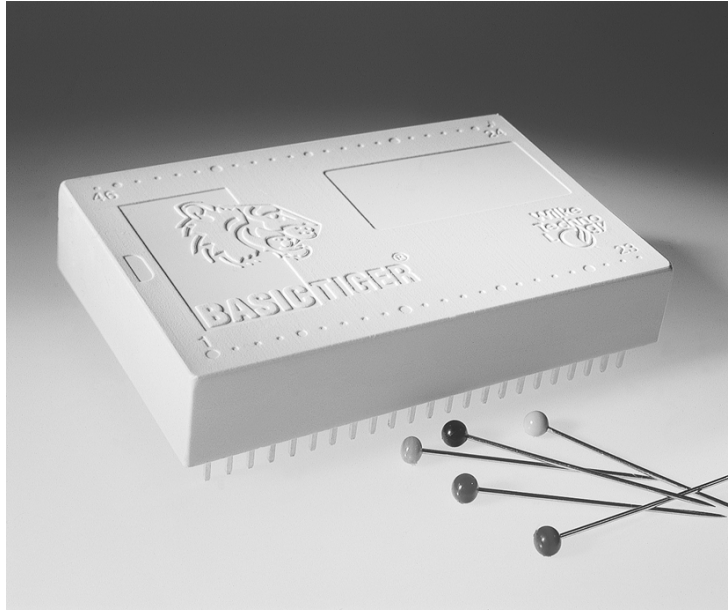




BASIC-TIGER®

Programmier-Handbuch

Einführung	1
Die Tiger-BASIC[®]-Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger[®]	4
Tiger-BASIC[®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Redaktion Klaus Hiltrop
Abbildungen Joji
Cover Werbeagentur Kordoni, Aachen
Druck Print Production, Aachen
5. Auflage - Version 5.0

Copyright © Wilke Technology GmbH
Heider-Hof-Weg 23D
52080 Aachen, Germany

Dieses Handbuch, sowie die Hard- und Software, die es beschreibt, ist urheberrechtlich geschützt und darf ohne ausdrückliche schriftliche Genehmigung von Wilke Technology GmbH in keiner Weise vervielfältigt, übersetzt oder in eine andere Darstellungsform gebracht werden.

Warenzeichen BASIC-Tiger®, TINY-Tiger®, TigerCube® sind eingetragene Warenzeichen von Wilke Technology GmbH.
TouchMemory® ist eingetragene Warenzeichen von Dallas Semiconductors.
Windows™, Windows 95, Windows NT sind eingetragene Warenzeichen der Microsoft Corp.

Diejenigen Bezeichnungen in dieser Publikation von Erzeugnissen und Verfahren, die zugleich Warenzeichen sind, wurden nicht besonders kenntlich gemacht. Solche Namen sind Warenzeichen der jeweiligen Warenzeicheninhaber. Aus dem Fehlen der Markierung® kann nicht geschlossen werden, daß diese Bezeichnungen freie Warennamen sind.

Hinweis Herausgeber, Übersetzer und Autoren dieser Publikation haben mit größter Sorgfalt die Texte, Abbildungen und Programme erarbeitet. Dennoch können Fehler nicht völlig ausgeschlossen werden. Wilke Technology übernimmt daher weder eine Garantie noch eine juristische Verantwortung oder Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen. Mitteilungen über eventuelle Fehler werden jederzeit gerne entgegengenommen.

Die Angaben in diesem Handbuch gelten nicht als Zusicherung bestimmter Produkteigenschaften. Änderungen, die dem technischen Fortschritt dienen, bleiben vorbehalten.

Alle Rechte vorbehalten • Printed in Germany
Gedruckt auf chlorfrei gebleichtem Papier

Inhaltsverzeichnis

1	Einführung	3
	Neue Instruktionen und Funktionen in der Version 5.0	4
	Was ist nicht ganz so neu?	6
	Systemvoraussetzungen	8
	Sicherheitshinweis	9
	Typografische Konventionen und Symbole	10
	BASIC-Tiger®	11
	Multitasking	12
	Device-Treiber	12
	Funktionen	14
2	Die Tiger-BASIC®-Sprache	17
	Struktur von Tiger-BASIC®-Programmen	17
	Schreibweise	17
	Überlange Zeilen	18
	Kommentare	18
	Die Task MAIN	18
	Unterprogramme	21
	Datentypen	24
	BYTE	24
	WORD	24
	LONG	25
	REAL	25
	STRING	26
	DATALABEL	27
	Zusammengesetzte Datentypen	28
	ARRAY	28
	FIFO-Puffer	30
	Variable initialisieren	31
	Variable deklarieren	32
	Gültigkeitsbereich von Variablen	33
	Variablennamen	33
	Operatoren	34
	Prioritäten beim Rechnen	35
	Präprozessor	36
	#include	36
	#define	36
	#comment, #endcomment	37
	#project_model	38
	BASIC-Funktionen	39
	Compiler-Anweisungen	40

Inhaltsverzeichnis

USER_STACK_SIZE	40
USER_STRING_SIZE	41
USER_VAR_STRICT	41
USER_SECURITY	42
SET_DATA_ADDR	43
DATA	44
Daten aus Datei ins Flash einbinden.	47
Positionen im Flash benennen	49
Flashdaten aus User-Flashbereich in Datei speichern	50
Fehlerbehandlung während der Laufzeit	51
Explizite Fehlerbehandlung	51
Implizite Fehlerbehandlung	52
Run-Time-Fehler	53
3 Instruktionen und Funktionen	57
Übersicht nach Gruppen	57
Kommentierte Übersicht	61
ABS	72
ACOS	73
ACOT	74
ADD\$	75
AND\$	77
ANIB_BYTE\$	79
ASC	81
ASC_BCD\$	82
ASC_PBCD\$	84
ASIN	86
ATAN	87
BCD_ASC\$	88
BCD_PBCD\$	90
BIT	91
BIT_DEMIX\$	92
BIT_MIRR	93
BIT_MIRR\$	95
BIT_MIX\$	97
BITNOT	105
BYTE_ANIB\$	106
BYTE_DEMIX\$	108
BYTE_INIB\$	110
BYTE_MIRR	112
BYTE_MIX\$	114
CALC_CRC	118
CALL	120
CHECK_FLASH	125
CHR\$	127

Inhaltsverzeichnis

CLEAR_FIFO	128
CODE_1TON\$	129
CONT_TASK	133
CONVERT\$	134
CONVERT7\$	136
COS	138
COSH	139
COT	140
COTH	141
CUT_AND_PASTE	142
CUT_AND_PASTE\$	144
CUT_BITS	146
DEBOUNCE	148
DECODE	151
Fehlerkorrigierende Codes	154
DELETE_PROG	164
DEVEN	166
DIFF_TICKS	167
DIR_PIN	169
DIR_PORT	170
DISABLE_TSW	171
DISTRIBUTE\$	173
ENABLE_TSW	190
END	191
ERASE_FLASH	192
EXIT_TASK	195
EXP	197
EXPE	198
FIFO	199
FILL\$	200
FIX	203
FOR...NEXT	204
FRAME	206
FREE_FIFO	208
GET	209
GET_FIFO	212
GOTO	214
HREAL	215
I2C_ACK_POLL	216
I2C_READ\$	218
I2C_RESULT	220
I2C_SETUP	222
I2C_WRITE	225
IF...THEN	227
IMASK	229

Inhaltsverzeichnis

IN	230
INDEX	231
INDEX_1D	233
Hysterese	236
Anwendungen für INDEX_1D	239
Tastatur mit INDEX_1D	239
INDEX_2D	243
INIB_BYTE\$	248
INPUT	250
INPUT_LINE	252
INSTALL_DEVICE	254
INSTR	256
INT	258
INTEGRAL_FIFO	259
INV_BIT	262
LD	263
LEFT\$	264
LEN	265
LEN_FIFO	266
LET\$	267
LIMIT	268
LIN_APPROX	270
LL_IPORT_AND	272
LL_IPORT_IN	273
LL_IPORT_OR	274
LL_IPORT_OUT	275
LL_IPORT_PULSE	276
LL_IPORT_XOR	277
LLTOR	278
LN	279
LOG	280
LOOP	281
LREAL	283
LTR	284
MASK	285
MAX_LEN	286
MID\$	287
MIRROR\$	288
MOD	289
MODULO_INC	290
MODULO_UPDO	293
Unregelmäßige Zahlenfolge	296
NFROMS	297
NTOS\$	299
ON_ERROR_RESET	303

Inhaltsverzeichnis

ON_ERRTASK_CALL	304
ON_ERRTASK_GOTO	306
OR\$	307
OUT	309
PARITY	311
PACK_BYTE\$	313
PBCD_ASC\$	315
PBCD_BCD\$	317
PEEK_FLASH	318
POKE_FLASH	320
POKEM_FLASH	322
PRIME	325
PRINT	328
PRINT_USING	331
PUT	334
PUT_FIFO	336
PUTU_FIFO	338
QUICK_WORD_COS	340
QUICK_WORD_SIN	342
RANDOMIZE	344
READ_FIFO	345
RELEASE_TASK	346
REMDOUBLE\$	347
REMOVE\$	348
RES_BIT	349
RESTART_PROG	350
RETURN	351
RFROMS	352
RIGHT\$	354
RND	355
RTL	356
RTOS\$	357
RUN_TASK	359
SCAN_TO_CHAR\$	361
SCAN_TO_CHAR_SETUP	363
SCAN_TO_TOGGLE	365
SCRAMBLE\$	366
SELECT\$	370
SERIAL_NO\$	372
SET_BIT	374
SET_LEN\$	375
SET_SERIAL_NO	376
SET_SYSVARN	379
SET_TASK_PRIO	381
SET_TICKS	383

Inhaltsverzeichnis

SGN	384
SHIFT_IN	385
SHIFT_OUT	389
SHIFTL\$	393
SHIFTM\$	396
SIGNAL_ADD	399
SIGNAL_AVERAGE	401
SIGNAL_BIGGER	403
SIGNAL_CUT_AMP	404
SIGNAL_CUT_MAX	406
SIGNAL_CUT_MIN	408
SIGNAL_GAP_END	409
SIGNAL_GAPI_END	411
SIGNAL_MAX	413
SIGNAL_MAX_AMPL	415
SIGNAL_MIN	417
SIGNAL_MUL	419
SIGNAL_NEXT_GAP	421
SIGNAL_NEXT_GAPI	423
SIGNAL_OFFSET	425
SIGNAL_SCALE	426
SIGNAL_SMALLER	428
SIGNAL_SMOOTH	429
SIGNEXT	431
SIN	434
SINH	435
SQRT	436
START_FIFO	437
STOP_TASK	439
STOS\$	441
STR\$	444
STRI\$	445
SUB	446
SWITCH...CASE	450
SWITCHI...CASE	454
SYSVAR\$	458
SYSVARN	460
TABS	463
TAN	470
TAN – Trans-Aktions-Nummern	471
TAN_CHK	473
TAN_COUNT	477
TAN_DEL	479
TAN_FIRST	483
TANH	485

TASK	486
THIS_VERS	487
TICKS	489
TRIM\$	490
UNGET_FIFO	492
UNPACK_BYTE\$	494
UPPER\$	496
UPPER7\$	497
USING	498
Formatstrings	499
VAL_NUM	518
VAL_REAL	520
WAIT_CLOCK	521
WAIT_DURATION	523
WAIT_NEXT	525
WHILE	527
XBUS_INR\$	530
XBUS_OUTR	532
XIN, XIN\$	534
XOR\$	537
XOUT	539
XSETUP	541
Umschreibung mathematischer Funktionen	544
4 Graphik im BASIC-Tiger®	549
Übersicht über Grafikfunktionen	549
Pixelgrafik - Vektorgrafik	552
Bedeutung von Pixelgrafik	555
Bearbeitungsmöglichkeiten von Pixelgrafik	556
Koordinatensystem für Pixelgrafik	557
Typische Beispiele für Pixelgrafik	558
Vektorgrafik	560
Arbeiten mit Pixelgrafik	567
Grafik im Flash	567
Grafik & Geschwindigkeit	570
Grafik mit PEEK_FLASH	572
Grafik direkt aus dem Flash	574
Bitmap mit zu kurzen Zeilen	576
Bitmap mit zu kleiner Höhe	578
Bitmap mit zu großer Höhe	580
Beispiel-Bitmaps	582
AND\$, OR\$, XOR\$	589
GRAPHIC_MASK_COPY	600
GRAPHIC_MIRROR	604
GRAPHIC_COPY	607

Inhaltsverzeichnis

Durchrollende Ziffer	611
Spruchband	614
Bewegte Straßenkarte und bewegte Box	616
Mehrere Fenster auf einem Bildschirm	619
Einkopieren eines Cursors mit beliebiger Kontur	622
Grafische Fenster kopieren mit verschiedenen Überlagerungsarten	625
INVERT	627
GRAPHIC_FILL_MASK	629
GRAPHIC_EXP\$	638
Arbeiten mit Vektorgrafik	644
DRAW_LINE	645
Zeichen einer Verbindungslinie zwischen 2 Punkten	650
Zeichen einer Linie mit mehreren Punkten	653
Zeichen einer geschlossenen Linie mit mehreren Punkten	655
Verschieben eines Linienzuges	657
Rotieren eines Linienzuges	661
Skalieren eines Polygons	663
„Weiße“ Linien zeichnen	665
Zeichnen mit mehreren Stiften	667
Zeichnen mit Polarkoordinaten	669
Zeichnen und füllen mit Polarkoordinaten	671
Zeichnen und füllen mit kartesischen Koordinaten	673
Polygon zeichnen und füllen	675
Polygon zeichnen und füllen - jedoch fehlerhaft	677
Polygon zeichnen und füllen, sowie schwarze Linien zeichnen	679
Timing- / Phasenproblem bei bewegten Grafiken	681
Timing- / Phasenproblem behoben	683
Anwendungsbeispiel „Uhr“	685
Transformationen: erst skalieren, dann rotieren und umgekehrt	688
Polarkoordinaten - PHI-Skalierung	692
Polarkoordinaten - PHI-Skalierung (2)	695
Polarkoordinaten - R-Skalierung	699
Automatische Begrenzung von Vektorgrafik auf aktiven Grafikbereich	702
Vektorgrafik ohne automatische Begrenzung auf aktiven Grafikbereich	704
Gefüllte Flächen und kleine Wiedergabegrößen	707
Kleine Strukturen mit DRAW_LINE und FILL_AREA	711
Dicke Linien	715
Spitze Konturen	720
Überlagerung von Flächen	723
Zeichen mit „unsichtbarer“ Linie als Test.	725

DRAW_NEXT_LINE	727
CLOSE_LINE	729
SET_BASE	731
SET_SCALE	734
SET_ROTATION	736
SET_GRAREA	739
SET_DOT	741
DISTANCE	743
FILL_AREA	745
5 Tiger-BASIC® für Fortgeschrittene	751
Allgemeines zur Software-Entwicklung:	751
Dokumentation	751
Symbolische Werte verwenden	752
Systemdefinitionen	754
Variablendefinition	754
Variablen-Typen richtig benutzen	755
Format-Umwandlungen	758
Geschwindigkeit	760
Geschwindigkeit messen	760
PC-Mode	760
Numerik	760
Zwischenergebnisse	761
Strings	761
Programmschleifen	763
IN, OUT	764
Pulse	765
Funktionen	765
Multi-Tasking	766
6 Häufig gestellte Fragen - Support	771
Rat und Hilfe	775
7 Stichwortverzeichnis	779
8 Anhang	787
ASCII-Tabelle	787
EBCDIC Codes	788
Baudot-Code-Satz	789
ANSI-Steuersequenzen	790
Windows 95/98/NT Shortcuts	792
Short-Cuts Tiger-BASIC® Version 5	794
Kennzeichnung von Widerständen und Kondensatoren	796
Farbcodes	796

Inhaltsverzeichnis

Wertkennzeichnung durch Buchstaben	797
Toleranzkennzeichnung durch Buchstaben	798
Mittlere Schrittweite für Widerstands-Wachstum aufeinanderfolgender Werte:	800
Normreihen für Widerstandswerte	800
BASIC-Tiger [®] Modul A – Nutzung der Pins	807
TINY-Tiger [®] – Nutzung der Pins	811
TINY-Tiger [®] Modul E – Nutzung der Pins	815
BASIC-Tiger [®] -CAN-Modul – Nutzung der Pins	819

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

1 Einführung

Willkommen beim BASIC-Tiger[®] Entwicklungssystem.

Dieses Buch macht Sie mit der Programmierung des BASIC-Tigers[®] vertraut und ermöglicht Ihnen einen schnellen Start.

Lange Zeit galt die Programmierung von Microcontrollern als ein Bereich, der nur Spezialisten vorbehalten war. Die Programmierung fand meistens in Assembler statt, wodurch die Programme zwar schnell, aber schwer verständlich waren. Der Entwicklungsaufwand selbst kleinerer Projekte, hat schnell den Umfang von mehreren Monaten angenommen. Entschied man sich, in BASIC zu programmieren, so wurde die Verarbeitungsgeschwindigkeit relativ langsam und die Möglichkeiten waren eingeschränkt.

Der BASIC-Tiger[®] schließt nun diese Lücke. Durch den innovativen BASIC-Dialekt ist die Programmierung sehr einfach und verkürzt die Einarbeitungszeit. Die Verarbeitungsgeschwindigkeit des BASIC-Tigers[®] liegt sehr hoch und stellt damit so manchen in Assembler oder C programmierten Controller in den Schatten. Multitasking, das ohne komplizierten Overhead auskommt, führt zu besser strukturierten und leichter pflegbaren Programmen. Für immer wiederkehrende Programmieraufgaben steht eine wachsende Bibliothek von Funktionen und Beispiel-Applikationen zur Verfügung. Durch die Unterstützung von Gerätetreibern, die ständig weiterentwickelt werden, kann der BASIC-Tiger[®] mit dem Projekt wachsen und so auch Aufgaben wahrnehmen, an die bei Projektbeginn noch nicht gedacht wurde.

Wenn Sie direkt mit der Programmierung beginnen möchten, schlagen Sie im Handbuch "Hardware - Installation" nach und führen Sie die Installation durch, und folgen Sie direkt dem Kapitel „Blitzstart/Erste Schritte“.

Neue Instruktionen und Funktionen in der Version 5.0

Wenn Sie bereits erfahrener Tiger-BASIC[®]-Anwender sind, werden Sie sich sicher für die neuen Instruktionen und Funktionen interessieren. Sie finden hier eine kurze Übersicht über das Wichtigste, was neu ist seit dem letzten Up-to-Date. Neuigkeiten zu Hardware und Device-Treibern finden Sie in den entsprechenden Handbüchern. Unter der nächsten Überschrift 'Was ist nicht ganz so neu?' finden Sie die Neuerungen des vorigen Up-to-Dates.

Wichtiger Hinweis: Behalten Sie Ihre vorige(n) Tiger-BASIC[®]-Versionen. Durch ein größeres Run-Time-Modul kann es vorkommen, daß vorhandene Projekte, wenn Sie neu kompiliert werden, nicht mehr in das bisher verwendete Modul hineinpassen (Siehe dazu neue Präprozessor-Direktive ‚project_model‘). Es ist auch nicht ausgeschlossen, daß sich die eine oder andere Funktion vielleicht durch Beseitigen eines Bugs etwas anders verhält als vorher.



Insbesondere als wertvolle Ergänzung zu dem Grafik-Kapitel dieses Handbuches befinden sich unter der Überschrift Applikationen im Device-Treiber-Handbuch die Beispiele zum BASIC-Tiger[®] Graphic-Toolkit. Das Touch-Panel als Tastatur ist dort ebenfalls beschrieben.

Es gibt sehr viele neue Funktionen, die die Themen Verschlüsselung, TAN-Nummern und den besseren Zugriff auf die erweiterten I/O-Ports betreffen.

Die Trennung zwischen Instruktionen und Funktionen besteht in Tiger-BASIC[®] zwar weiterhin, jedoch sind sowohl Instruktionen als auch Funktionen in einem alphabetischen Teil zusammengefaßt. Um die Orientierung zu erleichtern, gibt es eine Übersicht, die ähnliche Befehle in Rubriken auflistet.

