegel innerhalb des
   Bereiches */
   TRIGGER_START_BEREICH = INNERHALB;
}
write_cmd(1);
```

TRIGGER_START_FLANKE

Syntax:

TRIGGER_START_FLANKE = STEIGEND/FALLEND;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Hier legen Sie fest, ob die Triggerstartfunktion für die Triggerart **FLANKE** für die steigende oder die fallende Flanke aktiviert wird.

TRIGGER_START_KANAL

Syntax:

TRIGGER_START_KANAL = (int)k;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Durch Aufruf dieses Blockbefehls wählen Sie den Kanal aus, bei dem die Bedingung für den Triggerstart abgeprüft wird.

Beispiel:

```
start_cmd()
{
   /* Triggerstartfunktion für steigende
   Flanke */
   TRIGGER_START = FLANKE;
   /* Trigger wird für Kanal 3 aktiviert */
   TRIGGER_START_KANAL = 3;
}
write_cmd(adc_h004);
```

TRIGGER_START_PEGEL

Syntax:

TRIGGER_START_PEGEL = UEBERSCHRITTEN/ UNTERSCHRITTEN;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Legen Sie mit diesem Befehl fest, ob bei der Triggerstartfunktion **TRIGGER_START** = **PEGEL** die Messung bei Überschreiten oder Unterschreiten des Pegels beginnen soll.

Beispiel:

```
start_cmd()
{
   /* Triggerstartfunktion für
   Pegelüberschreitung */
   TRIGGER_START = PEGEL;
   /* Trigger für Pegelüberschreitung
   aktivieren */
   TRIGGER_START_PEGEL = UEBERSCHRITTEN;
   /* Triggerpegel liegt bei 3 Volt */
   TRIGGER_START_PEGEL_WERT = 3.0;
}
write_cmd(adc_h003);
```

TRIGGER_START_PEGEL_WERT

Syntax:

TRIGGER_START_PEGEL_WERT = (float)u;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit dieser Funktion legen Sie den Triggerpegel fest, bei dem die Messung für die Triggerstartfunktion **TRIGGER_START** = **PEGEL** beginnen soll. Je nach Einstellung wird die Messung bei Überschreiten oder Unterschreiten des Pegels begonnen. Der Wert für *u* ist in Volt anzugeben.

Beispiel: siehe **TRIGGER_START_PEGEL**

TRIGGER_START_OBERER/UNTERER_PEGEL

Syntax:

TRIGGER_START_OBERER_PEGEL = (float)p;
TRIGGER_START_UNTERER_PEGEL = (float)p;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Befehl definieren Sie die oberen und unteren Bereichsgrenzen für die Triggerstartfunktion **TRIGGER_START** = **PEGELBEREICH**. Der Wert für *p* ist in Volt anzugeben.

Beispiel:

```
start_cmd()
{
   /* Triggerstartfunktion für
   Pegelbereichsüberschreitung */
   TRIGGER_START = PEGELBEREICH;
   /* Trigger für
   Pegelbereichsüberschreitung
   aktivieren, gemessen wird sobald
```