Übersicht über neue Funktionen:

BIT_DEMIX\$	92
BIT_MIX\$	97
BYTE_DEMIX\$	108
BYTE_MIX\$	114
CODE_1TON\$	129
CUT_AND_PASTE	142
CUT_AND_PASTE\$	144
DEBOUNCE	148
DECODE	151
DELETE_PROG	164
DISTRIBUTE\$	173
INDEX_1D	233

INDEX_2D	243
LET\$	267
LIMIT	268
MAX_LEN	286
PARITY	311
POKEM_FLASH	322
PRIME	325
RESTART_PROG	350
SCAN_TO_CHAR\$	361
SCAN_TO_CHAR_SETUP	363
SCAN_TO_TOGGLE	365
SCRAMBLE\$	366
SERIAL_NO\$	372
SET_LEN\$	375
SET_SERIAL_NO	376
SET_SYSVARN	379
TAN_CHK	473
TAN_COUNT	477
TAN_DEL	479
TAN_FIRST	483
THIS_VERS	487
XBUS_INR\$	530
XBUS_OUTR	532
XIN\$	534
XOUT	539
XSETUP	541

Einige Funktionen, darunter auch bereits ältere, haben mehrere Beispiele bekommen. Dazu zählen z.B. Varianten in der Schreibweise, wenn ein Parameter ein String oder eine Flash-Adresse sein kann.

Ältere Projekte können weiterhin mit dem kleineren Laufzeitsystem compiliert werden. Dazu dient die Direktive **#project_model**. In der Übersicht vor dem alphabetischen Teil der Beschreibung sind die Funktionen, die nur im Projektmodell ‚PM_FULL‘ zur Verfügung stehen, mit einem Punkt in der mittleren Spalte gekennzeichnet.

Was ist nicht ganz so neu?

Sie finden hier eine kurze Übersicht über das Wichtigste, was bei dem letzten Up-to-Date neu hinzugekommen ist.

Es gibt sehr viele neue Funktionen, unter anderem Funktionen, die sich auf Signalverarbeitung beziehen und Funktionen, die Grafikbearbeitung unterstützen. Zum Thema Grafik gibt es ein eigenes Kapitel, in dem die Grafikfunktionen beschrieben sind.

Übersicht über neue Funktionen im alphabetischen Teil der Funktionsbeschreibung:

ADD\$	75
AND\$	77
ANIB_BYTE\$	79
ASC_BCD\$	82
ASC_PBCD\$	84
BCD_ASC\$	88
BCD_PBCD\$	90
BIT_MIRR\$	95
BYTE_ANIB\$	106
BYTE_INIB\$	110
CALC_CRC	118
CUT_BITS	146
INIB_BYTE\$	248
MODULO_INC	290
MODULO_UPDO	293
OR\$	307
PARITY	311
PACK_BYTE\$	313
PBCD_ASC\$	315
PBCD_BCD\$	317
QUICK_WORD_COS	340
QUICK_WORD_SIN	342
SHIFTL\$	393
SHIFTM\$	396
SIGNAL_ADD	399
SIGNAL_AVERAGE	401
SIGNAL_BIGGER	403
SIGNAL_CUT_AMP	404
SIGNAL_CUT_MAX	406
SIGNAL_CUT_MIN	408
SIGNAL_GAP_END	409
SIGNAL_GAPI_END	411
SIGNAL_MAX	413
SIGNAL_MAX_AMPL	415

SIGNAL_MIN	417
SIGNAL_MUL	419
SIGNAL_NEXT_GAP	421
SIGNAL_NEXT_GAPI	423
SIGNAL_OFFSET	425
SIGNAL_SCALE	426
SIGNAL_SMALLER	428
SIGNAL_SMOOTH	429
UNPACK_BYTE\$	494
XOR\$	537

Übersicht über neue Grafikfunktionen:

AND2\$	589
AND3\$	589
AND4\$	589
OR2\$	589
OR3\$	589
OR4\$	589
XOR1\$	589
GRAPHIC_MASK_COPY	600
GRAPHIC_MIRROR	604
GRAPHIC_COPY	607
INVERT	627
GRAPHIC_FILL_MASK	629
GRAPHIC_EXP\$	638
DRAW_LINE	645
DRAW_NEXT_LINE	727
CLOSE_LINE	729
SET_BASE	731
SET_SCALE	734
SET_ROTATION	736
SET_GRAREA	739
SET_DOT	741
DISTANCE	743
FILL_AREA	745

Systemvoraussetzungen

Um mit der Entwicklungsumgebung des BASIC-Tigers[®] arbeiten zu können, benötigen Sie einen IBM-kompatiblen Rechner mit folgender Mindestausstattung:

- Pentium-Prozessor 233MHz
- Festplatte mit mind. 100 MByte freiem Speicher
- VGA-Grafikkarte
- 16 MB Arbeitsspeicher
- Maus
- Windows 95/98/2000/NT
- ein freier COM-Port
- CD-ROM-Laufwerk

Sicherheitshinweis

Die in diesem Entwicklungspaket enthaltenen Komponenten wie Plug & Play Lab, Prototyping-Boards, Toolkits, Adapter, Kabel etc. dienen nur als Arbeitshilfe bei der Entwicklung und Erprobung von Computer-Software und Hardware-Schaltungen. Ihr Zweck ist, ausschließlich dem sachkundigen Entwickler bei der Anfertigung von Laboraufbauten Zeit zu ersparen und Anregungen für eigene Designs zu geben.

In keinem Fall dürfen diese Komponenten von Nicht-Fachleuten installiert oder betrieben werden. Ebenso ist es nicht vorgesehen, daß derartige Laboraufbauten zur Steuerung von Anlagen und Geräten eingesetzt werden, insbesondere, wenn deren Fehlfunktion zu Schäden oder Gefahren führen können. In keinem Fall dürfen mit diesen Laboraufbauten hohe Spannungen oder Ströme gehandhabt werden.

Bei allen Änderungen an Schaltungen sowie beim Öffnen von Gehäusen, sind die Geräte bzw. Versuchsaufbauten vollständig von der Stromversorgung zu trennen.

Empfindliche Bauteile wie MOS Schaltkreise sind zur Vermeidung von Beschädigung nur in einer antistatisch ausgestatteten Laborumgebung zu handhaben.

Typografische Konventionen und Symbole

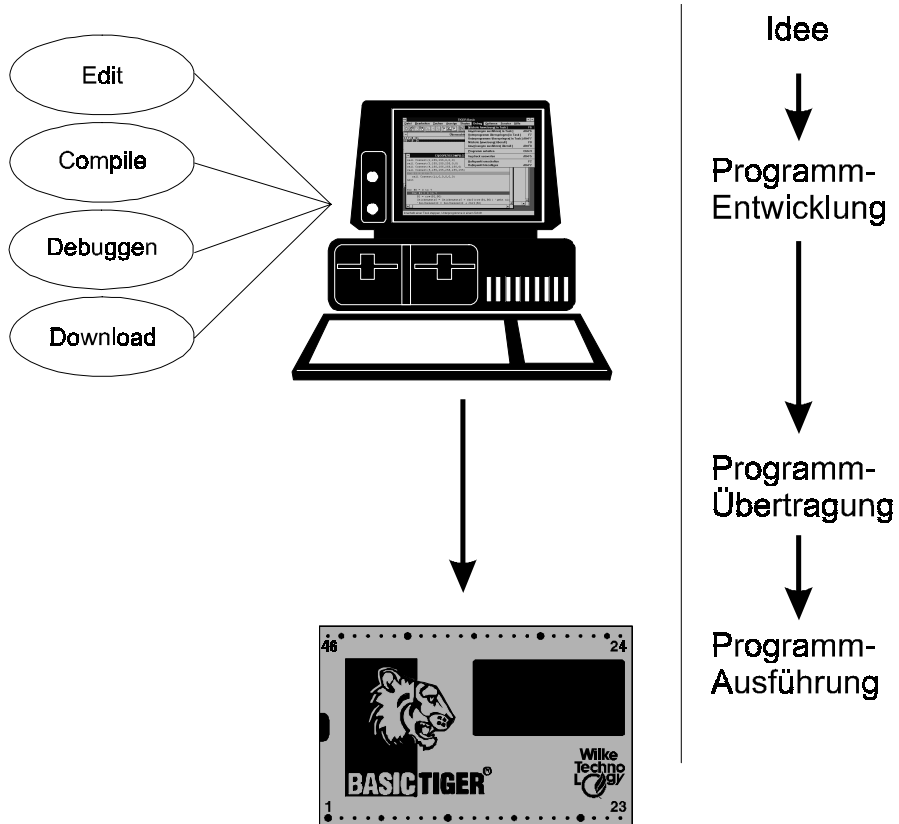
Die folgenden Schriften und Symbole wurden verwendet, damit Sie wichtige Informationen schnell erkennen können:

Element	Bedeutung
TASTE	Tastenbezeichnungen, z.B. RETURN
<i>Programmlisting</i>	Tiger-BASIC-Programmlisting
Instruktion	Tiger-BASIC®-Instruktion
<i>Variable</i>	Platzhalter für Elemente, die Sie Ihrer Anwendung gemäß eingeben müssen.
[]	Elemente, die optional eingegeben werden können.
!	Wichtige Anmerkung: bitte beachten!
Tip	Tips und Hinweise, die Ihnen die Arbeit erleichtern.

BASIC-Tiger®

In diesem Kapitel erfahren Sie etwas über die Besonderheiten des BASIC-Tigers®.

1



Multitasking

Die auffälligste Besonderheit des BASIC-Tigers[®] ist seine Multitasking-Fähigkeit. Obwohl BASIC-Tiger[®]-Module nicht viel größer sind als ein CPU Chip, ist ein kompletter Multitasking Steuercomputer enthalten, der über eigenen Programmspeicher (FLASH), Arbeitsspeicher (SRAM + FLASH) sowie eine Reihe von Standard-I/Os verfügt. Mehrere Tiger-BASIC[®]-Programme (Tasks) können in den Programmspeicher der Tigers geladen werden und bleiben dort, ähnlich der Festplatte eines PC, dauerhaft gespeichert, bis sie durch neue Programme überschrieben werden. Der FLASH-Speicher läßt sich ferner auch als dauerhafter Speicher für Daten verwenden, der von BASIC-Programmen aus beschrieben, gelesen und gelöscht werden kann. Der Arbeitsspeicher kann bis zu mehreren MBytes SRAM betragen und läßt sich ebenfalls gegen Stromausfall sichern.

Der Vorteil von Multitasking wird sofort offenkundig, wenn man sich reale Aufgabenstellungen für einen Steuercomputer ansieht. Selten besteht eine Anwendung nur aus einer monolithischen, einzelnen Aufgabe, die linear in einer großen Schleife abgearbeitet wird. Schon bei kleinen Anwendungen gibt es meist 3, 4, 5 oder mehr eigenständige Aufgaben zu erledigen, die weitgehend unabhängig voneinander abzuarbeiten sind. Man denke nur an Ausgaben auf einen Drucker, Eingaben von Keyboards oder seriellen Eingängen etc., die Anwendungen oft „hängen“ lassen können. Um solche Zustände zu vermeiden ist oft einiger zusätzlicher Programmier- und Testaufwand erforderlich. Entsprechend schwerer verstehbar und pflegbar werden solche Programme.

Programmiert in Multitasking, bleibt nichts so leicht hängen. Eingaben, Ausgaben, abgeschlossene Steuerungen oder Auswertungen werden in eigenständigen Tasks abgearbeitet. Ist z.B. eine rechenintensive Auswertung noch nicht abgeschlossen, so werden dennoch erforderliche Steuersignale generiert, der Dialog mit einer Schnittstelle fortgeführt, Informationen auf Displays aufgefrischt und Bedienungstasten überwacht. Derartige Multitasking-Programme laufen nicht nur schneller und zuverlässiger ab, sie sind auch leichter zu pflegen und bieten mehr Übersicht. Zusätzliche Aufgaben können auch später noch leicht als neue Tasks hinzugefügt werden. Der Leistungsbedarf der einzelnen Aufgaben läßt sich durch Priorisierung der Tasks fein austarieren, Kontroll-Tasks können wichtige Funktionen im Auge behalten und ggf. Notprogramme starten und Alarmer auslösen.

Die Programmierung in Multi-Tasking ist beim BASIC-Tiger[®] denkbar einfach und bereits mit wenigen BASIC-Zeilen zu realisieren. Einige einfache Beispiele finden Sie Hardware- und Installationshandbuch unter der Überschrift 'Blitzstart/Erste Schritte'.

Device-Treiber

Durch die Verwendung von Device-Treibern, die die gerätespezifischen Eigenschaften von Peripheriegeräten berücksichtigen, erreicht der BASIC-Tiger[®] ein hohes Maß an Flexibilität und Leistungsfähigkeit bei einfacher Handhabung. Egal wie unterschiedlich verschiedene I/O-Geräte auch sein mögen, stets werden I/O-Kanäle, die

mit Device-Treibern arbeiten, über die 6 standardmäßigen BASIC-Instruktionen **PRINT**, **PRINT USING**, **PUT**, **INPUT**, **INPUT LINE** und **GET** angesprochen. Die systematische Ansteuerung mit einer Device-Nr. (Device=Gerät) sowie optionaler Sekundär-Adresse und Funktions-Code gestattet einen systematischen, klar verstehbaren Programmaufbau. Die Umleitung von I/O auf alternative Kanäle, das Hinzufügen weiterer I/O-Kanäle und der Übergang zu anderer Hardware wird denkbar einfach. Eine I/O-Instruktion wie: „**PUT #PUMPE, LEISTUNG4**“ leitet den unformatierten Output der Variablen „**LEISTUNG4**“ auf das Gerät „**PUMPE**“, der in der physikalischen Realität aus ganz unterschiedlichen Kanälen bestehen kann: z.B. aus einem asynchronen seriellen Kanal, aus einem PWM-Ausgang, einer parallelen Schnittstelle oder einem ganz anderen Typ.

Die physikalischen Eigenschaften eines I/O-Devices werden weitgehend im Device-Treiber definiert und durch die Instruktion **INSTALL_DEVICE #Nr, Name** dem BASIC-Programm verfügbar gemacht. Um eine Ein- oder Ausgabe auf ein physikalisch anderes Gerät zu leiten, ist lediglich ein anderer Device-Treiber zu wählen oder ein Parameter in der **INSTALL_DEVICE**-Instruktion zu ändern.

Es ist die Aufgabe der Device-Treiber dem Programmierer das Leben zu erleichtern. An Stelle umfangreicher eigener Programmierungen zur Ansteuerung von I/O-Geräten kann sich der Programmierung auf die Bedienung des jeweiligen Gerätes konzentrieren. Übertragungsspezifische Details wie Puffer verwalten, Strobe-Signale erzeugen und auswerten, physikalische Adressen und Zeitverhalten handhaben, werden vom Device-Treiber übernommen. Der augenblickliche Satz von Device-Treibern wird laufend erweitert. Für spezielle Anforderungen können spezielle Treiber entwickelt werden.

Als ein Beispiel für die Vereinfachung durch Device-Treiber kann man sich den Treiber 'LCD1.TDD' ansehen, der die Ansteuerung eines LCD-Displays und einer Tastatur-Matrix mit bis zu 128 Tasten, Shift-LED und Piepser übernimmt. Der Treiber verwaltet nicht nur die normale Ansteuerung dieser Geräte einschließlich gepufferter Ein- und Ausgabe, es stehen auch eine Reihe leistungsfähiger ESC-Sequenzen zur Verfügung, mit denen man z.B. Tasten-Codes, Wiederholrate, Tasten-Klick, Attribute, Sonderzeichen etc. individuell einstellen kann. In seiner Gesamtheit ersetzt dieser Treiber mehrere 1000 Zeilen BASIC-Code.

Der Einsatz dieses Treibers wird u.a. in den Applikations-Programmen 'ANA1_DEM.TIG' und 'LCD_SON2.TIG' veranschaulicht. (siehe Device-Treiber-Handbuch).

Funktionen

Die ständig wachsende Bibliothek von Funktionen bildet ein mächtiges Tool, um Programmieraufgaben effektiv und mit wenigen Instruktionen zu realisieren. BASIC-Tiger[®]-Funktionen decken die Bereiche

- Integer Arithmetik
- Floating-Point Arithmetik
- String-Operationen
- Spezial-Funktionen

ab. Immer wiederkehrende Programmieraufgaben können auf fertige Funktionen zurückgreifen, die hohe Verarbeitungsgeschwindigkeit und kompakten Code erreichen.

Die 32-Bit-Integer-Arithmetik des BASIC-Tigers zeichnet sich durch hohe Geschwindigkeit und eine für viele Anwendungen bei weitem ausreichende Genauigkeit aus. Bei entsprechender Abbildung kann mit dem verfügbaren Wertebereich von -2.147.483.648 ... + 2.147.483.647 z.B. eine Skala von -20.000 ... +20.000 mit einer Auflösung von 0,000 01 (= 1/100.000) aufgebaut werden! Dies können Meßwerte wie Temperatur, Druck, Weg, Geschwindigkeit und vieles mehr sein, wie sie in Forschungsprojekten und Steuerungsaufgaben immer wieder vorkommen. Zu den Funktionen im Bereich Integer Arithmetik zählen u.a EXP, LD, MOD, SGN, ABS, RND, BIT, MASK, IMASK, LREAL, HREAL, LLTOR, LEN, LEN_FIFO, FREE_FIFO, READ_FIFO, etc.

Floating-Point-Arithmetik arbeitet im BASIC Tiger mit doppelter Genauigkeit (15-16 relevante Stellen), was auch hohe, wissenschaftliche Ansprüche abdeckt. Zudem stehen eine Reihe wichtiger Funktionen für komplexe Berechnungen zur Verfügung, u.a. SIN, COS, TAN, COT, ASIN, ACOS, ATAN, ACOT, SINH, COSH, TANH, COTH, LOG, LN, EXP, EXPE, SQRT, etc.

Bei den String-Funktionen wurde zu dem üblichen Funktionsumfang wie CHR\$, LEFT\$, RIGHT\$, MID\$, etc. eine Reihe starker Funktionen hinzugefügt, die eine sehr elegante und schnelle Programmierung anspruchsvoller Aufgaben erlauben. Hierdurch ist es möglich, Variablen vom Typ String in einem weit allgemeineren Zusammenhang einzusetzen, als dies traditionellerweise der Fall wäre. Insbesondere das Suchen, Selektieren, Ersetzen, Füllen, Fragmentieren und Konvertieren ist mit den neuen String-Funktionen ebenso schnell programmiert, wie diese Programme durch hohe Ablaufgeschwindigkeiten beeindrucken. Zu den neuen String-Funktionen zählen u.a. UPPER\$, CONVERT\$, NTO\$, RTO\$, STOS\$, NFROM\$, RFROM\$, SELECT\$, INDEX\$, REMOVE\$, REMDOUBLE\$, STRI\$, etc.

Und schließlich gibt es spezielle Funktionen aus dem systemnahen Bereich die diverse Status-Informationen, Prozeßzeit, Versions-Nr, Fehler-Informationen etc. liefern.

Einführung	1
Die Tiger-BASIC®-Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger®	4
Tiger-BASIC® für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

2 Die Tiger-BASIC®-Sprache

Sowohl die BASIC-Tiger® Computer-Module als auch die Tiger-BASIC® Sprache und Entwicklungsoberfläche sind von Wilke Technology entwickelt und aufeinander abgestimmt worden. Anwendungen mit BASIC-Tiger® Computer-Modulen werden mit einer komfortablen Entwicklungsumgebung unter Windows erstellt. Die verwendete Sprache Tiger-BASIC® ist abgestimmt auf die Belange eines Prozeßrechners. Immer wiederkehrende Aufgaben sind als fertige Instruktionen oder Funktionen realisiert, Device-Treiber und Beispiel-Applikationen beschleunigen den Software-Entwicklungsprozeß.