```
   die Spannung am Triggerkanal den Bereich
   von -3.0 bis +1.5 Volt verläßt */
   TRIGGER_START_BEREICH = AUSSERHALB;
   TRIGGER_START_OBERER_PEGEL = 1.5;
   TRIGGER_START_UNTERER_PEGEL = -3.0;
}
write_cmd(adc_h00);
```

TRIGGER_START_PRETRIGGER

Syntax:

TRIGGER_START_PRETRIGGER = (int)s;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Befehl legen Sie fest, wieviele Werte *s* vor dem Triggerereignis noch in die Messung miteinbezogen werden sollen. Synonym zu diesem Befehl kann auch der Befehl **PRETRIGGER** verwendet werden.

TRIGGER_START_UNTERER_PEGEL

Syntax:

TRIGGER_START_UNTERER_PEGEL = (float)p;

siehe: **TRIGGER_START_OBERER_PEGEL**;

TRIGGER_STOP

Syntax:

TRIGGER_STOP = (float)trigger;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Rufen Sie diesEn Befehl auf, um die Bedingung für die Triggerstopfunktion, d.h. das Meßende zu setzen. *trigger* kann folgende Werte annehmen:

- **ENDLOS_ABTASTEN** - nach Meßbeginn wird endlos abgetastet, d.h. bis zum **STOP**-Befehl.
- **MESSZEIT** - nach Meßbeginn wird die Messung nach Ablauf der vorgegebenen Meßzeit beendet.
- **DATENZAHL** - nach Meßbeginn wird die Messung nach der Anzahl der in DATENZAHL vorgegebenen Abtastungen beendet.
- **PEGEL** - Messung endet bei Erreichen des vorgegebenen Pegels.
- **FLANKE** - Messung endet bei Durchschreiten der vorgegebenen Flanke.
- **PEGELBEREICH** - Messung endet bei Erreichen des vorgegebenen Pegelbereichs.
- **FLANKENBEREICH** - Messung endet bei Durchschreiten des vorgegebenen Flankenbereichs.
- **PEGEL_TRIGGERBUCHSE** - Messung endet bei Erreichen des vorgegebenen Pegels am externen Triggereingang des jeweiligen IO PORTs. Die Triggerbuchse ist nicht bei allen IO PORTs vorhanden.
- **FLANKE_TRIGGERBUCHSE** - Messung endet bei Durchschreiten der vorgegebenen Flanke an der jeweiligen externen Triggerbuchse.

Diese Werte entsprechen den im Auswahlfeld **Trigger Stop** wählbaren Triggerfunktionen, soweit diese dort vorhanden sind.

Beispiel:

```
adc_h004=2;
start_cmd()
{
   /* Triggerstopfunktion für
   Pegelbereichsüberschreitung */
   TRIGGER_STOP = PEGELBEREICH;
   /* Trigger für
   Pegelbereichsüberschreitung
   aktivieren. Es wird solange gemessen, bis
   die Spannung am Triggerkanal den Bereich
   von -3.0 bis +1.5 Volt verläßt */


   TRIGGER_STOP_BEREICH = AUSSERHALB;
   TRIGGER_STOP_OBERER_PEGEL = 1.5;
   TRIGGER_STOP_UNTERER_PEGEL = -3.0;
}
write_cmd(adc_h004);
```

TRIGGER_STOP_BEREICH

Syntax:

TRIGGER_STOP_BEREICH = INNERHALB/AUSSERHALB;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Legen Sie mit diesem Befehl fest, ob die Triggerstopfunktion für die Triggerart **PEGELBEREICH** innerhalb oder außerhalb des spezifizierten Bereichs aktiviert wird.

Beispiel: siehe **TRIGGER_STOP;**

TRIGGER_STOP_FLANKE

Syntax:

TRIGGER_STOP_FLANKE = STEIGEND/FALLEND;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Befehl legen Sie fest, ob die Triggerstopfunktion **FLANKE** für die steigende oder die fallende Flanke aktiviert wird.

TRIGGER_STOP_PEGEL

Syntax:

TRIGGER_STOP_PEGEL =
UEBERSCHRITTEN/UNTERSCHRITTEN;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Legen Sie hier fest, ob bei der Triggerstopfunktion **TRIGGER_STOP** = **PEGEL** die Messung bei Überschreiten oder Unterschreiten des Pegels beendet werden soll.

TRIGGER_STOP_MESSZEIT

Syntax:

TRIGGER_STOP_MESSZEIT = (int)t ;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Mit diesem Blockbefehl definieren Sie, nach welcher Zeit *t* bei der Triggerstopfunktion **TRIGGER_STOP** = **MESSZEIT** die Messung beendet werden soll. Der Wert von *t* ist in Millisekunden anzugeben. Als Synonym kann der Befehl **MESSZEIT** verwendet werden.

TRIGGER_STOP_OBERER/UNTERER_PEGEL

Syntax:

TRIGGER_STOP_OBERER_PEGEL = (float)p;
TRIGGER_STOP_UNTERER_PEGEL = (float)p;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Legen Sie die oberen und unteren Bereichsgrenzen für die Triggerstopfunktion **TRIGGER_STOP PEGELBEREICH** fest. Der Wert für *p* ist in Volt anzugeben.

Beispiel:

```
adc_h004=2;
start_cmd()
{
   /* Triggerstopfunktion für
   Pegelbereichsüberschreitung */
   TRIGGER_STOP = PEGELBEREICH;
   /* Trigger für
   Pegelbereichsüberschreitung
   aktivieren. Es wird solange gemessen,
   bis die Spannung am Triggerkanal den
   Bereich von -3.0 bis +1.5 Volt verläßt */
   TRIGGER_STOP_BEREICH = AUSSERHALB;
   TRIGGER_STOP_OBERER_PEGEL = 1.5;
   TRIGGER_STOP_UNTERER_PEGEL = -3.0;
}
write_cmd(adc_h004);
```

TRIGGER_STOP_PEGEL_WERT

Syntax:

TRIGGER_STOP_PEGEL_WERT = (float)u;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Bestimmen Sie hier den Triggerpegel, bei dem die Messung für die Triggerstopfunktion **TRIGGER_STOP PEGEL** beendet werden soll. Der Wert für *u* ist in Volt anzugeben.

TRIGGER_STOP_KANAL

Syntax:

TRIGGER_STOP_KANAL = (int)k;

Gültig für:

ADC CARD, ADC H004, ADC H109, ADC S002, IO PORT H407, SK PORT S100, SK PORT S102

Legen Sie hier den Kanal fest, bei welchem die Triggerstopfunktion aktiv werden soll, d.h. welcher Kanal auf den eingestellten Pegel oder die vorgegeben Flanke überprüft wird.

TRIGGER_STOP_WERTE

Synonym zu **DATENZAHL**, siehe Seite 108.

UNTERER_PEGEL

Syntax:

UNTERER_PEGEL = (float)p;

Gültig für:

GRENZWERT

Dieser Befehl setzt den unteren Grenzwert für die Bereichsüberwachung.

Mit dem Blockbefehl **NUMMER** muß ausgewählt werden, auf welche Bereichsüberwachung sich der **UNTERE_PEGEL** bezieht.

VERSTAERKUNG

Syntax:

VERSTAERKUNG = (int)v;

Gültig für:

ADC H109/H110

Mit diesem Blockbefehl stellen Sie den Verstärkungsfaktor des angeschlossenen HYDRA-ADC Blocks ein. Die Werte von *v* können 1 2 4 8 oder 1 10 100 1000 sein, je nachdem welche Verstärker im System vorhanden sind.

WARTE_ENDE_VERARBEITUNG

Syntax:

WARTE_ENDE_VERARBEITUNG;

Gültig für:

DAC H004

Dieser Befehl dient der Synchronisation des DAC-Blockes mit dem Sequenzer. Der Aufruf des Befehls bewirkt. daß keine weiteren Befehle vom DAC-Block entgegengenommen werden bevor der gerade verarbeitete datensatz nicht vollständig angegeben ist. Die Initialisierung des DAC-Blockes erfolgt im Gegensatz zum Befehl **LETZTER_PARAMETER** also erst nachdem der aktuelle Datensatz ausgegeben ist. Eine lückenlose Ausgabe mehrerer Datensätze ist damit nicht möglich.

Beispiel: Im Beispiel wird die Anwendungsweise des Befehls gezeigt.Der zweite start_cmd()/write_cmd()-Aufruf ist optional. Er dient lediglich dazu, dem Sequenzer "mitzuteilen", daß der vorhergehende Datensatz vollständig ausgegeben ist.

Der Befehl **WARTE_ENDE_VERARBEITUNG** hat hier lediglich Dummy-Charakter.

```
start_cmd()
{

   KANAL_NUMMER = 1;
   RAMPE_ENDWERT = 5.;
   AUSGABERATE = 10;
   START;
   WARTE_ENDE_VERARBEITUNG;
}
write_cmd(1);

start_cmd()
{
   WERTE_ENDE_VERARBEITUNG;
}
write_cmd(1);