Vom Eingeben des Programms über das Fehlersuchen bis hin zum Programmieren der Module findet alles in dieser Entwicklungsumgebung statt. Tiger-BASIC®-Programme haben die Namensweiterung '.TIG'. Die in diesem Handbuch aufgeführten Beispiele finden Sie im Unterverzeichnis 'BEISPIEL' in Ihrem TIGER-Verzeichnis. Die meisten Beispiele sind sofort ausführbare, vollständige Programme und können direkt geladen, kompiliert, übertragen und in einem BASIC-Tiger® Modul ausgeführt werden.

Struktur von Tiger-BASIC®-Programmen

Da Tiger-BASIC® ein Multitasking-BASIC ist, kann eine Anwendung mehr als nur ein Programm im herkömmlichen Sinn enthalten. Die Einzelprogramme werden hier Tasks genannt. Jede Task ist zunächst ein vollkommen eigenständiges Programm das nach Belieben gestartet oder gestoppt werden kann und das ganz unabhängig von den übrigen Tasks ablaufen kann. Tasks können jeweils eigenständige Aufgaben erledigen oder gemeinsam an einer größeren Aufgabe zusammenarbeiten. Tasks können sich gegenseitig beeinflussen und miteinander kommunizieren. Im einfachsten Fall besteht eine Task aus einigen wenigen Instruktionen, sie kann aber auch beliebig umfangreich werden und sich über tausende von Codezeilen erstrecken.

Schreibweise

Tiger-BASIC® Source-Programme bestehen aus reinem ASCII-Text und werden mit dem Editor der Entwicklungsoberfläche erstellt. Ebenso können reine ASCII-Texte von anderen Editoren übernommen werden.

Bei Symbolen und Schlüsselwörtern wird Groß- und Kleinschreibung nicht unterschieden. Man kann hier bei der Notation seiner eigenen Vorliebe folgen und z.B. sehr lange Namen durch Einsatz von Groß-/Kleinschreibung besser lesbar gestalten: „AntriebXAchse“ = ANTRIEBXACHSE. Ausnahmen hiervon sind in der Beschreibung der Instruktionen aufgeführt.

Instruktionen, die aus zwei Worten bestehen, können auch mit '_' statt Leerzeichen geschrieben werden (RUN TASK = RUN_TASK). Instruktionen und Argumente sind durch mindestens ein Leerzeichen oder Tabulatorzeichen getrennt. Pro Zeile

Tiger-BASIC®-Sprache

kann jeweils 1 BASIC-Instruktion mit Kommentar enthalten sein. Überlange Zeilen lassen sich auf mehrere Zeilen verteilen.

Überlange Zeilen

2

RETURN oder das Kommentar-Zeichen signalisieren das Ende einer BASIC-Instruktion. Solche Instruktionen, die sehr lange Zeilen erforderlich machen würden, kann man mit dem Zeichen '&' auf mehrere Source-Code Zeilen verteilen. Dadurch läßt sich oft die Übersichtlichkeit im Listing verbessern und auch extrem lange Anweisungen sind möglich. Als Beispiel finden Sie hier die Besetzung eines Strings, der zur Erzeugung großer Zeichen auf dem LC-Display benötigt wird:

```
PRINT #1, "&  
Dieser Text wird als eine Zeile & ` Kommentar  
auf dem Ausgabekanal ausgegeben"
```

Und so drucken Sie das Zeichen '&' und das Anführungszeichen, indem Sie den Code angeben:

```
PRINT #1, "Firma XYZ <26H> Co." ` Firma XYZ & Co.  
PRINT #1, "Anführungszeichen: <22H>"
```

Kommentare

Ein Kommentar wird mit dem Apostroph eingeleitet und endet mit dem Ende der Zeile. Kommentarzeilen können nicht mit '&' gebrochen werden.

Die Task MAIN

Jede Tiger-BASIC®-Anwendung besteht aus mindestens einer Task. Dies ist die Task **MAIN**. Bei Bedarf können weitere Tasks sowie Unterprogramme hinzukommen. Die Menge der möglichen Tasks und Unterprogramme ist vom Modultyp und seiner Speicher-Ausstattung abhängig.

Nach dem Einschalten eines Tiger-Moduls oder nach einem Hardware-Reset wird automatisch die Task **MAIN** gestartet (im RUN-Mode). Wenn weitere Tasks ablaufen sollen, sind sie durch eine bereits laufende Task (zB: MAIN) zu starten. Mit Ausnahme der Task MAIN kann jede Task nach Belieben gestartet und gestoppt werden. Alle Tasks können nach Belieben Unterprogramme aufrufen. Ein Unterprogramm kann auch gleichzeitig von mehreren Tasks verwendet werden.

Die Task-Definition beginnt mit der Anweisung **TASK** und dem Namen der Task und endet mit **END**. In der Task definierte Variablen besitzen nur innerhalb dieser Task Gültigkeit. Globale Variablen werden außerhalb der Task **MAIN** definiert. Da die Task **MAIN** automatisch und stets als die erste Task nach dem Einschalten ausgeführt wird, sollte man gleich zu Beginn der Task **MAIN** alle für das Projekt relevanten Initialisierungen durchführen. Dazu zählen:

- Einbinden wichtiger INCLUDE-Dateien
- Initialisierung von globalen Variablen (soweit gewünscht)
- Definition von FIFO-Puffern (stets in Task MAIN, dennoch GLOBAL !)
- Installation aller benötigten Device-Treiber
- Setzen der Task-Prioritäten
- Start der anderen Tasks

Device-Treiber In der Task **MAIN** werden alle benötigten Device-Treiber eingebunden werden. In anderen Tasks oder Unterprogrammen ist dies nicht mehr möglich. Die Programmzeilen mit den BASIC-Instruktionen 'INSTALL_DEVICE' dürfen während der gesamten Programm-Ausführung nur einmal ausgeführt werden.

Globale Variable werden außerhalb der Tasks deklariert.

Lokale Variable Variable, die innerhalb einer Task oder eines Unterprogramms deklariert werden, sind lokale Variable, die nur innerhalb dieses Geltungsbereichs angesprochen werden können. Bei Namensgleichheit von lokalen und globalen Variablen wählt der Compiler stets die hier gültige lokale Variable. Die Benutzung von globalen Variablen in Unterprogrammen, als Ein-/Ausgangs-Parametern von Unterprogrammen und in den verschiedenen Tasks ist sorgfältig zu planen. Auch ist in Multi-Task-Umgebungen zu berücksichtigen, daß unter Umständen andere Tasks zur gleichen Zeit mit den gleichen globalen Variablen arbeiten.

FIFO-Puffer Als Besonderheit sind FIFO-Puffer per Definition immer globaler Natur, obwohl sie in der MAIN-Task deklariert werden.

Tip Der Tiger-BASIC®-Compiler bietet als eine Erleichterung in kleinen Projekten eine automatische Typ Deklaration für Variablen an. Man kann also Variablen einfach in BASIC-Instruktionen benutzen, ohne sie vorher explizit deklariert zu haben. Der Compiler setzt dann je nach Kontext den Variablen-Typ LONG, REAL oder STRING fest. Dies ist bequem und kann für schnell erstellte kleine Programme ganz hilfreich sein.

Sobald Aufgabenstellung und Programmcode umfangreicher werden, empfiehlt es sich dringend, jede Variable explizit zu deklarieren. Durch die bewußt vorgenommene Deklaration schafft man sich die beste Klarheit über Typ und Geltungsbereich der

Tiger-BASIC®-Sprache

Variablen und vermeidet Fehler, die sich leicht durch falsche Variablen- und Konstanten-Benutzung einschleichen können.

Durch die Compiler-Anweisung: `USER_VAR_STRICT` (Seite 31) wird der Compiler angehalten, die Variablen-Deklaration und Konstanten-Benutzung strikt zu überprüfen. Nicht deklarierte Symbole werden dann als Fehler gemeldet.

2

Die Typ-Deklaration allein weist Variablen noch keinen Wert zu. Dies ist von Bedeutung in Systemen mit Batterie-Backup für den RAM, wo Variablen noch die gleichen Werte enthalten (nur globale und Task lokale Variable), die sie vor dem Power-Down des Systems besaßen. Lokale Variable können niemals durch Batterie-Backup erhalten bleiben! Gibt es keinen geschützten RAM-Speicher, sind die Variablen entsprechend zu initialisieren.

Namen in Tiger-BASIC®-Programmen können bis zu 32 Zeichen lang werden und neben Buchstaben auch Ziffern und Underline-Zeichen enthalten. Groß- und Kleinschreibung im Namen hat keine unterscheidende Wirkung, so daß die Schreibweise so gewählt werden kann wie sie dem Programmierer jeweils am übersichtlichsten erscheint. Bei der Namensvergabe von Konstanten und Variablen haben sich u.a. folgende Methoden bewährt - eine Ideenliste:

- Je größer der Geltungsbereich einer Variablen um so ausgeprägter der Name
- Namen von globalen Variablen werden nicht als lokale Variablennamen benutzt
- Der Variablen-Typ wird durch den 1. Buchstaben des Namens gekennzeichnet, z.B:
 bEmpfang = BYTE
 wCount = WORD
 lAbsolut_X = LONG
 rWinkel_Geschwindigkeit4 = REAL
- Lange Namen werden optisch aufbereitet: „HauptSchleusenEingang“ oder „Haupt_Schleusen_Eingang“ statt HAUPTSCHLEUSENEINGANG

Nach dem Einbinden der Variablendeklaration und der Gerätetreiber folgen die weiteren Instruktionen des Programms. Die Task-Definition endet mit der Anweisung **END**. Tasks können nicht verschachtelt werden.

Ein typischer Programm-Anfang eines kleineren Projektes könnte also etwa so aussehen:

```
USER_VAR_STRICT      ' Variable muessen deklariert werden!

STRING  Z0$ (7), Z1$ (7)  ' Globale Strings fuer
STRING  Z2$ (7), Z3$ (7)  ' LCD Cursor-Positionen

#define LCD 1            ' Device-Driver-Nr fuer LCD
#define ANALOG 4          ' Device-Driver-Nr fuer ANALOG-In
```

```

#include KEYB_GR2.INC      ' Unterprogramme und Strings fuer
                           ' Dt. Tastatur auf Plug+Play-Lab

TASK MAIN
  STRING CH$(1)           ' Character (lokal)
  LONG S, N               ' Select, Loop (lokal)

  INSTALL DEVICE #LCD, "LCD1.TDD"      ' LCD-Display
  INSTALL DEVICE #ANALOG, "ANALOG1.TDD" ' Analog-Inputs (intern)

  CALL INIT_KEYB (LCD)      ' dt. Zeichensatz Plug & Play Lab

                           ' Werte initialisieren:
  Z0$ = "<1BH>A<00><0><F0H>" ' <- Cursor auf Zeile-0
  Z1$ = "<1BH>A<00><1><F0H>" ' <- Cursor auf Zeile-1
  Z2$ = "<1BH>A<00><2><F0H>" ' <- Cursor auf Zeile-2
  Z3$ = "<1BH>A<00><3><F0H>" ' <- Cursor auf Zeile-3

  SET_TASK_PRIO T2,123      ' Prioritaet der TASK T2
  SET_TASK_PRIO T1,42      ' Prioritaet der TASK T1
  RUN TASK T1              ' TASK T1 starten
  RUN TASK T2              ' TASK T2 starten

  ' hier folgen Programmzeilen der Task MAIN ...
END

```

Die Task **MAIN** wird nach Reset gestartet. Andere Tasks müssen explizit gestartet werden (RUN_TASK).

Alle Tasks außer MAIN sind gleichberechtigt und können prinzipiell jede andere Task starten, stoppen oder beenden.

Unterprogramme

Unterprogramme sind ein mächtiges Werkzeug um Programm-Abläufe strukturiert und übersichtlich zu gestalten. Mit der Übergabe von Ein- und Ausgangs-Parametern, lokalen Variablen, unbegrenzter Länge und der Fähigkeit von mehreren Tasks gleichzeitig benutzt zu werden, tragen Unterprogramme zur orthogonalen und staffen Programmierung bei.

Unterprogramme sind ähnlich wie eine Task aufgebaut:

- Beginn des Unterprogramms mit Nennung seines Namens und Auflistung der Parameter in Klammern
- Deklaration lokaler Variable
- Programm-Code des Unterprogramms
- END Anweisung signalisiert das Ende des Unterprogramm-Codes und hat gleichzeitig die Wirkung einer RETURN-Instruktion.

Tiger-BASIC®-Sprache

Jede Unterprogramm-Definition beginnt mit der Anweisung **SUB** und dem Namen des Unterprogramms.

Im Anschluß können bis zu 16 Ein- und Ausgangs-Parameter definiert werden. Das Schlüsselwort **VAR** signalisiert, daß die Werte der nachfolgenden Parameter dieser Liste im Unterprogramm verändert werden können. Parameter, die hier nur mit **BYTE**, **STRING**, etc. deklariert sind, werden als reine Eingangs-Parameter verwendet. Ihnen kann im Unterprogramm kein neuer Wert zugewiesen werden.

Es folgt die Deklaration der lokalen Variablen. Variable, die innerhalb eines Unterprogramms deklariert werden, sind stets lokal. Bei jedem Aufruf des Unterprogramms wird ein neuer Satz lokaler Variablen erzeugt. Dadurch ist gewährleistet, daß auch bei gleichzeitiger Verwendung aus verschiedenen Tasks heraus oder bei rekursivem Aufruf keine Konflikte zwischen den Variablen auftreten.

```
SUB TEST ( BYTE HOCH, BREIT; VAR STRING CHAR$.)
  BYTE J                \ lokale Variable
  CHAR$ = "A"           \ CHAR$ ist veraenderbar
  ....
END
```

Dann folgen die Instruktionen des Unterprogramms. Die Größe des Unterprogramms ist nur durch den vorhandenen Speicher begrenzt. Die Unterprogramm-Definition endet mit der Anweisung **END**, die sowohl das Ende des Unterprogramm-Codes anzeigt, als auch die Wirkung der **RETURN** Instruktion besitzt.

Das folgende Beispiel zeigt ein kleines Multi-Tasking-Programm mit zwei Tasks und einem Unterprogramm. Die Task **MAIN** zählt in einer leeren **FOR**-Schleife die globale Laufvariable "Count" hoch, während eine 2. Task einmal pro Sekunde den aktuellen Wert der Laufvariablen auf dem LC-Display anzeigt. Nach 10 Sekunden wird Task "Second" beendet und zeigt abschließend den Wert: "Loops / s" an.

Programmbeispiel:

```

' -----
' Name: 2TASKS.TIG
' -----
LONG Count                                ' globale Variable deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
RUN TASK Second                          ' Starte Task "Second"
FOR Count = 0 TO 9999999                 ' Anfang (leere) FOR-Schleife
NEXT                                     ' Ende (leere) FOR-Schleife
END                                       ' Ende Task MAIN

' -----
' Task Second: Zeigt 1x pro Sekunde die Var. "Count" auf LC-Display
' -----
TASK Second                              ' Beginn Task Second
LOOP 10                                 ' 10 Schleifen ausfuehren
    PRINT #1, "<1>sum loops :"; Count    ' zeige Laufvariable aus MAIN
    WAIT DURATION 1000                  ' 1 Sek warten
ENDLOOP                                 ' naechster Durchlauf
CALL Ready (10)                         ' Aufruf Unterprogramm
END                                     ' Ende Task Second

' -----
' Subroutine: Zeigt berechneten Wert "Loops pro Sekunde"
' -----
SUB Ready (BYTE d)                       ' Beginn Subroutine Ready
    PRINT #1, "Loops / s: "; Count / d   ' d (lokal), Count (global)
END                                       ' Ende Subroutine Ready

```

2

Datentypen

Tiger-BASIC® stellt folgende Datentypen zur Verfügung:

Typ	Wertebereich	Länge
BYTE	0...255	1 Byte
WORD	0...65535	2 Bytes
LONG	-2.147.483.648... 2.147.483.647	4 Bytes
REAL	$4.19 \cdot 10^{-307} \dots 1.67 \cdot 10^{308}$	8 Bytes
STRING	Alle Codes zulässig: 00...0FFh	0...65535 Bytes
DATALABEL	gibt Adresse im Flash-Speicher an	

Zahlen beginnen immer mit einer Ziffer und werden dezimal, oktal, hexadezimal, binär oder mit der Einheit Kilo(1024) geschrieben.

Beispiel	Bedeutung
31	dezimal
37o	oktal
1Fh	hexadezimal
0FFh	hexadezimal, die führende 0 unterscheidet die Zahl von der Variablen 'FFH'
11111b	binär
1k	=1024

BYTE

Bytekonstanten und Variable besitzen den Wertebereich $0 \leq x \leq 255$.

Byte Werte sind immer positiv und belegen 1 Byte Speicherplatz.

WORD

Word-Konstanten und Variable besitzen den Wertebereich $0 \leq x \leq 65535$.

Word-Werte sind immer positiv und belegen 2 Byte Speicherplatz

LONG

Long-Konstanten und Variable besitzen den Wertebereich
 $-2.147.483.648 \leq x \leq 2.147.483.647$.

Long-Werte sind immer vorzeichenbehaftet und belegen 4 Bytes Speicherplatz.

REAL

Real-Konstanten und Variable besitzen einen vorzeichenbehafteten Wertebereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$ (double precision). Es werden 15 bis 16 relevante Stellen berechnet. Real-Werte belegen 8 Bytes Speicherplatz.

Berechnungen mit REAL-Zahlen dauern länger als Berechnungen mit BYTE, WORD oder LONG-Zahlen. Eine Berechnung wird stets zuende geführt, bevor eine andere Task CPU-Zeit erhält. Dies ist zu beachten, wenn eine andere Task kurze Reaktionszeit erfordert.

Tip

Wenn es auf hohe Verarbeitungsgeschwindigkeit ankommt, lassen sich viele Aufgaben mit LONG-Zahlen erledigen, auch wenn Nachkommastellen gefordert sind. Der Wertebereich von: $-2.147.483.648 \dots +2.147.483.647$ kann z.B. abgebildet werden auf den Festkomma-Zahlenbereich: $-2000\,000,000 \dots +2000\,000,000$ mit 3 Nachkommastellen. Man rechnet einfach in $1/1000\,^{\circ}\text{C}$, oder $1/1000\,\text{mm}$ etc. Bei der Ausgabe auf einen Ausgabe-Kanal (mit PRINT_USING) kann ein entsprechender Formatstring diesen Sachverhalt wiedergeben (Formatstring siehe Seite 499).

Bei Zuweisungen und Berechnungen zur Laufzeit des Programms werden in der Regel automatisch Typumwandlungen vorgenommen. Bei Wandlung von einer LONG-Variablen zur WORD- oder BYTE-Variablen oder von einer WORD-Variablen zur BYTE-Variablen kann dabei der Wert verfälscht werden, wenn der Zahlenwert zu groß für die Variable ist oder bei den WORD- und BYTE-Variablen das Vorzeichen verlorengeht.

Während der Compilation erlaubt der Compiler solche Zuweisungen und schneidet die Variable entsprechend ab: so wird aus WORD 123h in der BYTE-Variablen 23h. Aus LONG -15 (FFFFFFEBh) wird in der BYTE-Variablen 0EBh.

STRING

Strings können 0...32765 Zeichen enthalten. Jeder Zeichencode von 00...0FFh ist zulässig. Stringvariablenamen enden stets mit '\$'. String-Konstanten stehen in Anführungszeichen. Durch ein '%' -Zeichen hinter dem String wird dessen Inhalt als Folge hexadezimaler Codes interpretiert. Die Codes dürfen dann übersichtlicher durch ein Leerzeichen getrennt geschrieben werden. Speziell für Pixelgrafiken können Strings in besonderer Form eingegeben werden: jedes '0'-Bit wird als Punkt und jedes '1'-Bit als '*' eingegeben. Der String wird mit einem 'b' oder 'B' unmittelbar hinter den abschließenden Anführungszeichen gekennzeichnet. Auch hier dürfen Leerzeichen verwendet werden, um eine eventuell vorhandene Verzerrung der Textbildschirmdarstellung gegenüber dem echten Grafik-LCD auszugleichen. Die Leerzeichen bedeuten dann weder 0 noch 1, also nichts.

In String-Konstanten sind Code-Angaben in Form von Hex- oder Dezimal-Codes möglich. So können nicht druckbare Zeichen in den String eingefügt werden. Es sind verschiedene Notationen möglich.

Das Zeichen '<' in String-Konstanten wird so eingegeben: '<<'.

Beispiele:

Beispiel	Bedeutung
"ABC"	ASCII-String
"414243%"	der gleiche ASCII-String, in hexadezimaler Codierung
"A<42h>C"	der gleiche ASCII-String, mit einem Hex-Code im String
"A<66>C"	der gleiche ASCII-String, mit einem Dezimal-Code im String
"Hallo Welt<13>"	Hallo Welt mit RETURN
"Hallo Welt<0DH>"	Hallo Welt mit RETURN
"48 61 6C 6C 6F 20 57 65 6C 74 13%"	Hallo Welt mit RETURN in Hex-Codes. Zur besseren Übersicht sind Leerzeichen erlaubt.
"<<<Hallo Welt><13>"	<Hallo Welt> mit RETURN
"..****..."B	ein Byte mit den Bits '001111000'

Die Standardlänge eines Strings ist 64 Bytes. Die Größe einzelner Strings kann durch Angabe der Größe in Klammern festgelegt werden, z.B. um mit dem Speicherplatz auf dem Stack sparsam umzugehen:

A\$(5)

Strings, die größer als 64 Bytes sein sollen, müssen entsprechend deklariert werden:

B\$(8000)

Die Standardlänge der Strings kann mit der Compiler-Anweisung

USER_STRING_SIZE Wert

auf einen anderen Wert gesetzt werden (Wert=1...32767).

2

DATALABEL

Um Flashdaten im Programm leicht zugänglich zu machen, können Sie Data-Marken (Data-Labels) verwenden. **Datalabels müssen in jedem Fall deklariert werden**, auch dann, wenn die Variablendeklaration nicht mit USER_VAR_STRICT erzwungen ist. Datalabels bezeichnen Stellen im User-Flash und ermöglichen symbolische Bezeichnungen für Flashbereiche. Beim setzen von Datalabels werden sie von Sprungmarken unterschieden, indem sie mit zwei Doppelpunkten enden.

```
DATALABEL DL1           ' deklarieren
DL1::                   ' Datalabel setzen
DATA LONG 1234567       ' zugehoeriger Wert
```

Programmbeispiel:

```
' -----
' Name: DATALABEL.TIG
' -----
TASK MAIN                ' Beginn Task MAIN
  STRING A$              ' STRING-Variable deklarieren
  DATALABEL DL0, DL1, DL2 ' drei Datalabels deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

DL0::                    ' Datalabel DL0 setzen
  DATA BYTE "Bytes at DL0" ' einen String ins FLASH
DL1::                    ' Datalabel DL1 setzen
  DATA FILE "C:\TIGERBAS\EXAMPLES\FLASH.TXT" ' Datei ins FLASH
DL2::                    ' Datalabel DL2 als Referenz

  PEEK_FLASH DL0, A$, DL1 - DL0 ' lese A$ aus FLASH pos. DL0
  PRINT #1, A$                  ' Ausgabe auf LCD
  PEEK_FLASH DL1, A$, DL2 - DL1 ' lese A$ aus FLASH pos. DL1
  PRINT #1, A$;                 ' Ausgabe auf LCD
END                             ' Ende Task MAIN
```

Zusammengesetzte Datentypen

Aus den einfachen Datentypen lassen sich zusammengesetzte Variable konstruieren, die mehrere Variablen des einfachen Typs enthalten.

Tiger-BASIC® kennt folgende zusammengesetzte Datentypen:

- ARRAY
- FIFO

ARRAY

Deklaration eines Arrays:

ARRAY Name (Anzahl1 [,Anzahl2,..., Anzahl8]) OF Typ

Ein Array ist ein Variablenfeld mit einer Anzahl Variablen eines einfachen Datentyps BYTE, WORD, LONG, REAL oder STRING. Beachten Sie, daß es beim Datentyp STRING heißt: ARRAY A\$(20) of STRING**S**, also STRING in der Mehrzahl!

Für die Namensgebung gelten die gleichen Regeln wie für einfache Variable.

Anzahlx gibt die mögliche Anzahl von Elementen in dieser Dimension an, die das Array aufnehmen kann.

Bei der Anwendung des Arrays steht an der Stelle Anzahlx der Index des Arrayelements. Der Index ist also ein Zeiger auf das Element.

Der Index des Arrays beginnt bei 0.

Tiger-BASIC® unterstützt Arrays mit 1...8 Dimensionen.

Der benötigte Speicherplatz für ein Array berechnet sich aus der Anzahl der Elemente * Platzbedarf eines Elements.

Der verfügbare RAM-Speicherplatz bestimmt die maximale Größe des Arrays.

Programmbeispiel:

```

-----
' Name: ARRAY.TIG
-----
ARRAY M$(12,2) OF STRINGS          ' globales Array mit 2 x 12
                                   ' Eintraegen (2 Dimensionen)

TASK MAIN                          ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8

CALL INIT_ARRAY                    ' Aufruf Unterprogramm
FOR J = 0 TO 1                    ' Index J laeuft von 0 bis 1
  FOR I = 0 TO 11                 ' Index I laeuft von 0 bis 11
    PRINT #1, M$(I,J)             ' Ausgabe von Eintrag I,J
    WAIT_DURATION 500             ' warte 500 ms
  NEXT I                          ' naechster Index I
NEXT J                            ' naechster Index J
END                               ' Ende Task MAIN

' _Wilke Technology GmbH • Tel: +49 (2405) 408 550 • Fax: +49 (2405) 408 554 44_
' Subroutine INIT_ARRAY besetzt Array M$ mit "Werten"
-----
SUB INIT_ARRAY ()                  ' Beginn Subroutine
  M$(0,0) = "Januar  "
  M$(1,0) = "Februar "
  M$(2,0) = "Maerz   "
  M$(3,0) = "April   "
  M$(4,0) = "Mai     "
  M$(5,0) = "Juni    "
  M$(6,0) = "Juli    "
  M$(7,0) = "August  "
  M$(8,0) = "September"
  M$(9,0) = "Oktober "
  M$(10,0) = "November"
  M$(11,0) = "Dezember "

  M$(0,1) = "january  "
  M$(1,1) = "february "
  M$(2,1) = "march    "
  M$(3,1) = "april    "
  M$(4,1) = "may      "
  M$(5,1) = "june     "
  M$(6,1) = "july     "
  M$(7,1) = "august   "
  M$(8,1) = "september"
  M$(9,1) = "october  "
  M$(10,1) = "november "
  M$(11,1) = "december "

END                               ' Ende Subroutine

```

FIFO-Puffer

FIFO Name (*Anahl*) OF Typ

Deklaration eines FIFO-Puffers (FIFO=First-In-First-Out):

2

FIFO-Puffer werden stets innerhalb der MAIN Task deklariert. Dennoch sind sie per Definition stets global. Die FIFO-Deklaration erzeugt Code, der zur Initialisierung des FIFO führt. Der Code muß vor der ersten FIFO-Operation ausgeführt sein.



FIFO-Puffer sind Warteschlangen-Puffer. Die zuerst hineingeschriebenen Werte werden auch zuerst wieder entnommen (First-In-First-Out-Puffer). FIFOs gleichen zum Beispiel unterschiedliche Datengeschwindigkeiten zwischen Daten-Quelle und Senke aus (z.B. Ein- / Ausgangs-Puffer für I/Os).

Eine interessante Instruktion für Filterung verrauschter und gestörter Meßwerte ist INTEGRAL_FIFO. Mit dieser Instruktion kann sehr einfach eine schnelle, fortlaufende Kurzzeit-Integration über eine setzbare Fenster-Weite vorgenommen werden. Eine Integrationsweite, die dem Vielfachen der Periode der Netzfrequenz entspricht vermag z.B. 50 Hz Einkoppelungen zu dämpfen. A/D-Wandler verwenden Oversampling und anschließende Kurzzeit-Integration dazu um die Wandler-Auflösung um einige Bits zu erhöhen.

FIFO-Puffer enthalten Elemente des Datentyps: BYTE, WORD oder LONG.

Anzahl gibt in 2er-Potenzen die Anzahl der Elemente an, die der FIFO-Puffer aufnehmen kann. Bei ungültigen Werten für **Anzahl** rundet der Compiler auf den nächstgrößeren zugelassenen Wert auf.

Der Zugriff auf die Elemente erfolgt über spezielle Instruktionen:

Instruktion	Seite	Beschreibung
FIFO	199	Deklaration
GET_FIFO	212	lesen
PUT_FIFO	336	schreiben
PUTU_FIFO	338	schreiben (unconditional)
INTEGRAL_FIFO	259	integrieren
READ_FIFO	259	lies Element im freien Zugriff. (Funktion)

Der Speicherplatzbedarf errechnet sich aus der

Anzahl der Elemente * dem Platzbedarf eines Elements.

Die Größe eines FIFO-Puffers reicht von 16 Byte bis 4 Mbyte.

Der verfügbare RAM-Speicherplatz bestimmt die max. Größe des FIFO-Puffers.

Variable initialisieren

In BASIC-Tiger®-Modulen mit Batteriepufferung behalten globale und Task-lokale Variable ihren Wert auch dann, wenn die Stromversorgung an VCC abgeschaltet wird. Beim ersten Einschalten oder nachdem die Batteriepufferung unterbrochen war oder auch nach einem Absturz z.B. durch Störimpulse **sind die Werte von Variablen nicht festgelegt**. Relativ harmlos für Tiger-BASIC® sind die numerischen Variablen, die einfach irgendwelche Werte enthalten (für Ihre Anwendung mag das eventuell fatal sein). Strings hingegen können unsinnige Längenangaben besitzen, z.B. können sie länger sein als die maximal mögliche Länge oder sogar eine negative Länge haben. Das ist – unabhängig von der Anwendung – auch für Tiger-BASIC® fatal und kann dazu führen, daß die ganze Anwendung nicht funktioniert. Es ist daher Aufgabe des Programm-Entwicklers, alle Variablen vor ihrer ersten Benutzung zu initialisieren. In größeren Programmen können durch versehentlich nicht-initialisierte Variable Fehler entstehen, die schwer zu finden sind und eventuell nur sporadisch auftreten. Wenn Variable nicht durch Akku-Pufferung erhalten bleiben müssen, dann können Sie auf Nummer sicher gehen, indem Sie die Compiler-Anweisung `USER_VAR_INIT` an den Anfang des Quelltextes setzen.

2

Programmbeispiel:

2

```
'-----
' Name: USER_VAR_INIT.TIG
' Initialisiere alle Variable vor der ersten Verwendung
' Compiler-Anweisung
'-----
USER_VAR_STRICT          ' Vars deklarieren!
USER_VAR_INIT             ' Initialisiere alle Variablen
                          ' vergleiche:
                          '   a) with USER_VAR_INIT
                          '   b) without USER_VAR_INIT

TASK MAIN
  REAL R1
  STRING A$               ' Variable, die Initialisiert werden
  LONG N                  ' durch USER_VAR_INIT

' LCD-Treiber installieren (BASIC-Tiger)
install_device #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' install_device #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8      IN

PRINT #1, "<1>";R1;" (";A$;") ";N'
R1=1.23
A$="abcd"              ' setze Variable auf definierte Werte
N=77                   ' und siehe den Unterschied nach
PRINT #1,R1;" (";A$;") ";N ' einem RESET mit u. ohne
                          ' USER_VAR_INIT
END
```

Siehe auch: USER_VAR_STRICT

Variable deklarieren

In Tiger-BASIC® brauchen numerische Variable und String-Variable vor ihrer ersten Benutzung nicht deklariert zu werden. String-Variable erkennt der Compiler an der Namensendung '\$'. Solche Strings werden automatisch mit der jeweils gültigen Standard-Stringlänge definiert (64Bytes, oder mit USER_STRING_SIZE gesetzt). Numerische Variable bekommen je nach Kontext den Typ LONG oder REAL zugewiesen. Diese automatische Typzuweisung ist für kleine, schnell geschriebene Programme ganz angenehm. In größeren Programmen können sich jedoch hierdurch leicht Fehler durch falsch benutzte Variable einschleichen, die dann schwer zu finden sind. Deswegen beherzigen Sie in größeren Projekten folgenden Tip:

Insbesondere in größeren Anwendungen empfiehlt es sich, jede Variable explizit zu deklarieren. Nur so kann der Compiler falsche Benutzung - nicht deklarierte Variablen und nicht definierte Konstanten - aufdecken und melden. Falsch benutzte Variablen können zu Fehlererscheinungen führen, die extrem schwer nachzuvollziehen sind.



Die Compiler-Anweisung **USER_VAR_STRICT** zwingt Sie dazu, Konstanten und Variable zu deklarieren. Nicht deklarierte Variable oder falsche Variablentypen bei der Übergabe in Unterprogramme werden als Fehler gemeldet. Setzen Sie die Anweisung **USER_VAR_STRICT** einfach an den Anfang Ihres Programms.

Durch das Deklarieren einer Variablen ist diese noch nicht initialisiert. Soll der Wert einer Variablen mit einem Startwert vorbesetzt werden, ist sie entsprechend zu initialisieren oder die Compiler-Anweisung **USER_VAR_INIT** anzuwenden. Beachte: In Batterie-Anwendungen mit Batterie-Backup ist die unbedingte Initialisierung von Variablen unsinnig, da hierdurch die geretteten Daten-Inhalte von Variablen überschrieben würden.

Gültigkeitsbereich von Variablen

Variable, die außerhalb einer Task oder eines Unterprogrammes deklariert werden, sind global und können von allen Tasks und aus allen Unterprogrammen heraus gelesen und verändert werden. Variable, die innerhalb einer Task oder eines Unterprogrammes deklariert werden, sind stets lokal. Lokale Variablen der Tasks arbeiten statisch und stehen in Systemen mit Batterie-gepuffertem RAM - ebenso wie alle globalen Variablen - auch nach einem Power-Down noch zur Verfügung. Lokale Variablen von Unterprogrammen sind stets dynamisch und können nicht durch ein Batterie-Backup gerettet werden.

Ausnahme: FIFO-Puffer sind stets global, obwohl sie innerhalb der MAIN-Task zu deklarieren sind.

Variablennamen

Variablennamen bestehen aus Buchstaben, Ziffern und Unterstrichen. Das erste Zeichen ist immer ein Buchstabe. Groß- und Kleinschreibung wird nicht unterschieden. Die maximale Länge eines Namens beträgt 32 Zeichen. Durch Groß- / Kleinschreibung und durch Unterstriche werden lange Namen besser lesbar gemacht.

Beispiele für Variablennamen:

```
A
A2
TASTE
Taste
Pumpe_A
Zylinder_3_Zuendung
Kleingeld_Erkennungs_Ergebnis
```

Operatoren

Mathematisch		Beispiel
+	addieren	A=A+1
-	subtrahieren	A=B-3
*	multiplizieren	A=B*C
/	dividieren	A=B/2
-(unary)		A=-B
Boolsche		Beispiel
AND	logisch UND	IF A=1 AND C=2 ...
OR	logisch ODER	IF A\$=""" OR B\$="2" ...
XOR	logisch exklusiv ODER	IF A=1 XOR B>C ...
NOT	logisch NICHT	IF NOT (A=1) ...
>	größer	IF A>B ...
>=	größer oder gleich	IF A>=10 ...
<	kleiner	IF A<3.2 ...
<=	kleiner oder gleich	IF A<=20 ...
<>	ungleich	IF A<>1 ...
=	gleich	IF A=B ...
Bitweise		Beispiel
BITAND	UND	A=B BITAND 000110011b
BITOR	ODER	A=A BITOR 1FH
BITXOR	Exklusiv ODER	A=B BITXOR C
SHL n	links schieben um n Bits	A=B SHL 4
SHR n	rechts schieben um n Bits	A=12 SHR 2
SHRA n	arithmetisch rechts schieben	A=B SHRA X
ROL n	links rotieren um n Bits	A=B ROL 4
ROR n	rechts rotieren um n Bits	A=B ROR X
String		
+	anhängen	A\$=A\$+B\$

Weitere Operationen werden durch Funktionen ausgeführt, z.B. BITNOT, BIT_MIRR, Wurzelziehen (SQRT), Potenzieren(EXP), Sinus und Cosinus-Funktionen etc., sowie die Funktionen, die ganze Strings bearbeiten: ADD\$, AND\$, BIT_MIRR\$, OR\$, XOR\$.

Prioritäten beim Rechnen

Die Prioritäten beim Rechnen sind von der höchsten zur niedrigsten Priorität:

- Klammern, Vorzeichen plus und minus
- Operatoren *, /, BITAND, alle Schiebe-Operationen
- Operatoren +, -, BITOR, BITXOR
- Vergleichsoperatoren <, >, <=>, <=, >=
- logisch AND
- logisch OR, XOR

Präprozessor

Der Präprozessor bearbeitet den Programmtext, bevor dieser vom Compiler übersetzt wird. Dies kann dazu benutzt werden um häufig verwendete, komplizierte Notationen durch einfachere und aussagekräftigere zu ersetzen. Ebenso werden üblicherweise konstante Zahlenwerte durch eine symbolische Schreibweise ersetzt. Mit dem Präprozessor können umfangreiche Files mit Standard-Definitionen und Unterprogramm-Bibliotheken eingebunden werden.

Die Präprozessor-Instruktionen beginnen mit '#'.

#include

#include *Dateiname*

Bindet an dieser Stelle eine andere Datei ein.

In verschiedenen Projekten benötigte gleiche Definitionen oder Programmabschnitte können so in getrennten Dateien gepflegt werden. Ein Projekt läßt sich in verschiedene kleinere Dateien aufteilen, die von einer Hauptdatei dann eingebunden werden.

Es wird nur für tatsächlich verwendete Unterprogramme Code erzeugt, so daß Unterprogrammsammlungen (Bibliotheken) eingebunden werden können.

Beispiel:

```
#INCLUDE KEYB_PP.INC      ' Unterprogramme und Strings fuer  
                          ' Dt. Tastatur auf Plug+Play-Lab
```

#define

#define *Text1 Text2*

Ersetzt alle im folgenden Text vorkommenden **Text1** durch **Text2**.

Text1 endet mit dem ersten Leerzeichen oder Tabulatorzeichen.
Text1 wird nicht ersetzt, wenn er in Anführungszeichen steht oder im Zusammenhang mit anderen Buchstaben.

Text2 darf auch Leerzeichen oder Tabulatorzeichen enthalten und endet mit dem Zeilenende oder Kommentarzeichen.

Ersetzungen können nur vorwärts im Quelltext vorgenommen werden, d.h. zum Zeitpunkt der Ersetzung muß die Zeile, die das entsprechende '#define' enthält, bereits durchlaufen sein.

Einige Beispiele für Ersetzungen mit '#DEFINE':

```
#DEFINE LCD 1           \ alle Texte 'LCD' werden durch '1' ersetzt
#DEFINE CLS CALL CLEAR_LCD \ 'CLS' wird ersetzt durch 'CALL CLEAR_LCD'
#DEFINE RINT 2          \ 'RINT' -> '2' aber nicht 'PRINT' zu 'P2'
```

2

#comment, #endcomment

#comment

Markiert den Beginn bzw. das Ende eines Kommentarbereichs. Verschachtelte Kommentarbereiche sind möglich. Die Anweisungen können kürzer so geschrieben werden:

```
#CM           \ Beginn des Kommentars
  PRINT #1, "abc" \ wird nicht compiliert, da Kommentar
#ENCDM        \ Ende des Kommentars
```

Ein Beispiel für verschachtelten Kommentar:

```
#CM           \ Beginn des Kommentars (a)
  PRINT #1, "abc" \ wird nicht compiliert, da Kommentar
#CM           \ Beginn des Kommentars (b)
  WAIT_DURATION 1000 \ wird nicht compiliert, da Kommentar
#ENCDM        \ Ende des Kommentars (b)
  PUT #2, "c" \ wird nicht compiliert, da Kommentar
#ENCDM        \ Ende des Kommentars (a)
```

#project_model

Das Laufzeit-Modul von Tiger-BASIC® nimmt im Zuge der Weiterentwicklung durch neue Funktionen an Umfang zu. Damit bereits bestehende Projekte weiterhin in die Module hineinpassen, für die Sie entwickelt wurden, besteht die Möglichkeit, zwischen einem kleinen und dem vollen Compilermodell zu wählen. Standardmäßig wird automatisch das volle Modell compiliert. Das gewünschte Modell kann jedoch aktiv ausgewählt werden:

Für minimales Compiler-Modell:

#project_model pm_min

Für volles Compiler-Modell:

#project_model PM_FULL

In der Übersicht sind die nur im vollen Projektmodell enthaltenen Instruktionen und Funktionen gekennzeichnet. Ebenso befindet sich die gleiche Kennzeichnung auf jeder Seite der detaillierten Beschreibung.