start_cmd()
{
   KANAL_NUMMER = 1;
```

```
    RAMPE_ENDWERT = 0;
    AUSGABERATE = 20;
    SATRT;
    WARTE_ENDE_VERARBEITUNG;
}
write_cmd(1);
```

WERT

Syntax:

WERT = (float) w;

Gültig für:

GRENZWERT

Mit diesem Befehl kann ein Ausgang Rn des Grenzwertblockes auf den Wert w gestezt werden. Die Auswahl des Ausgangs Rn erfolgt indirekt über die Nummer der entsprechenden Grenzwertüberwachung (siehe Dialogbox)

Beispiel:

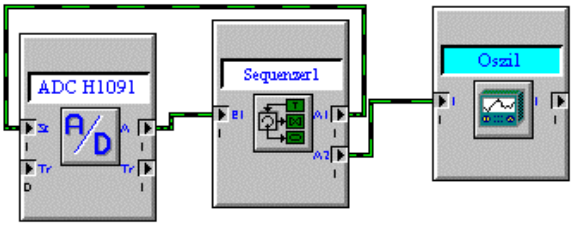
```
start_cmd()
{
    NUMMER = 5;
    WERT = 6.0 ;
}
write_cmd(1)
```

Im Beispiel wird der Wert 6 auf dem Ausgang des fünften Grenzwertes ausgegeben.

Alle Grenzwertüberwachungen, die ihre Reaktion auf dem fünften Ausgang ausgeben, können bei aktiver Grenzwertüberwachung überschreiben. Dadurch kann die Reaktion auf einen Fehler nur dann mit Wert zurückgesetzt werden, wenn dieser Fehler nicht mehr vorliegt.

Beispiel

Das folgende Beispiel zeigt die Steuerungsmöglichkeiten von HYDRA-Blöcken mit Hilfe des **SEQUENZERS**. Die Aufgabe besteht darin, die Einstellungen der Eingangsverstärker des A/D-Wandlers auf das anliegende Eingangsignal zu optimieren und die Darstellung auf dem Oszilloskop automatisch anzupassen.



Der Schaltplan ist untenstehendem **Sequenzer** Programm zugeordnet.

Das Beispielprogramm nimmt alle notwendigen Einstellungen des ADC H109 Blockes vor und ruft das Unterprogramm **data_to_digis** auf. Das Programm **data_to_digis** nimmt Daten vom ADC H109 Block über **E1** auf und schickt die Daten an das Oszilloskop über **A2** weiter, ruft vorher zur Skalierung das Programm **set_range** auf. Anschließend wird der optimale Verstärkungsfaktor ermittelt und dieser am ADC H109 Block eingestellt.

Beispiel1 **zur Sequenzerprogrammierung:**

```
/* AD-Wandlersteuerung für H109 Kanal 3
anschließen */
float len;
float dat[200];
float i;
float min, max;
float delta;
HEADER hd_1;

void start(PAR)
{
```

```
start_cmd()
{
   /* Messung in Blöcke der Länge 200
   zerlegen */
   BLOCKLAENGE = 200;
   /* Verstärkung zunächst auf 1 */
   VERSTAERKUNG = 1;
   /* Kanal 3 aktivieren */
   KANALMUSTER = 0100;
   SAMPLERATE = 0.1;  /* 0.1 ms
   Flankentrigger mit steigender Flanke
   auf Kanal 3 bei steigendem Durchgang
   von 0 Volt */

   TRIGGER_START =
   FLANKE;TRIGGER_START_FLANKE = STEIGEND;
   /* Triggerkanal 3 */
   TRIGGER_START_KANAL = 3;
   TRIGGER_START_PEGEL_WERT = 0;
   /* Messende nach 600 Werten, d.h. 3
   Datenblöcke */
   TRIGGER_STOP = DATENZAHL;
   DATENZAHL = 600;
   PRETRIGGER = 0;
}
write_cmd(1);

/* Automatische Verstärkungsumschaltung
bis Eingangssignal > 8 Volt */
while(1)
{
   start_cmd()
   {
      START;
   }
   write_cmd(1);
   call(data_to_digis);
   if(max > 0.92)
   {start_cmd()
      {
         VERSTAERKUNG = 1;
      }
      write_cmd(1);
   }
   if(max < 0.9)
   {
      start_cmd()
      {
```