BASIC-Funktionen

Tiger-BASIC®-Funktionen liefern einen skalaren numerischen Funktionswert bzw. ein String-Ergebnis zurück. Funktionen stehen in Ausdrücken und Zuweisungen immer rechts von einem '='-Zeichen.

Funktionen, deren Name mit einem '\$'-Zeichen enden, liefern als Ergebnis einen String. Alle anderen Funktionen liefern entweder INTEGER (BYTE, WORD, LONG) oder REAL-Werte.

Konstanten, Variable oder Ergebniswerte erfahren automatisch eine Typwandlung, wenn dies für die Berechnung oder für die Zuweisung des Ergebniswertes erforderlich ist. Ungewollte Typumwandlungen kosten in jedem Fall zusätzliche Laufzeit und können die Genauigkeit des Ergebnisses beeinflussen. Von Vorteil ist, auf eine möglichst typenkonforme Verwendung von Variablen in Ausdrücken zu achten.

In der Regel dürfen anstelle von Variablen auch Konstanten oder Ausdrücke stehen.

```
CR$ = CHR$ ( 13 )      ` String CR$ enthaelt Return-Zeichen
Y = SIN ( X )          ` X und Y sind REAL-Zahlen
L_LONG = SIN ( X )     ` sinnlos, da SIN(X) immer kleiner als 1 ist
A = ASC ( CR$ )        ` A erhaelt als Wert Code von CR$
L_LONG = R_REAL / W_WORD ` automatische Typwandlungen finden statt
```

Compiler-Anweisungen

Compiler-Anweisungen stehen am Anfang des Quelltextes außerhalb von Task- oder Unterprogrammdefinitionen. Compiler-Anweisungen sind Kommandos, die den Ablauf der Compilation beeinflussen, sie erzeugen jedoch keinen unmittelbar ausführbaren BASIC-Code für das BASIC-Tiger®-Modul. Compiler-Anweisungen sind erkennbar an der Vorsilbe „USER_“.

USER_STACK_SIZE

USER_STACK_SIZE Wert

Setzt den Stack auf die Größe, die mit **Wert** vorgegeben wird.

Wert ist eine Konstante.

Default (Standard)	maximal	minimal
2048 Byte	32767 Byte	512 Byte

Die Aufgabe des Stacks ist es für Unterprogramme Speicherplatz dynamisch bereitzustellen. Auf dem Stack geführt werden u.a. lokale Variable der Unterprogramme, die Referenzen auf übergebene Parameter sowie die Rückkehradresse.

Wird der Stack zu klein definiert, kann er irgendwann zur Laufzeit überlaufen, was zu einer schwerwiegenden Störung des Programm-Ablaufs führt. Die entsprechende Task wird angehalten. Im Debug-Modus erscheint dann eine Fehlermeldung, im RUN-Modus erfolgt keine weitere Anzeige. Es ist daher eine wichtige Aufgabe in der Entwicklungs-Phase, die maximal benötigte Stack-Größe durch Worst-case-Betrachtung festzulegen.

Der vom Stack belegte gesamte Speicherplatz (RAM) ermittelt sich aus der Anzahl der Tasks (einschließlich MAIN) * der hier spezifizierten Stackgröße. Beispiel: eine Anwendung mit 7 Tasks und einer Stackgröße von 2048 Byte belegt also insgesamt 7 * 2048 = 14 336 Bytes RAM.

Soll Speicherplatz gespart werden, sollte man zunächst auf die Verwendung von lokalen Variablen achten. Möglichst BYTE oder WORD Variablen verwenden statt LONG. STRING explizit mit maximaler Größe deklarieren, so daß nur diese Anzahl von Bytes freigehalten werden müssen, die auch gebraucht werden. Sonst wird stets die Standard-String-Länge verwendet (64Bytes, oder mit USER_STRING_SIZE gesetzt). Natürlich sollte in diesem Fall mit lokalen Arrays sparsam umgegangen werden. Während lokale Variable vollständig auf dem Stack geführt werden, belegen Ein- und Ausgangs-Parameter jeweils nur wenige Bytes für ihre Referenz.

USER_STRING_SIZE**USER_STRING_SIZE Wert**

Setzt die Standard-Größe von Strings, deren Größe nicht explizit deklariert wird, auf die Größe, die mit **Wert** angegeben wird.

Wert ist eine Konstante.

Default (Standard)	maximal	minimal
64 Byte	32765 Byte	1 Byte

Die Größe einzelner Strings kann durch Angabe der Größe in Klammern festgelegt werden, z.B. um mit dem Speicherplatz auf dem Stack sparsam umzugehen:

A\$(5)

USER_VAR_STRICT**USER_VAR_STRICT**

Erzwingt Deklaration von Variablen im Programm (siehe 'Variable initialisieren

In BASIC-Tiger®-Modulen mit Batteriepufferung behalten globale und Task-lokale Variable ihren Wert auch dann, wenn die Stromversorgung an VCC abgeschaltet wird. Beim ersten Einschalten oder nachdem die Batteriepufferung unterbrochen war oder auch nach einem Absturz z.B. durch Störimpulse **sind die Werte von Variablen nicht festgelegt**. Relativ harmlos für Tiger-BASIC® sind die numerischen Variablen, die einfach irgendwelche Werte enthalten (für Ihre Anwendung mag das eventuell fatal sein). Strings hingegen können unsinnige Längenangaben besitzen, z.B. können sie länger sein als die maximal mögliche Länge oder sogar eine negative Länge haben. Das ist – unabhängig von der Anwendung – auch für Tiger-BASIC® fatal und kann dazu führen, daß die ganze Anwendung nicht funktioniert. Es ist daher Aufgabe des Programm-Entwicklers, alle Variablen vor ihrer ersten Benutzung zu initialisieren. In größeren Programmen können durch versehentlich nicht-initialisierte Variable Fehler entstehen, die schwer zu finden sind und evetuell nur sporadisch auftreten. Wenn Variable nicht durch Akku-Pufferung erhalten bleiben müssen, dann können Sie auf Nummer sicher gehen, indem Sie die Compiler-Anweisung **USER_VAR_INIT** an den Anfang des Quelltextes setzen.

Tiger-BASIC®-Sprache

Programmbeispiel:

```
'-----
' Name: USER_VAR_INIT.TIG
' Initialisiere alle Variable vor der ersten Verwendung
' Compiler-Anweisung
'-----
USER_VAR_STRICT          ' Vars deklarieren!
USER_VAR_INIT            ' Initialisiere alle Variablen
                        ' vergleiche:
                        ' a) with USER_VAR_INIT
                        ' b) without USER_VAR_INIT

TASK MAIN
  REAL R1
  STRING A$              ' Variable, die Initialisiert werden
  LONG N                 ' durch USER_VAR_INIT

' LCD-Treiber installieren (BASIC-Tiger)
install_device #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' install_device #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8      IN

  PRINT #1,"<1>";R1;" (";A$;")";N'
  R1=1.23
  A$="abcd"              ' setze Variable auf definierte Werte
                        ' und siehe den Unterschied nach
  N=77                   ' einem RESET mit u. ohne
  PRINT #1,R1;" (";A$;")";N  ' USER_VAR_INIT
END
```

Siehe auch: USER_VAR_STRICT

Variable deklarieren', Seite 31).

USER_SECURITY

Um Ihre Software im BASIC Tiger®- oder TINY Tiger®-Modul zu schützen, fügen Sie folgende Zeile in Ihren Quelltext ein:

USER_SECURITY "Codewort"

"Codewort" besteht aus 3 bis 31 Zeichen. Das Codewort ist eine Kombination von Buchstaben und Ziffern und beginnt mit einem Buchstaben. Groß- und Kleinschreibung wird unterschieden.

Wenn ein Programm, welches eine USER_SECURITY-Zeile enthält, in das Modul geladen wurde, ist Debugging oder Programm löschen nur möglich, wenn das gleiche Codewort im Dialog **Security Code** des Menüs **Option** eingetragen wurde. Stellen Sie sicher, daß Sie das Codewort nicht vergessen, da es keine Möglichkeit gibt, das Programm im Modul ohne Codewort zu ändern.

SET_DATA_ADDR**SET_DATA_ADDR Startadr**

SET_DATA_ADDR: Setzt Adresse in Data-FLASH Bereich

Compiler-Anweisung: Setzt die Startadresse im Flash-Speicher, um konstante Daten mit der Instruktion DATA abzulegen.

Startadr ist eine Konstante vom Typ BYTE, WORD oder LONG und bestimmt die die Startadresse im User-Flash-Bereich für darauffolgende DATA-Instruktionen.

Mit der Instruktion SET_DATA_ADDR wird die Anfangsadresse festgelegt, ab der im Flash-Speicher konstante Daten dauerhaft abgelegt werden. Standard-Anfangsadresse ist Adresse 0.

Programmbeispiel:

```

'-----
' Name: SET_DATA_ADDR.TIG
'-----
TASK Main                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

SET_DATA_ADDR 40                        ' Anfangsadresse setzen
DATA &                                  ' Laenge:
41, &                                  ' 1
1234h                                   ' 2 Reihenfolge: 34 12

SET_DATA_ADDR 50                        ' Anfangsadresse setzen
DATA "America"                          ' 9 (mit Laengenangabe)

DataSize = 10 + 2 + LEN("America")      ' Groesse belegter FLASH
USING "UH<2><2> 2.2.2.2.2"
FOR I = 0 TO DataSize + 10
    PEEK_FLASH 40+I, B, 1                ' zeige Daten in HEX + einige
    PRINT_USING #1, B;                   ' Bytes mehr
NEXT
END                                      ' Ende Task MAIN

```

Siehe auch: DATA

DATA

DATA [Typ] *Konstlist*

DATA: Definiert Daten im FLASH: Bytes, Words, ... Files ...

Compiler-Anweisung: Initialisiert ein Datenfeld im Flash-Speicher.

Konstlist ist eine Liste von Konstanten vom Typ BYTE, WORD, LONG, REAL oder STRING und bestimmt die abzulegenden Werte.

Typ ist eine Angabe des Datentyps BYTE, WORD, LONG, REAL, STRING oder FILE und bestimmt den Datentyp und damit die Größe der abzulegenden Werte.

Die Instruktion DATA initialisiert einen FLASH-Bereich mit Konstanten. Die Datentypen der Konstanten dürfen verschieden sein. Die Listenelemente sind durch Kommata getrennt.

Die Anfangsadresse des Datenfeldes im User-Flashbereich wird mit der Instruktion **SET_DATA_ADDR** gesetzt.

Die Konstanten belegen soviel Platz im Flash, wie es Ihrer Größe entspricht. Zeichenketten werden mit deren Längenangabe abgespeichert.

Eine feste Größe wird durch die Angabe von 'BYTE', 'WORD', 'LONG' oder 'REAL' vorgegeben. Diese Vorgabe gilt für die folgenden Listenelemente, bis ein Semikolon als Trennzeichen erscheint. Ist der Datentyp angegeben, werden die Listenelemente in jedem Fall entsprechend der Angabe abgelegt. Werden Zeichenketten als BYTE abgelegt, so wird die Angabe der Länge nicht gespeichert.

WORD oder LONG-Werte werden in der Reihenfolge mit dem niederwertigen Byte zuerst abgelegt. Das gilt auch für die Längenangabe des Strings.

Die Typangabe **FILE** veranlaßt das Einlesen einer Datei, um byteweise Daten im Flash abzulegen. Insbesondere für vorbereitete Sound-Daten oder Grafikdaten ist diese Möglichkeit sehr nützlich.

Beispiele:

Angabe	Anz. Bytes	Bytes (hex.)
DATA 55	1	55
DATA 1234h	2	34 12
DATA 700000	4	60 AE 0A
DATA 24.5	8	REAL 8 Bytes
DATA "Hello"	7	05 00 'H' 'e' 'l' 'l' 'o'
DATA BYTE "Hello"	5	'H' 'e' 'l' 'l' 'o'
DATA BYTE 0CCDDEE	1	EE
DATA 24.5, 41, 0ffff, "Amerika"	20	8 Bytes REAL, 29 FF 0F 07 00 'A' 'm' 'e' 'r' 'i' 'k' 'a'
DATA BYTE 80, 65535, "A"; LONG 255	5	50 FF 'A' FF 00

Die im FLASH gespeicherten Daten können mit Hilfe der Instruktion PEEK_FLASH gelesen werden und mit POKE_FLASH können zusätzliche Daten geschrieben werden.

Tiger-BASIC®-Sprache

Programmbeispiel:

2

```
'-----
' Name: DATA.TIG
'-----
TASK Main                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  SET_DATA_ADDR 40                        ' Anfangsadresse setzen
  DATA &                                ' Laenge:
  41, &                                  ' 1
  1234h, &                                ' 2 Reihenfolge: 34 12
  "America"; &                            ' 9 (mit Laengenangabe)
  BYTE 0eeeeeeh, &                        ' 1 nur BYTE!
  0ff9h; &                                ' 1 nur BYTE!
  BYTE "Process "                          ' 8 (ohne Laengenangabe)

  DataSize = 1+2+2+LEN("America")&        ' Groesse des mit den Daten
                +1+1+LEN("Process ")      ' belegten FLASH-Bereiches
  USING "UH<2><2> 2.2.2.2.2"
  FOR I = 0 TO DataSize + 10              ' zeige Daten in HEX + einige
    PEEK_FLASH 40+I, B, 1                  ' Bytes mehr
    PRINT_USING #1, B;
  NEXT
END                                        ' Ende Task MAIN
```

Siehe auch: SET_DATA_ADDR

Daten aus Datei ins Flash einbinden.

Die Typangabe **FILE** veranlaßt das Einlesen einer Datei, um byteweise Daten im Flash abzulegen. Insbesondere für vorbereitete Sound-Daten oder Grafikdaten ist diese Möglichkeit sehr nützlich.

DATA FILE "Name" [, Offset, Anzahl]

Name	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und gibt die einzulesende Datei an. Laufwerk und Pfad werden entweder mit angegeben oder sind im Dialog Verzeichnisse des Menüs Optionen eingetragen.
Offset	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und Offset in der Datei an, ab dem Bytes für den Flashbereich eingelesen werden.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der zu lesenden Bytes fest.

Ohne Angabe von Offset wird ein Offset von 0 angenommen. Ohne angegebene Anzahl wird die ganze Datei eingelesen. Hinter FILE kann keine Liste von Dateien stehen.

DATA FILTER "*Name*", "*Filter*", [, *Argumente*]

Name ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und gibt die einzulesende Datei an. Laufwerk und Pfad werden entweder mit angegeben oder sind im Dialog **Verzeichnisse** des Menüs **Optionen** eingetragen.

Filter ist ein String, der den Filter bezeichnet.

Argumente sind die Argumente, die bei jedem Filtertyp verschieden sind.

Folgende Filter sind bisher implementiert:

Filter	Argumente / Beschreibung
GRAPHFLT	0: wandelt BMP in Grafik-Rohdaten, wie von Tiger-BASIC® benötigt.
WAVEFLT	wandelt WAV-Dateien in Sound-Rohdaten, wie von Tiger-BASIC® benötigt. Der erste Parameter hinter dem Filternamen ist ,0' und muß angegeben werden. Argumente (optional) geben das Zielformat in Tiger an: [,BitPerSample[, SamplePerSec]] BitPerSample: 8, 16 (default: 8) SamplePerSec: 44100, 22050, 11025, 8000 (default: 22050) wenn SamplePerSec in der Quelldatei kleiner als diese Angabe ist, wird die Samplerate nicht umgesetzt.)

Beispiel für eine Zeile, in der die Bitmap-Datei ABC.BMP eingebunden wird:

DATA FILTER "ABC.BMP","GRAPHFLT", 0

Beispiel für eine Zeile, in der die Sound-Datei ABC.WAV eingebunden wird:

DATA FILTER "ABC.WAV", "WAVEFLT", 0, 8, 11025

Positionen im Flash benennen

Um Flashdaten im Programm leicht zugänglich zu machen, können Sie Data-Marken (Data-Labels) verwenden. Genau wie Sprungmarken bezeichnen sie Stellen im User-Flash und ermöglichen symbolische Bezeichnungen für Flashbereiche. Datalabels enden im Gegensatz zu Sprungmarken zwei Doppelpunkten.

Programmbeispiel:

```

' -----
' Name: DATAFILE.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  DATALABEL DL0, DL1, DL2                ' drei Datalabels deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8

DL0::                                    ' Datalabel DL0 setzen
  DATA BYTE "Bytes at DL0"              ' einen String ins FLASH
DL1::                                    ' Datalabel DL1 setzen
  DATA FILE "C:\TIGERBAS\EXAMPLES\FLASHTXT.TXT" ' Datei ins FLASH
DL2::                                    ' Datalabel DL2 als Referenz

  PEEK FLASH DL0, A$, DL1 - DL0          ' lese A$ aus FLASH pos. DL0
  PRINT #1, A$                           ' Ausgabe auf LCD
  PEEK FLASH DL1, A$, DL2 - DL1          ' lese A$ aus FLASH pos. DL1
  PRINT #1, A$;                          ' Ausgabe auf LCD
END                                        ' Ende Task MAIN

```

Flashdaten aus User-Flashbereich in Datei speichern

Mit dem Befehl **FLASH-Daten lesen** aus dem Menü **Debug** gelangen Sie in ein Dialogfeld, in dem Sie die Start- und die Endadresse des zu lesenden Flashbereichs sowie den Dateinamen eingeben können. Die User-Daten aus dem Flash werden dann in eine Datei auf Ihrem Rechner übertragen.

2

Fehlerbehandlung während der Laufzeit

Zur Laufzeit können verschiedene Fehler auftreten:

- Fehler, die vom Programmierer vorhergesehen werden (falsche Eingaben, Pufferüberläufe, etc.)
Diese Fehler werden in der Regel durch entsprechende Abfragen im Programm oder durch explizite Fehlerbehandlung mancher Kommandos abgehandelt (**ON_ERROR_CALL**).
- Fehler, die nicht vorhergesehen wurden, die zu Datenverlusten oder falschen Werten führen.
Diese Fehler werden ignoriert. Das Programm soll auf jeden Fall weiterarbeiten (implizite Fehlerbehandlung).
- Fehler, die so fatal sind, daß das Programm nicht weiterlaufen kann (Stacküberlauf).
Solche Fehler führen zum Stoppen der Task, in der der Fehler aufgetreten ist. Im Debug-Mode erscheint eine Fehlermeldung.

Explizite Fehlerbehandlung

Bei einigen Instruktionen ist die Angabe einer Fehlerbehandlung nur für dieser einen Instruktion möglich (FIFO- und Flash-Instruktionen).

Informationen zur Fehler-Art und Ursache kann aus SYSTEM-Variablen gelesen werden (siehe SYSVARx-Funktionen, ab Seite 458).



Am Ende der Fehlerbehandlung muß das Fehlerflag mit der Instruktion **ON_ERROR_RESET** zurückgesetzt werden.

Es wird zur Einleitung von Maßnahmen ein Fehler-Behandlungs-Unterprogramm aufgerufen:

ON_ERROR_CALL Name

Die Fehlerbehandlung mit **ON_ERROR_CALL** hat den großen Vorteil, daß nach der Fehlerbehandlung das Programm normal weiterläuft.

Es erfolgt ein Abbruch. Das Programm springt aus der laufenden Bearbeitung heraus und verzweigt an eine durch das Label **Name** vorgegebene Stelle:

ON_ERROR_GOTO Name

Implizite Fehlerbehandlung

Wird die implizite Fehlerbehandlung ganz dem Tiger-BASIC®-System überlassen, führt dies im Fehlerfall zu Datenverlusten, falschen Werten oder anhalten von Tasks.

In jeder Task kann eine implizite Fehlerbehandlung so gesteuert werden, daß im Fehlerfall ein eigenes Unterprogramm aufgerufen wird.

ON_ERRTASK_CALL *Name*

Es wird das Unterprogramm **Name** aufgerufen, wenn ein Fehler außerhalb der Instruktionen mit expliziter Fehlerbehandlung auftritt.

ON_ERRTASK_GOTO *Name*

Es wird an die Stelle **Name** verzweigt., wenn ein Fehler außerhalb der Instruktionen mit expliziter Fehlerbehandlung auftritt.

Run-Time-Fehler

Nachdem im PC-Mode ein Run-Time-Fehler aufgetreten ist, steht die Markierung auf der **nächstfolgenden Befehlszeile**. Bei Task-Switching kann der Fehler in der Task aufgetreten sein, die unmittelbar vorher an der Reihe war. Um die Fehlerverursachende Programmzeile zu lokalisieren, steppen Sie mit 'F8' (nächste Anweisung - überall). So können Sie feststellen, an welcher Stelle die Programmausführung in den anderen Tasks ist, und welche Programmzeile den Fehler verursacht hat.

#DEFINE	Nr	Bedeutung
RE_TASK_NAVAIL	07H	Task-Nr. nicht verfügbar
RE_STR_OVL	08H	String-Länge überschritten
RE_PARAM_ILL	0BH	Parameter-Wert ist unzulässig
Device Fehler		
RE_DEVNR	20H	Geräte-Kanal-Nr. fehlerhaft
RE_PORTADR	21H	PORT-Adreß-Fehler
RE_DEV	22H	Geräte-Fehler
RE_NO_PACC	23H	I/O-Pin(s) bereits belegt
RE_NO_DEVN_INST	26H	Device-Treiber nicht installiert
FLASH-Speicher Fehler		
RE_NO_FLASH	30H	FLASH: nicht vorhanden
RE_FLASH_WR	31H	FLASH: Schreibfehler
RE_FLASH_BUSY	32H	FLASH: nicht bereit
RE_FLASH_ADR	34H	FLASH: Adreß-Fehler
RE_FLASH_PEEKL	35H	FLASH: PEEK-Länge ungültig z.B. 0 oder bei REAL-Wert <>8
RE_FLASH_TOT	37H	FLASH: Timeout
Puffer Fehler: (Fifo u.a.)		
RE_FIFO	40H	FIFO-Fehler
RE_FIFO_LEER	41H	FIFO ist leer
RE_FIFO_VOLL	42H	FIFO ist voll

Leere Seite

2

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

3 Instruktionen und Funktionen

Übersicht nach Gruppen

Programmfluß	Ein- und Ausgabe	Puffer	Numerische Funktionen
Unterprogramm	Seriell	Fifo	Bitweise
SUB...END CALL RETURN	SHIFT_IN SHIFT_OUT I2C_ACK_POLL I2C_READ\$ I2C_RESULT I2C_SETUP I2C_WRITE	FIFO CLEAR_FIFO LEN_FIFO FREE_FIFO GET_FIFO READ_FIFO UNGET_FIFO PUT_FIFO PUTU_FIFO START_FIFO INTEGRAL_FIFO	BIT BIT_MIRR BITNOT CUT_BITS INV_BIT RES_BIT SET_BIT MASK IMASK SIGNEXT PARITY siehe auch: Ver- schlüsselung
Schleifen und Verzweigungen	Geräte	String als Puffer	Mathematisch
LOOP...ENDLOOP FOR...NEXT WHILE...ENDWHILE IF...THEN ...ELSE...ENDIF GOTO SWITCH...CASE ...ENDSWITCH SWITCHI...CASE ...ENDSWITCH	INSTALL_DEVICE GET PUT INPUT INPUT_LINE PRINT PRINT USING	NTOS\$ NFROMS RTOS\$ RFROMS STOS\$	ABS CALC_CRC EXP EXPE FIX INT LD LIMIT LN LOG MOD MODULO_INC MODULO_UPDO LIN_APPROX PRIME RND RANDOMIZE SGN SQRT
Taskverwaltung	Pins und Ports	Flash	
TASK...END RUN_TASK STOP_TASK CONT_TASK RESTART_PROG SET_TASK_PRIO RELEASE_TASK EXIT_TASK DISABLE_TSW ENABLE_TSW ON_ERROR_RESET ON_ERRTASK_CALL ON_ERRTASK_GOTO	DIR_PIN DIR_PORT IN OUT LL_IPOINT_AND LL_IPOINT_IN LL_IPOINT_OR LL_IPOINT_OUT LL_IPOINT_PULSE LL_IPOINT_XOR XSETUP XIN, XIN\$ XOUT XBUS_INR\$ XBUS_OUTR	PEEK_FLASH POKE_FLASH POKEM_FLASH CHECK_FLASH ERASE_FLASH SET_SERIAL_NO SERIAL_NO\$ DELETE_PROG SET_DATA_ADDR DATA LET\$	

Übersicht nach Gruppen

3

Numerische Funktionen
Trigonometrische Funktionen
ACOS
ACOT
ASIN
QUICK_WORD_COS
QUICK_WORD_SIN
ATAN
COS
COSH
COT
COTH
SIN
SINH
TAN
TANH

String-Operationen
Allgemein
LET\$
LEN
SET_LEN\$
MAX_LEN
FILL\$
LEFT\$
MID\$
RIGHT\$
REMOVE\$
REMDOUBLE\$
ADD\$
AND\$
OR\$
XOR\$
SHIFTL\$
SHIFTM\$
UPPER\$
UPPER7\$
CONVERT\$
CONVERT7\$
siehe auch:
String als Puffer
Suchen
Spiegelung
Verschlüsselung
Komprimierung
Signalverarbeitung
Formatierung
FRAME
TABS
TRIM\$
USING
PRINT_USING
SPC
TAB
STRI\$
Formatstrings

Grafik
Pixel
OR\$, AND\$, XOR\$
GRAPHIC_MASK_COPY
GRAPHIC_MIRROR
GRAPHIC_COPY
INVERT
GRAPHIC_FILL_MASK
Special
GRAPHIC_EXP\$
FILL_AREA
Vektor
DRAW_LINE
DRAW_NEXT_LINE
CLOSE_LINE
SET_BASE
SET_SCALE
SET_ROTATION
SET_GRAREA
SET_DOT
DISTANCE

Spezielles
Konvertierung
VAL_NUM
VAL_REAL
STR\$
STRI\$
CHR\$
ASC
LTR
LLTOR
HREAL
LREAL
RTL
SIGNEXT
Spiegelungen
BIT_MIRR
BYTE_MIRR
BIT_MIRR\$
MIRROR\$
GRAPHIC_MIRROR
Dateneingabe und – übertragung
DEBOUNCE
DECODE
SCAN_TO_CHAR_
SETUP
SCAN_TO_CHAR
SCAN_TO_TOGGLE
siehe auch: Komp-
rimierung, Ver-
schlüsselung
Suchen
INSTR
INDEX
INDEX_1D
INDEX_2D
SELECT\$
System
SYSVAR
SYSVAR\$
SET_SYSVAR
RESTART_PROG
DELETE_PROG
THIS_VERS
DEVEN

Übersicht nach Gruppen

Spezielles
Komprimierung
PACK_BYTE\$
UNPACK_BYTE\$
ASC_BCD\$
BCD_ASC\$
ASC_PBCD\$
PBCD_ASC\$
PBCD_BCD\$
BYTE_ANIB\$
ANIB_BYTE\$
BYTE_INIB\$
INIB_BYTE\$
Zeit
WAIT_DURATION
WAIT_NEXT
WAIT_CLOCK
TICKS
SET_TICKS
Verschlüsselung
BIT_MIX\$
BIT_DEMIX\$
BYTE_DEMIX\$
BYTE_DEMIX\$
SCRAMBLE\$
BYTE_MIX\$
CODE_1TON\$
DISTRIBUTE\$
TAN – Trans-Aktions-Nummern
TAN_CHK
TAN_COUNT
TAN_DEL
TAN_FIRST

Spezielles
Signal-verarbeitung
SIGNAL_ADD
SIGNAL_AVERAGE
SIGNAL_BIGGER
SIGNAL_CUT_AMP
SIGNAL_CUT_MAX
SIGNAL_CUT_MIN
SIGNAL_GAP_END
SIGNAL_GAPI_END
SIGNAL_MAX
SIGNAL_MAX_AMPL
SIGNAL_MIN
SIGNAL_MUL
SIGNAL_NEXT_GAP
SIGNAL_NEXT_GAPI
SIGNAL_OFFSET
SIGNAL_SCALE
SIGNAL_SMALLER
SIGNAL_SMOOTH

Spezielles
Operatoren
+
-
*
/
BITAND
BITOR
BITXOR
SHL
SHR
SHRA
ROL
ROR
Boolsche Operatoren
AND
OR
XOR
NOT
>
>=
<
<=
<>
=

Spezielles
Prä-Prozessor
#DEFINE
#INCLUDE
#PROJECT_MODEL
#COMMENT
#ENDCOMMENT
Compiler-Direktiven
USER_VAR_STRICT
USER_STACK_SIZE
USER_STRING_SIZE
USER_SECURITY
USER_EPORT
SET_DATA_ADDR
DATA
Deklarationen
BYTE
WORD
LONG
REAL
STRING
ARRAY
FIFO
DATALABEL

Übersicht nach Gruppen

Leere Seite

3

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

Programmfluß: Allgemein

SUB...END	Unterprogramm definieren
CALL	Unterprogramm aufrufen
RETURN	Unterprogramm verlassen

Programmfluß: Schleifen / Verzweigungen

LOOP...ENDLOOP	Anzahl von Schleifen unbedingt ausführen
FOR...NEXT	Schleifen mit Laufvariable: Start, Stopp, Schrittw.
WHILE...ENDWHILE	Schleife mit Abbruchbedingung am Anfang
IF..THEN..ELSE..ENDIF	Bedingte Programmausführung, ggf. mit Alternative
GOTO	Unbedingte Programmverzweigung
SWITCH..CASE..ENDCASE	Bedingte Programmausführung, Standard
SWITCHI..CASE..ENDCASE	Bedingte Programmausführung, high-speed

Programmfluß: Multitasking

TASK ... END	Task Definition
RUN_TASK	Startet Task
STOP_TASK	Stoppt Task
CONT_TASK	Setze gestoppte Task wieder fort
EXIT_TASK	Beendet Ausführung einer Task
DISABLE_TSW	Schaltet Task-Switching Mechanismus AUS
ENABLE_TSW	Schaltet Task-Switching Mechanismus EIN
RELEASE_TASK	Rest der augenblicklichen Zeitscheibe freigeben
SET_TASK_Prio	Setzt Priorität für Task-Switching
RESTART_PROG	• Startet BASIC-Programm neu, Warmstart
DELETE_PROG	• Löscht das aktuelle BASIC-Programm, PC-Mode
CUT_AND_PASTE	• Schneidet Wert heraus und kopiert
ON_ERROR_RESET	Setzt Fehler-Flag zurück nach der Bearbeitung
ON_ERRTASK_CALL	Ruft Unterprogramm auf bei Fehler in Task
ON_ERRTASK_GOTO	Springt zu Sprungmarke bei Fehler in Task

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

Ein- und Ausgabe: Seriell

SHIFT_IN	Getaktete, serielle Eingabe für externe Chips
SHIFT_OUT	Getaktete, serielle Ausgabe zu externen Chips
I2C_SETUP	I2C-Bus Einstellungen setzen
I2C_READ\$	Liest Bytes von I2C-Bus
I2C_WRITE	Schreibt Datenstring auf I2C-Bus
I2C_RESULT	Liest Status der letzten I2C-Bus Operation
I2C_ACK_POLL	ACK-Polling auf I2C-Bus ob Device bereit ist

Ein- und Ausgabe: Geräte / Device-Treiber

INSTALL_DEVICE	Device Treiber installieren
GET	Unformatierte Daten-Eingabe von Device-Treiber
PUT	Unformatierte Daten-Ausgabe an Device-Treiber
INPUT	Daten-Eingabe von Dev.-Treiber, Trennzeichen
INPUT_LINE	Daten-Eingabe von Dev.-Treiber, Zeilen
PRINT	Formatierte Daten-Ausgabe an Dev.-Treiber
PRINT_USING	Daten-Ausgabe an Dev.-Treiber, + Formatierungen
USING	Format-Anweisung für PRINT_USING

Ein- und Ausgabe: Pins und Ports

DIR_PIN	Setzt die Datenrichtung eines internen I/O-Pins
DIR_PORT	Setzt die Datenrichtung eines internen I/O-Ports
IN	Liest Bits von I/O-Port ein
OUT	Schreibt Bits auf I/O-Port
LL_IPORT_AND	Bitweise UND-Verknüpfung auf internem I/O-Port
LL_IPORT_IN	Einlesen von einem internem I/O-Port + BITAND
LL_IPORT_OR	Bitweise ODER-Verknüpfung auf internem I/O-Port
LL_IPORT_OUT	Schnelle Ausgabe auf internem I/O-Port
LL_IPORT_PULSE	Toggelt ein Bit eines internen I/O-Ports
LL_IPORT_XOR	Bitweise XOR-Verknüpfung auf internem I/O-Port
XSETUP	• Setzt Einstellung für XPort I/O-Erweiterung
XIN, XIN\$	• Liest Byte(s) von XPort I/O-Erweiterungs-System
XOUT	• Schreibt Byte(s) auf XPort I/O-Erweiterung
XBUS_INR\$	• Schnelles Einlesen von Daten über XPort-Bus
XBUS_OUTR	• Schnelles Ausgeben von Daten über Xport-Bus

• = nur in Projektmodell ‚PM_FULL‘

Puffer: Fifo

FIFO	Deklaration einer Variablen vom Typ Fifo
CLEAR_FIFO	Fifo löschen und in Anfangs-Zustand bringen
GET_FIFO	Hole Eintrag aus Fifo (wenn vorhanden)
PUT_FIFO	Schreibe Eintrag in Fifo (wenn Platz)
UNGET_FIFO	Mache GET_FIFO rückgängig (wenn möglich)
PUTU_FIFO	Schreibe Eintrag in Fifo (ältester Eintr. ev. verloren)
LEN_FIFO	Gesamt-Länge des Fifo melden
FREE_FIFO	Anzahl freier Fifo-Einträge melden
READ_FIFO	Beliebiges Element aus Fifo lesen
START_FIFO	Fifo löschen + vorbereiten für INTEGRAL_FIFO
INTEGRAL_FIFO	Fifo benutzen als Kurzzeit-Integrator (Window)

Puffer: String als Puffer

NTOS\$	Schnell: Numerischer Wert (Integer) nach String.
NFROMS	Schnell: Numerischer Wert (Integer) aus String.
RTOS\$	Schnelle Ausführung: Real Wert nach String.
RFROMS	Schnelle Ausführung: Real Wert aus String.
STOS\$	Schnelle Ausführung: String nach String.

Flash

DATA	Definiere Daten im FLASH: Bytes, Words, Files
SET_DATA_ADDR	Setzt Adresse in Data-FLASH Bereich
POKE_FLASH	Schreibt Daten in Daten-FLASH Bereich (standard)
PEEK_FLASH	• Liest Daten aus FLASH in Variable ein
POKEM_FLASH	• Schreibt Daten in Daten-FLASH Bereich
ERASE_FLASH	Löscht FLASH-Sektor(en), nur Daten-Bereich
SET_SERIAL_NO	• Schreibt Serien-Nummer, Installations-Code, etc.
SERIAL_NO\$	• Liest Serien-Nummer, Installations-Code
DELETE_PROG	• Löscht das aktuelle BASIC-Programm, PC-Mode
CHECK_FLASH	Findet letzte programmierte Stelle im User-Flash
LET\$	• Kopiert String (high-speed)
RND	Liefert Pseudo-Zufallswert
RANDOMIZE	Verändert Pseudo-Zufallswerte-Serien
LIN_APPROX	Zwischenschritte durch lineare Approximation

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

Numerische Funktionen: Bitweise

BIT	Testet Wert des gewünschten Bits in einem Wert
BIT_MIRR	Bit-Vertauschungen in einer Variablen (Spiegelung)
BITNOT	Invertiert alle Bits eines numerischen Wertes
CUT_BITS	• Schneidet Bit-Gruppe aus numerischem Wert aus
INV_BIT	Invertiert 1 Bit in numerischer Variable
RES_BIT	Setzt ein Bit in numerischer Variable auf ‚0‘
SET_BIT	Setzt ein Bit in numerischer Variable auf ‚1‘
MASK	Gibt Maske mit einem gesetzten Bit zurück
IMASK	Gibt inverse Maske zurück, ein „0“ Bit
SIGNEXT	Vorzeichenrichtige Wert-Erweiterung
PARITY	• Berechnet Parity eines numerischen Wertes

Numerische Funktionen: Mathematisch

ABS	Betrag eines Integer- oder Fließkomma-Wertes
SGN	Vorzeichen eines numerischen Wertes
CALC_CRC	• Berechnung einer CRC-Prüfsumme
EXP	Exponential-Funktion für Integer-Werte
EXPE	Exponential-Funktion zur Basis e, Real-Werte
FIX	Schneidet Nachkommastellen eines REALs ab
INT	Liefert nächst gelegene Ganzzahl eines REALs
LIMIT	• Begrenzt Integer Wert auf zulässigen Bereich
LD	Logarithmus Dualis, Basis 2
LN	Logarithmus Naturalis, Basis e
LOG	10-er Logarithmus
MOD	Modulo, Rest einer Integer-Division
SQRT	Wurzel
MODULO_INC	• Modulo Increment, steigend
MODULO_UPDO	• Umlaufendes Modulo-INC/DEC
PRIME	• Testet, ob Integerwert Primzahl ist.