```
            VERSTAERKUNG = 10;
         }
         write_cmd(1);
      }
      if(max < 0.09)
      {
         start_cmd()
         {
            VERSTAERKUNG = 100;
         }
         write_cmd(1);
      }
      if(max < 0.009)
      {
         start_cmd()
         {
            VERSTAERKUNG = 1000;
         }
         write_cmd(1);
      }
   }
}
/* Daten von Eingang 1
an Ausgangsbutton 2 weiterleiten bis zum
Ende der Messung. Die Variable 'read_end'
ist dann 1 */
void data_to_digis(PAR)
   {
   int read_end;

   while(1) /* diese Schleife wird 3 mal
   durchlaufen */
   {
      len = 200;
      read(1, dat, len, hd_1);
      call(set_range);
      write(2, dat, len, hd_1);
      test_lastblock(hd_1, read_end);
      if(read_end == TRUE) return;
   }
}

/* Maximum und Minimum bestimmen,
Bereich am Oszilloskop setzen */
void set_range(PAR)
{
   /* debug(0); */
   i = 0;
```

```
   min = 10; /* min auf größten möglichen
   Wert setzen */
   max = -10; /* max auf kleinsten möglichen
   Wert setzen */
   do
   {
      if(dat[i] > max) max = dat[i];
      if(dat[i] < min) min = dat[i];
      i = i + 1;
   }
   while(i < len);
   if(max > 0) max = max * 1.1;
   if(min < 0) min = min * 1.1;
   delta = max - min;
   set_y0(hd_1, min);

   set_yrange(hd_1, delta);
   /* debug(1); */
}
```

Beispiel2:

```
/* (c) Kinzinger Systeme GmbH
   Industriestr. 15
   76437 Rastatt
   Germany */

/* Dieser Sequenzer liest vom Eingang 1
immer einen Wert ein und gibt diesen auf dem
Ausgang 1 aus. */


/* Globale Variablen */

HEADER h_x;
HEADER h_y;

float lese;
float schreibe;

/* Buttons */

/* Eingänge von Digital-In */

int EINGANG_1 = 1;

/* Ausgänge */

int AUSGANG_1 = 1;
```

```
/* Funktionen */

void lesen(PAR, int ein)
{
   read_var(ein, lese, h_x);

/* Diese Zeile wird nur ausgegeben, wenn
in der Dialogbox des Sequenzers
die Option 'Standardausgabe aktivieren'
aktiviert wurde */

   printf("<demo> gelesener Wert =
   %f\n",lese);
}

/* Hauptroutine */

void start(PAR)
{
   int button;

   int SYNCHRON = 0;
   /* Initialierung fuer synchrones Lesen */

   int ASYNCHRON = 1;
   /* Initialierung fuer asynchrones Lesen
   */

   err_printf("<demo> Start\n");
   init_header(h_x);
   init_header(h_y);

   init_read_par(EINGANG_1, SYNCHRON);
   init_write_par(AUSGANG_1);

   while(1)
   {
      /* warte bis von einem der Buttons was
      ankommt */
      wait_read_par_list(button, EINGANG_1, -
      1);
      lesen(p, button);
      schreibe = lese;
      write_var(AUSGANG_1, schreibe, h_y);
   }
} /* start() */
```

Fehlerbehandlung

Fehlermeldungen teilen sich auf in:

- syntaktische Fehler
- Laufzeitfehler

Fehlermeldungen werden generell im Ausgabefenster **STD-IO** des **HYDRA Control Programms** angezeigt.

Werden syntaktische Fehler im **Sequenzer** Programm festgestellt, so wird das **Sequenzer** Programm nicht gestartet. Alle anderen HYDRA-Blöcke bleiben davon unberührt und laufen wie gewohnt weiter.

Laufzeitfehler werden auch im Ausgabefenster **STD-IO** des **HYDRA Control Programms** gemeldet.

Zu jedem Fehler wird die entsprechende Zeilennummer des **Sequenzer** Programms mit ausgegeben, in der der Fehler auftrat. Damit sind die Fehler schnell lokalisierbar.

Als weitere Hilfe zur Fehleranalyse zur Laufzeit dient der Debugmodus, der mit der Debug()-Funktion an und abgeschaltet werden kann. Mit seiner Hilfe läßt sich der Programmablauf gezielt im Fenster **STD-IO** des **HYDRA Control Programms** verfolgen.

Detaillierte Informationen über das **HYDRA Control Programm** erhalten Sie im folgenden Kapitel.

Index