• = nur in Projektmodell „PM_FULL“

Numerische Funktionen: Trigonometrische Funktionen	
SIN	SINUS, Real-Werte
COS	COSINUS, Real-Werte
TAN	TANGENS, Real-Werte
COT	COTANGENS, Real-Werte
ASIN	ARCUS SINUS, Real-Werte
ACOS	ARCUS COSINUS, Real-Werte
ATAN	ARCUS TANGENS, Real-Werte
ACOT	ARCUS COTANGENS, Real-Werte
SINH	SINUS hyperbolicus, Real-Werte
COSH	COSINUS hyperbolicus, Real-Werte
TANH	TANGENS hyperbolicus, Real-Werte
COTH	COTANGENS hyperbolicus, Real-Werte
QUICK_WORD_COS	• Integer COSINUS, schnelle Berechnung
QUICK_WORD_SIN	• Integer SINUS, schnelle Berechnung

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

String-Operationen: Allgemein

LET\$	• String-Zuweisung, high-speed
LEN	Ermittelt die aktuelle String-Länge
MAX_LEN	Ermittelt die maximale String-Länge
SET_LEN\$	Setzt die Länge eines Strings gezielt
FILL\$	Füllt String mit bel. Muster-String
LEFT\$	Liefert linken Teilstring
MID\$	Liefert mittleren Teilstring
RIGHT\$	Liefert rechten Teilstring
REMOVE\$	Entfernt Zeichen aus String
REMOUABLE\$	Entfernt dopplete Zeichen aus String
TRIM\$	Entfernt führende und/oder nachfolgende Zeichen
ADD\$	Addiert Wert auf alle Bytes eines Strings
AND\$	Maskiert alle Bytes eines Strings per AND
OR\$	Maskiert alle Bytes eines Strings per OR
XOR\$	Verknüpft alle Bytes eines Strings per XOR
SHIFTL\$	• Schiebt die Bits eines Strings um 1...8 Positionen
SHIFTM\$	• Schiebt die Bits eines Strings um 1...8 Positionen
UPPER\$	Konvertiert Buchstaben in String zu Großbuchst.
UPPER7\$	Dito, jedoch entferne ev. gesetztes Bit-7
CONVERT\$	Zeichen-zu-Zeichen Konversion lt. eigener Tabelle
CONVERT\$	Dito, jedoch entferne ev. gesetztes Bit-7
CODE_1TON\$	• Konvertiert in String: 1 Byte ==> N Bytes
DECODE	• Decodiert Stringdaten, (fehlerkorrigierende Codes)

Grafik: Pixel-Grafik

OR2\$, OR3\$, OR4\$	• Vereint 2, 3 oder 4 Strings bytewise per OR
AND2\$, AND3\$, AND4\$	• Vereint 2, 3 oder 4 Strings bytewise per AND
XOR1\$	• Überlagert 1 String bytewise per XOR
GRAPHIC_MASK_COPY	• Vereinigt 2 Grafiken lt. Grafik-Make
GRAPHIC_COPY	• Kopiert Fenster aus einer Grafik in andere Grafik
GRAPHIC_MIRROR	• Spiegelt eine Pixel-Grafik
GRAPHIC_FILL_MASK	• Baut rechteckige Maske in Grafik-String auf
GRAPHIC_EXP\$	• Bit-Expansion zur Anpassung an Ausgabe-Geräte
INVERT	• Invertiert Anzahl Bytes in einem String
FILL_AREA	• Füllt Gebiete mit beliebiger Kontur
LET\$	• Kopiert String (high-speed)

• = nur in Projektmodell ‚PM_FULL‘

Grafik: Vektor-Grafik

DRAW_LINE	• Zeichnet Linie inklusive Linienanfang
DRAW_NEXT_LINE	• Fügt weiteres Linienelement an Linie an
CLOSE_LINE	• Schließt einen offenen Linienzug
SET_BASE	• Setzt den Koordinaten-Ursprung neu
SET_SCALE	• Setzt X-/Y-Skalierungs-Faktoren neu
SET_ROTATION	• Setzt Schreibstift Rotation
SET_GAREA	• Setzt Grafik-Area
SET_DOT	• Setzt einzelnes Pixel in Grafik
DISTANCE	• Ermittelt den Abstand zweier Koordinaten-Paare

3

Spezielles: Konvertierung

VAL_NUM	Liest Integer Wert aus einem String
VAL_REAL	Liest Real Wert aus einem String
STR\$	Wandelt numerischen Wert um in String
STRIS\$	Wandelt numerischen Wert in formatierten String
CHR\$	Typumwandlung: Byte -> 1 Stringzeichen
ASC	Liefert den numerischen Wert eines Stringzeichens
LTR	Typumwandlung: LONG to REAL
LLTOR	Kombiniert Bytes aus 2 LONGs zu 1 REAL-Var.
HREAL	Liefert high 32-Bits of REAL-Wert
LREAL	Liefert low 32-Bits of REAL-Wert
RTL	Typumwandlung: REAL to LONG
SIGNEXT	Vorzeichenrichtige Wert-Erweiterung

Spiegelungen

BIT_MIRR	Kehrt Reihenfolge der Bits um in num. Variable
BYTE_MIRR	Kehrt Reihenfolge der Bits um in num. Variable
BIT_MIRR\$	Kehrt Reihenfolge der Bits um in String
MIRROR\$	Kehrt Reihenfolge der Zeichen in einem String um
GRAPHIC_MIRROR	• Spiegelt Pixel-Grafik an X- und/oder Y-Achse

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

Komprimierung / Konvertierung

PACK_BYTE\$	• Komprimiert Strings
UNPACK_BYTE\$	• De-Komprimiert Strings
BYTE_ANIB\$	• Komprimiert Bytes zu Nibbles, Datenreduktion
ANIB_BYTE\$	• De-Komprimiert Nibbles wieder zu Bytes
BYTE_INIB\$	• Komprimiert Bytes zu Incremental-Nibbles
INIB_BYTE\$	• De-Komprimiert I-Nibbles wieder zu Bytes
ASC_BCD\$	• Konvertiert: ASCII-HEX Ziffern -> BCD-Bytes
BCD_ASC\$	• Konvertiert: BCD -> ASCII-HEX
ASC_PBCD\$	• Konvertiert: ASCII-HEX Ziffern zu „packed BCD“
PBCD_ASC\$	• Konvertiert: packed BCD -> ASCII-HEX
BCD_PBCD\$	• Konvertiert: BCD -> „packed BCD“
PBCD_BCD\$	• Konvertiert: „packed BCD“ -> BCD

Suchen / Quantifizieren

INSTR	Sucht Zeichenfolge in einem String, liefert Position
INDEX	Sucht Zeichenfolge in einem String, liefert Index
INDEX_1D	• 1D-Wertklassen, Widerst.-Tastaturen, Toleranzgr.
INDEX_2D	• 2D-Wertklassen, X/Y-Pos. (Index, Touchpanel)
SELECT\$	Selektiert String-Baustein an Hand eines Indexes

Konvertierung / Chiffrierung

BIT_MIX\$	• Verwürfelt Bits in String lt. Verwürfelungs-String
BIT_DEMIX\$	• Entwürfelt Bits in String lt. Verwürfelungs-String
BYTE_MIX\$	• Verwürfelt Bytes in String lt. Verwürfelungs-String
BYTE_DEMIX\$	• Entwürfelt Bytes in String lt. Verwürfelungs-String
SCRAMBLE\$	• Macht Stringbytes unkenntlich lt. Scramble-String
CODE_1TON\$	• Konvertiert in String: 1 Byte ==> N Bytes
DISTRIBUTE	• Verteilt Bits in einem String (Bitstream)

Transaktions-Nummern (TAN)

TAN_CHK	• Prüft ob gültige TAN, setzt „benutzt“
TAN_COUNT	• Zählt TANs: freie, benutzte, gelöschte
TAN_DEL	• Löscht TAN
TAN_FIRST	• Sucht 1.TAN: freie, benutzte oder gelöschte

• = nur in Projektmodell ‚PM_FULL‘

Formatierungen

USING	Formatstr. f. formatierte Ausg. v. num. Werten
PRINT USING	Formatierte Daten-Ausgabe an Dev.-Treiber
SPC	Erzeugt feste Anz. Leerz. in der PRINT-Ausgabe
TAB	Leerz. in der PRINT-Ausgabe bis zu Tabulatorpos.
STRIS\$	Wandelt numerischen Wert in formatierten String
TRIM\$	Entferne führende und/oder nachfolgende Zeichen
FRAME	Erzeugt Ausgaberahmenstring f. PRINT_USING
TABS	Setzt Tabulatorpositionen für PRINT_USING

Sonstige

SCAN_TO_CHAR_SETUP	• Tastatur-Scan einrichten
SCAN_TO_CHAR	• Scan Taste, die Code generiert in Keyboards
SCAN_TO_TOGGLE	• Scan Toggles in Keyboards
DEBOUNCE	• Entprellt: Tasten, Kontakte, Touchpanel, Lichtschr.
DECODE	• Decodiere lt. Table, z.B: fehlerkorrigierende Codes

Systemnahe

RESTART_PROG	• Startet BASIC-Programm neu, Warmstart
DELETE_PROG	• Löscht das aktuelle BASIC-Programm, PC-Mode
READ_TSW	Liest akt. Einstellung des Task-Switches
THIS_VERS	Gibt die akt. Version dieses Runtime-Systems
DEVEN	Meldet letzten Device Error einer Task
SYSVARN	Liest numerische Systemvariable
SYSVAR\$	Liest String-Systemvariable
SET_SYSVARN	Setzt Systemparametert

Zeit

TICKS	Liefert akt. Zeit des 1 ms System-Taktes
SET_TICKS	Setzt 1 ms System-Taktzähler auf neuen Wert
DIFF_TICKS	Liefert Zeitdifferenz in 1 ms System-Ticks
WAIT DURATION	Ausführung dieser Task wartet Zeit-DAUER
WAIT CLOCK	Ausf. dieser Task wartet bis TICKS Wert erreicht ist
WAIT NEXT	Ausführung dieser Task wartet Zeit-RASTER

Kommentierte Übersicht

• = nur in Projektmodell ‚PM_FULL‘

Signalverarbeitung

SIGNAL_ADD	• Überlagert 2 Signale additiv
SIGNAL_AVERAGE	• Ermittelt den Signal-Mittelwert, (DC-Komponente)
SIGNAL_CUT_AMP	• Begrenzt Signal-Amplitude
SIGNAL_CUT_MAX	• Begrenzt Maximalwerte im Signal
SIGNAL_CUT_MIN	• Begrenzt Minimalwerte im Signal
SIGNAL_GAP_END	• Sucht eine Lücke im Signal
SIGNAL_NEXT_GAP	• Sucht nächste Lücke im Signal
SIGNAL_GAPI_END	• Sucht eine Lücke im Signal, Kurzzeitintegration
SIGNAL_NEXT_GAPI	• Sucht nächste Lücke im Signal, Kurzzeitintegration
SIGNAL_MAX	• Ermittelt Maximalwert eines Signals
SIGNAL_MIN	• Ermittelt Minimalwert eines Signals
SIGNAL_MAX_AMPL	• Ermittelt größte Amplitude eines Signals
SIGNAL_MUL	• Multipliziert 2 Signale
SIGNAL_OFFSET	• Verschiebt Signalpegel um einen Offset
SIGNAL_SCALE	• Verstärkt oder Dämpft das Signal
SIGNAL_BIGGER	• Ermittelt Samples die größer sind als Vergleichsw.
SIGNAL_SMALLER	• Ermittelt Samples die kleiner sind als Vergleichsw.
SIGNAL_SMOOTH	• Flacht die Enden eines Signals ab

Operatoren

+	Addieren (String: anhängen)
-	subtrahieren
*	multiplizieren
/	Dividieren
BITAND	Bitweise UND
BITOR	Bitweise ODER
BITXOR	Bitweise Exklusiv ODER
SHL n	links schieben um n Bits
SHR n	rechts schieben um n Bits
SHRA n	arithmetisch rechts schieben
ROL n	links rotieren um n Bits
ROR n	rechts rotieren um n Bits

- = nur in Projektmodell ‚PM_FULL‘

Operatoren: Boolsche

AND	logisch UND
OR	logisch ODER
XOR	logisch exklusiv ODER
NOT	logisch NICHT
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
<>	ungleich
=	gleich

3

Prä-Prozessor-Direktiven

#DEFINE	Definiert Symbol für Rest der Zeile
#INCLUDE	Bindet andere Datei in Quelltext ein
#PROJECT_MODEL	Legt Projektmodell fest (groß, klein)
#COMMENT	Beginnt Kommentarbereich in Quelltext
#ENDCOMMENT	Beendet Kommentarbereich in Quelltext

Compiler-Direktiven

USER_STACK_SIZE	Setzt Task-Stackgröße
USER_VAR_STRICT	Erzwingt Variablendefinition
USER_STRING_SIZE	Setzt Standard-Stringgröße
USER_EPORT	Setzt Parameter des EPORT-Systems
USER_SECURITY	Setzt Sicherungswort
SET_DATA_ADDR	Legt Adresse im Daten-Flash fest
DATA	Bindet Daten in Flash ein (binär, BMP, WAV)

Variablen-Deklarationen

BYTE	Kleine Werte ohne Vorzeichen (0...255)
WORD	Werte ohne Vorzeichen (0...65535)
LONG	Werte von -2.147.483.648 bis 2.147.483.647
REAL	Fließkommawerte von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$
STRING	Zeichenkette
DATALABEL	Adresse im Daten-Flash
ARRAY	Feld aus Bytes, WORDs, LONGs oder Strings
FIFO	First-in-First-out-Puffer

ABS

ABS

ABS (Wert)

ABS: Betrag eines Integer- oder Fließkomma-Wertes

Funktion: Liefert den Betrag (Absolutwert) von **Wert** zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL.

Funktionswert ist vom Typ BYTE, WORD, LONG oder REAL. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
'-----  
' Name: ABS.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  USING "NF<1><1>*<2>V0.0.0.0.0.0.0.1.2.0.0.0.0.0.0"  
  PRINT USING #1, "ABS(5.44) ="; ABS(5.44) ' Ausgabe auf LC-Display  
  PRINT USING #1, "ABS(-6.33) ="; ABS(-6.33) ' "  
  PRINT #1, "ABS(544) ="; ABS(544) ' "  
  PRINT #1, "ABS(-633) ="; ABS(-633) ' "  
END                                         ' Ende Task MAIN
```

Siehe auch: VAL_NUM, VAL_REAL

ACOS

ACOS (Wert)

ACOS: ARCUS COSINUS, Real-Werte

Funktion: Liefert den Arcuscosinuswert von **Wert** im Bogenmaß (rad) zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -1.0 bis 1.0.

Funktionswert ist vom Typ REAL und liegt im Bereich von 0 bis PI (PI=3.14..). Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{rad} * 180/\text{PI}$

Programmbeispiel:

```

' -----
' Name: ACOS.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  REAL Arg                              ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  FOR L = -100 TO 100                    ' Werte von -100 bis 100
    Arg = L / 100.0                      ' Arg fuer ACOS erzeugen
    USING "NF<0><1> <2>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT #1, "<1>ACOS (";               ' Ausgabe auf LC-Display
    PRINT USING #1, Arg; " )"            ' Argument ausgeben
    USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT USING #1, "="; ACOS (Arg)      ' Arcuscosinuswert ausgeben
    WAIT DURATION 1000                  ' 1 sek warten
  NEXT                                  ' naechster Wert
END                                      ' Ende Task MAIN

```

Siehe auch: SIN, COS, TAN, COT, ASIN, ATAN, SINH

ACOT

ACOT

ACOT (Wert)

ACOT: ARCUS COTANGENS, Real-Werte

Funktion: Liefert den Arcuscotangens von **Wert** im Bogenmaß (rad) zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $4,19 \cdot 10^{-307}$ bis $1,67 \cdot 10^{308}$. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von 0 bis PI ($\text{PI} \approx 3,14..$). Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{rad} \cdot 180/\text{PI}$

Programmbeispiel:

```
'-----
' Name: ACOT.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  REAL Arg                              ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  FOR L = -100 TO 100                    ' Werte von -100 bis 100
    Arg = LTR (L)                        ' Arg fuer ACOT erzeugen
    USING "NF<0><1> <2>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT #1, "<1>ACOT (";               ' Ausgabe auf LC-Display
    PRINT USING #1, Arg; " )"            ' Argument ausgeben
    USING "NF<0><1> <15>V0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT USING #1, "="; ACOT (Arg)      ' Arcuscotangenswert ausgeben
    WAIT DURATION 1000                  ' 1 sek warten
  NEXT                                  ' naechster Wert
END                                    ' Ende Task MAIN
```

Siehe auch: SIN, COS, TAN, COT, ASIN, ACOS, ATAN, SINH

ADD\$

ADD\$ (*String*, *Wert*)

ADD\$: Addiert Wert auf alle Bytes eines Strings
(nur in Projektmodell 'PM_FULL')

Funktion: Addiert zu allen Bytes in **String** den angegebenen **Wert**.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu bearbeitenden Zeichen.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält den 8-Bit-Wert für die ADD-Operation.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Zu den Zeichen in **String** wird byteweise der angegebene Wert addiert. Dabei werden Überläufe ignoriert.

Beispiel:

Quellstring (2 Bytes):	00001111 (0FH)	11111110 (FEH)
Wert 20H:	00100000 (20H)	00100000 (20H)
Ergebnis-String:	00101111 (2FH)	00011110 (1EH)

ADD\$

Programmbeispiel:

```
'-----
' Name: ADD$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$ = "<0FH><FEH>"                       ' das is A$
  PRINT #1, "A$= HEX:";                  ' zeige A$
  PRINT_USING #1, ASC(LEFT$(A$,1));" ";ASC(RIGHT$(A$,1))
  B$ = ADD$ ( A$, 20H )                   ' A$ AND 55H
  PRINT #1, "ADD$ ";
  PRINT_USING #1, 20H;" ";20H
  PRINT #1, "B$= HEX:";                  ' zeige Ergebnis
  PRINT_USING #1, ASC(LEFT$(B$,1));" ";ASC(RIGHT$(B$,1))
END                                       ' Ende Task MAIN
```

Siehe auch: AND\$, OR\$, BIT_MIRR\$, XOR\$, Operator '+'

AND\$

AND\$ (*String*, *Bitmuster*)

AND\$: Maskiert alle Bytes eines Strings per AND
(nur in Projektmodell 'PM_FULL')

Funktion: Führt mit allen Bytes in **String** eine AND-Operation mit **Bitmuster** aus.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu bearbeitenden Zeichen.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält das 8-Bit breite Bitmuster für die AND-Operation.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Zeichen in **String** werden byteweise mit einer AND-Operation bearbeitet. Dabei werden alle Bits, die im Bitmuster auf '1' stehen, aus dem Quellbyte ins Zielbyte übernommen. Alle Bits, die im Bitmuster auf '0' stehen, werden im Zielbyte zu '0'.

Beispiel:

Quellstring (2 Bytes):	01111111	01100110
Bitmuster:	01010101	01010101
Ergebnis-String:	01010101	01000100

AND\$

Programmbeispiel:

```
'-----
' Name: AND$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "<7FH><66H>"                       ' das is A$
  PRINT #1, "0:";                        ' zeige A$
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = AND$ ( A$, 55H )                  ' A$ AND 55H
  PRINT #1, "A:";
  CALL DISP_BIN ( 2, "<55H><55H>" )        ' zeige AND-Maske
  PRINT #1, "1:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
END                                       ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )            ' zeige CNT Bytes binaer
  BYTE B, I, J                          ' lokale Variable
  FOR I = 1 TO CNT                       ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ) ) ' hole ein Byte
    FOR J = 1 TO 8                       ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN           ' teste ob bit 1 ist
        PRINT #1, "1";                  ' drucke Bit
      ELSE                               ' falls kein Rest
        PRINT #1, "0";                  ' drucke Bit
      ENDIF
      B = B SHL 1                        ' B eins nach rechts (neues Bit)
    NEXT J
    PRINT #1, " ";                       ' Leerzeichen zwischen Bytes
  NEXT I
END                                       ' Ende Unterprogramm
```

Siehe auch: ADD\$, OR\$, BIT_MIRR\$, XOR\$, Operator BITAND

ANIB_BYTE\$

ANIB_BYTE\$ (String, Tabelle)

ANIB_BYTE\$: De-Komprimiert Nibbles wieder zu Bytes

Funktion: Dekomprimiert **absolute Nibbles** aus **String** wieder in Bytes. Die Dekomprimierung ist die inverse Funktion zu BYTE_ANIB\$.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu dekomprimierenden Zeichen in Form von Nibbles.

Tabelle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die der Komprimierung zugrundeliegenden absoluten Schwellenwerte.

Funktionswert ist vom Typ STRING und enthält den dekomprimierten String.

ANIB_BYTE\$ erweitert die Datengröße der Nibbles im String von 4 Bit auf jeweils ganze Bytes (8 Bit). Der 8-Bit-Wertebereich wird dazu in 16 Bereiche aufgeteilt, die in einem Nibble von 0 bis 15 adressierbar sind. Das zweite String-Argument der Funktion enthält die Tabelle, die mit genau 16 Bytes die (absoluten) Endwerte der Bereiche festlegt. Die Funktion ANIB_BYTE\$ ordnet jedem Nibble des Quellstrings den entsprechenden Bereichs-Endwert aus der Tabelle zu und legt im Ergebnis-String das Byte ab.

Sinnvollerweise ist der hier verwendete Tabellenstring dem bei der Komprimierung verwendeten angepasst. Der Aufbau des Tabellen-Strings muß genau diesen Kriterien entsprechen:

- enthält genau 16 Bytes
- das erste Byte ist 0
- es kommen niemals 2 gleiche Werte vor
- Werte sind monoton steigend

Beachte: wenn die Länge des Tabellen-Strings nicht genau 16 Bytes beträgt, dann wird folgende interne Standard-Tabelle verwendet:

"<0><1><2><3><4><6><9><13><20><28><39><53><73><104><141><199>"

ANIB_BYTE\$

Programmbeispiel:

```
'-----
' Name: ANIB_BYTE$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I
  STRING A$, B$, T$(16)                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  T$ = "00 04 08 10 20 30 40 50&"       ' Kompressionstabelle
      60 70 80 90 A0 B0 C8 E0"%
  A$ = "<1><22H><34H><55H>"               ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1                 ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  B$ = ANIB_BYTE$ ( A$, T$ )             ' dekomprimiere absolut Nibble
  PRINT #1, "<13><10>T$=";                ' zeige Kompressionstabelle
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(T$,I,1));
  NEXT
  PRINT #1, "<13><10>B$=";                ' zeige Ergebnis in B$
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(B$,I,1));
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                       ' Ende Task MAIN
```

Siehe auch: Komprimierung BYTE_ANIB\$, Incremental-Conversion BYTE_INIB\$, INIB_BYTE\$

ASC

ASC (*String*)

ASC: Gibt den numerischen Wert eines Stringzeichen wieder

Funktion: Liefert den ASCII-Code des ersten Zeichens der Zeichenkette **String** zurück.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.

Funktionswert ist vom Typ LONG und liegt im Bereich von 0 bis 255. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
'-----
' Name: ASC.TIG
'-----
TASK MAIN                                ' Task Beginn MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "<1>ASCII 0A0="; ASC ("A")    ' ASCII-Wert von "A" ausgeben
END                                       ' Task Ende MAIN
```

Siehe auch: VAL_NUM, VAL_REAL, LEN

ASC_BCD\$

ASC_BCD\$

ASC_BCD\$ (*String*)

ASC_BCD\$: Konvertiert: ASCII-HEX Ziffern -> BCD-Bytes

Funktion: Konvertiert alle ASCII-Bytes in **String** in das 'binary coded digit'-Format im Ergebnis-String. Jedes Byte im Ergebnis-String enthält nur ein BCD-codiertes Nibble.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden ASCII-Zeichen im Bereich von '0'...'9' und 'A'...'F' oder 'a'...'f'.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Zeichen in **String** werden in den 4-Bit breiten BCD-Code (binary coded digit) umgesetzt. Die Zeichen 'A'...'F' bzw 'a'...'f' werden als HEX-Ziffern umgesetzt. Alle anderen Zeichen werden im Zielstring zu Null. Am Ende enthält der Zielstring genauso viele Bytes wie der Quellstring.

Beispiel:

Quellstring (2 Bytes):	4	F
Binäre Darstellung:	34H (00110100)	46H (01000110)
Ergebnis-String (2 Bytes):	00000100	00001111

Programmbeispiel:

```

-----
' Name: ASC_BCD$.TIG
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"          ' fuer Ausgabe als HEX
  A$ = "4F"                             ' das is A$
  PRINT #1, "A$=";A$;"=HEX:";          ' zeige A$
  PRINT_USING #1, ASC(LEFT$(A$,1));" ";ASC(RIGHT$(A$,1))
  B$ = ASC_BCD$ ( A$ )                  ' wandele nach BCD
  PRINT #1, "after ASC_BCD$"
  PRINT #1, "B$=  HEX:";                ' zeige Ergebnis in B$
  PRINT_USING #1, ASC(LEFT$(B$,1));" ";ASC(RIGHT$(B$,1))
END                                     ' Ende Task MAIN

```

3

Siehe auch: ASC_PBCD\$, BCD_ASC\$, PBCD_ASC\$, BCD_PBCD\$, PBCD_BCD\$

ASC_PBCD\$

ASC_PBCD\$

ASC_PBCD\$ (*String*)

ASC_PBCD\$: Konvertiert: ASCII-HEX Ziffern zu „packed BCD“ Bytes

Funktion: Konvertiert alle ASCII-Bytes in **String** in das 'packed-binary-digit'-Format im Ergebnis-String. Jedes Byte im Ergebnis-String enthält zwei BCD-codierte Nibbles.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden Zeichen im Bereich von '0'...'9' und 'A'...'F' oder 'a'...'f'.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Zeichen in **String** werden in den 4-Bit breiten BCD-Code (binary coded digit) umgesetzt. Die Zeichen 'A'...'F' bzw 'a'...'f' werden als HEX-Ziffern umgesetzt. Alle anderen Zeichen werden im Zielstring zu Null. Die Bytes im Zielstring enthalten in ihren beiden Nibbles je zwei umgesetzte Zeichen. Am Ende enthält der Zielstring halbsoviele Bytes wie der Quellstring.

Beispiel:

Quellstring (2 Bytes):	4	F
Binäre Darstellung:	34H (00110100)	46H (01000110)
Ergebnis-String (1 Byte):	01001111	

Programmbeispiel:

```

'-----
' Name: ASC_PBCD$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"          ' fuer Ausgabe als HEX
  A$ = "4F"                             ' das is A$
  PRINT #1, "A$=";A$;"=HEX:";          ' zeige A$
  PRINT_USING #1, ASC(LEFT$(A$,1));" ";ASC(RIGHT$(A$,1))
  B$ = ASC_PBCD$ ( A$ )                 ' wandele nach packed BCD
  PRINT #1, "after ASC_PBCD$"
  PRINT #1, "B$=  HEX:";
  PRINT_USING #1, ASC(B$);
END                                      ' Ende Task MAIN

```

3

Siehe auch: ASC_BCD\$, BCD_ASC\$, PBCD_ASC\$, BCD_PBCD\$, PBCD_BCD\$

ASIN

ASIN

ASIN (Wert)

ASIN: ARCUS SINUS, Real-Werte

Funktion: Liefert den Arcussinus von **Wert** im Bogenmaß (rad) zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -1.0 bis 1.0.

Funktionswert ist vom Typ REAL und liegt im Bereich von -PI/2 bis PI/2 (PI=3.14..). Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{rad} * 180/\text{PI}$

Programmbeispiel:

```
' -----
' Name: ASIN.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  REAL Arg                              ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  FOR L = -100 TO 100                    ' Werte von -100 bis 100
    Arg = L / 100.0                      ' Arg fuer ASIN erzeugen
    USING "NF<0><1> <2>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT #1, "<1>ASIN (";                ' Ausgabe auf LC-Display
    PRINT USING #1, Arg; " )"            ' Argument ausgeben
    USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
    PRINT USING #1, "="; ASIN (Arg)      ' Arcussinuswert ausgeben
    WAIT DURATION 1000                  ' 1 sek warten
  NEXT                                  ' naechster Wert
END                                     ' Ende Task MAIN
```

Siehe auch: SIN, COS, TAN, COT, ACOS, ATAN, SINH

ATAN

ATAN (Wert)

ATAN: ARCUS TANGENS, Real-Werte

Funktion: Liefert den Arcustangens von **Wert** im Bogenmaß (rad) zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $-\pi/2$ bis $\pi/2$ ($\pi=3.14..$). Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{rad} \cdot 180/\pi$

Programmbeispiel:

```

' -----
' Name: ATAN.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  REAL Arg                              ' REAL-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  FOR L = -100 TO 100                    ' Werte von -100 bis 100
    Arg = LTR (L)                        ' Arg fuer ACOS erzeugen
    USING "NF<0><1> <2>V0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0"
    PRINT #1, "<1>ATAN (";                ' Ausgabe auf LC-Display
    PRINT USING #1, Arg; " )"            ' Argument ausgeben
    USING "NF<0><1> <15>V0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0"
    PRINT USING #1, "="; ATAN (Arg)      ' Arcustangenswert ausgeben
    WAIT DURATION 1000                  ' 1 sek warten
  NEXT                                  ' naechster Wert
END                                    ' Ende Task MAIN

```

Siehe auch: SIN, COS, TAN, COT, ASIN, ACOT, SINH

BCD_ASC\$

BCD_ASC\$

BCD_ASC\$ (*String*)

BCD_ASC\$: Konvertiert: BCD -> ASCII-HEX
(nur in Projektmodell 'PM_FULL')

Funktion: Konvertiert alle unteren Nibbles in **String** in das ASCII-Format im Ergebnis-String.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden Zeichen im unteren Nibble.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die unteren Nibbles in **String** werden in den ASCII-Code umgesetzt. Die Werte 10...15 werden in die HEX-Zeichen 'A'...'F' umgesetzt. Am Ende enthält der Zielstring genauso viele Bytes wie der Quellstring.

Beispiel:

Quellstring (2 Bytes):	00000100	00001111
Ergebnis-String (2 Bytes):	4	F
Binäre Darstellung:	34H (00110100)	46H (01000110)

Programmbeispiel:

```

'-----
' Name: BCD_ASC$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$ = "<04><0FH>"                       ' das is A$
  PRINT #1, "A$=  HEX:";                ' zeige A$
  PRINT_USING #1, ASC(LEFT$(A$,1));" ";ASC(RIGHT$(A$,1))
  B$ = BCD_ASC$ ( A$ )                  ' wandele nach ASCII
  PRINT #1, "after BCD_ASC$"
  PRINT #1, "B$=";B$;"=HEX:";          ' zeige Ergebnis in B$
  PRINT_USING #1, ASC(LEFT$(B$,1));" ";ASC(RIGHT$(B$,1))
END                                      ' Ende Task MAIN

```

3

Siehe auch: ASC_BCD\$, ASC_PBCD\$, PBCD_ASC\$, BCD_PBCD\$, PBCD_BCD\$

BCD_PBCD\$

BCD_PBCD\$

BCD_PBCD\$ (String)

BCD_PBCD\$: Konvertiert: BCD -> „packed BCD“
(nur in Projektmodell 'PM_FULL')

Funktion: Konvertiert alle Bytes in **String** in das 'packed-binary-digit'-Format im Ergebnis-String. Jedes Byte im Ergebnis-String enthält zwei BCD-codierte Nibbles.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden Zeichen im Bereich von '0'...'9' und 'A'...'F' oder 'a'...'f' im unteren Nibble.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die unteren Nibbles in **String** werden in das 'packed-binary-coded-digit' umgesetzt. Die Bytes im Zielstring enthalten in ihren beiden Nibbles je zwei umgesetzte Zeichen. Am Ende enthält der Zielstring halbsoviele Bytes wie der Quellstring.

Beispiel:

Quellstring (2 Bytes):	00000100	00001111
Ergebnis-String (1 Byte):	01001111	

Programmbeispiel:

```
'-----
' Name: BCD_PBCD$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$ = "<04><0FH>"                       ' das is A$
  PRINT #1, "A$="; "  HEX: ";           ' zeige A$
  PRINT USING #1, ASC(LEFT$(A$,1)); " ";ASC(RIGHT$(A$,1))
  B$ = BCD_PBCD$ ( A$ )                 ' wandele nach packed BCD
  PRINT #1, "after BCD_PBCD$"
  PRINT #1, "B$=  HEX: ";               ' zeige Ergebnis in B$
  PRINT USING #1, ASC(B$);
END                                       ' Ende Task MAIN
```

Siehe auch: ASC_BCD\$, ASC_PBCD\$, PBCD_ASC\$, PBCD_BCD\$

BIT

BIT (Wert, Position)

BIT: Testet Wert des gewünschten Bits in einem Wert

Funktion: Liefert 1 als Ergebnis, wenn das Bit an der durch Position angegebenen Stelle in Wert gesetzt ist, sonst 0.

Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.
Position	ist eine Variable, eine Konstante oder ein Ausdruck Typ LONG in Bereich von 0 bis 31 und gibt die Position des Bits an, welches getestet wird.
Funktionswert	ist vom Typ LONG und ist 1, wenn das Bit an der angegebenen Position gesetzt ist, sonst 0. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```

' -----
' Name: BIT.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG N                                ' N ist LONG-Variable
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UD<1><1> 0.0.0.0.0"              ' Format fuer Binaer-Ausgabe
  PRINT #1, "118dec = ";                  ' Ausgabe Dezimalwert
  FOR N = 7 TO 0 STEP -1                  ' Schleife ueber alle Bits
    PRINT USING #1, BIT (118,N);           ' Ausgabe des Bit-Werts
  NEXT                                     ' naechstes Bit
  PRINT #1, "b"                           ' Ausgabe der Binaer-Kennung
END                                         ' Ende Task MAIN

```

Siehe auch SET_BIT, RES_BIT, BITNOT, BIT_MIRR und Operatoren BITAND, BITOR, BITXOR

BIT_DEMIX\$

BIT_DEMIX\$

BIT_DEMIX\$ (Wert)

BIT_DEMIX\$: Entwürfelt Bits in String lt. Verwürfelungs-String
(nur in Projektmodell 'PM_FULL')

Funktion: Liefert **Wert** mit invertierten Bits zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ
BYTE, WORD oder LONG.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung
findet automatisch statt.

BITNOT setzt alle 1-Bits auf 0 und alle 0-Bits auf 1.

Programmbeispiel:

```
'-----
' Name: BIT_MIX$1.TIG
' gleichzeitig Beispiel fuer BIT_DEMIX$
' Einfaches Beispiel: mixt Bit-0 und Bit-1
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING SRC$, IDX$, MIXED$              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "A"
  IDX$ = &                                ' Mix-Indizes
"01 00 02 03 04 05 06 07"%              ' 0-te Bit an 1, 1.Bit an 0
  PRINT #1, "   SRC$=";SRC$              ' zeige SRC$
  MIXED$ = BIT_MIX$ ( SRC$, IDX$, 3 )     ' mixen
  PRINT #1, "   MIXED$=";MIXED$          ' zeige den gemixten String
  DEMIXED$ = BIT_DEMIX$ ( MIXED$, IDX$, 3 ) ' de-mixen
  PRINT #1, "DEMIXED$=";DEMIXED$         ' zeige den entmixten String
END                                       ' Ende Task MAIN
```

Siehe auch BIT_MIX\$

BIT_MIRR

BIT_MIRR (Wert, Anzahl)

BIT_MIRR: Bit-Vertauschungen in einer Variablen (Spiegelung)

Funktion: Spiegelt bitweise die angegebene **Anzahl** Bits von **Wert** und setzt die darüberliegenden Bits des Funktionswertes auf 0.

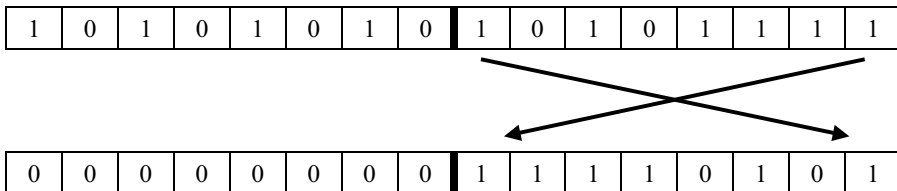
Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG in Bereich von 2 bis 31 und gibt die Anzahl der zu spiegelnden Bits an.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

BIT_MIRR spiegelt die Bits in Wert vom niederwertigsten Bit bis zu dem Bit, welches durch Anzahl spezifiziert ist. Die Ergebnisbits stehen wieder im selben Bereich. Nicht gespiegelte Bits werden auf 0 gesetzt.

Beispiel mit Wert = 0AAAFh vom Typ WORD: **BIT_MIRR (Wert, 8)**



BIT_MIRR

Programmbeispiel:

```
'-----
' Name: BIT_MIRR.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I                                ' I ist BYTE-Variable
  I = 10110101b                          ' Variable vorbesetzen
  DIR_PORT 8,0                            ' Port 8 als Ausgang
  LOOP 9999999                            ' sehr viele Loops
    OUT 8, 255, I                          ' Variable an Port 8 ausgeben
    WAIT_DURATION 1000                     ' 1 Sekunde warten
    I = BIT_MIRR (I,4)                     ' Die unteren 4 Bits spiegeln
  ENDLOOP                                  '
END                                         ' Ende Task MAIN
```

Siehe auch: BYTE_MIRR, BITNOT, MIRROR\$ und Operatoren BITAND, BITOR, BITXOR

BIT_MIRR\$

BIT_MIRR\$ (*String*)

BIT_MIRR\$: Kehrt Reihenfolge der Bits um in Zeichen eines Strings

Funktion: Spiegelt alle Bits in **String** byteweise.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu bearbeitenden Zeichen.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Bits der Zeichen in **String** werden byteweise gespiegelt. Dabei werden alle Bytes aus dem Quellstring gespiegelt in den Zielstring übernommen.

Beispiel:

Quellstring (2 Bytes):	01111111	00000110
Ergebnis-String:	11111110	01100000

BIT_MIRR\$

Programmbeispiel:

```
' -----
' Name: BIT_MIRR$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "<7FH><6>"                        ' das is A$
  PRINT #1, "0:";                        ' zeige A$
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = BIT_MIRR$ ( A$ )                  ' A$ byteweise spiegeln
  PRINT #1, "1:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
END                                        ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )              ' zeige CNT Bytes binaer
  BYTE B, I, J                            ' lokale Variable
  FOR I = 1 TO CNT                        ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ) )   ' hole ein Byte
    FOR J = 1 TO 8                        ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN            ' teste ob bit 1 ist
        PRINT #1, "1";                   ' drucke Bit
      ELSE                                ' falls kein Rest
        PRINT #1, "0";                   ' drucke Bit
      ENDIF
      B = B SHL 1                          ' B eins nach rechts (neues Bit)
    NEXT J
    PRINT #1, " ";                        ' Leerzeichen zwischen Bytes
  NEXT I
END                                        ' Ende Unterprogramm
```

Siehe auch: ADD\$, AND\$, OR\$, MIRROR\$, XOR\$, BIT_MIRR

BIT_MIX\$

BIT_MIX\$ (Src\$, Idx\$, Anz)

BIT_MIX\$ (Src\$, Flash_adr ,Len, Adrbits)

BIT_MIX\$: Verwürfelt Bits in String lt. Verwürfelungs-String
(nur in Projektmodell 'PM_FULL')

Funktion: Mixt die Bits in **Src\$** nach Vorgabetabelle in **Idx\$** oder einer Tabelle im Flash.

Src\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu mixenden Bits.

Idx\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die dem Mixvorgang zugrundeliegende Tabelle. Idx\$ kann auch eine Flash-Adresse (LONG) sein.

Flash_adr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG und enthält die Flash-Adresse der dem Mixvorgang zugrundeliegende Tabelle.

Len ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der Tabelle im Flash an. Dieser Parameter entfällt, wenn ein String übergeben wird.

Adrbits ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Adreßbits in den Bytes von Idx\$ an.

Funktionswert ist vom Typ STRING und enthält den bit-gemixten String.

BIT_MIX\$ vermischt die Bits eines Quellstrings nach Belieben und überträgt sie in einen Zielstring. Für jedes Quellbit wird eine eigene Zielbit-Adresse verwendet, die in einem Conversions-String vorliegt. Die Bytes in Idx\$ dienen als Adresse für die neue Position der Quellbits im Zielstring. Der Zielstring ist nach dem Mixvorgang genauso lang wie der Quellstring und enthält die gemixten Bits. Der Index-String enthält so viele Bytes wie der Quellstring zu mixende Bits enthält, also 8 Mal so viele Bytes wie Src\$.

In dem Index-String liegen die Zieladressen der Bits vor. Um nicht zwei Quellzeichen auf den gleichen Platz zu mixen, d.h. um Überschreiben zu vermeiden, darf kein Wert doppelt vorkommen. Der Aufbau des Adreß-Strings sollte diesen Kriterien entsprechen:

BIT_MIX\$

3

- enthält genausoviel Bytes wie der Quellstring
- alle Werte von 0 bis $8*(\text{LEN}(\text{Quellstring})-1)$ kommen vor
- Idx\$ enthält immer BYTE-Offsets + BIT-Adressen
- Idx\$ kann 1 Byte...16 KByte lang sein
- MIX_BIT Abbildungen sollen 1:1-deutig sein
- Jeder Index kommt nur genau 1-mal vor
- Der kleinste Index ist immer = 0
- Der größte Index ist immer = $8*(\text{LEN}(\text{Quellstring})-1)$
- Wenn vollständig zugewiesen wird, hat der Zielstring immer die gleiche Länge wie der Quellstring

Die Bitadresse im Zielstring muß bereits existieren, d.h. der Zielstring wird nicht verlängert durch BIT_MIX\$.

Zu den Anwendungen zählen:

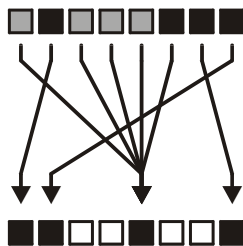
- Chiffrierung von Daten
- individuelle "Re-Distribution" von Bits:
- Einzelne Bits und Bitgruppen können frei plaziert werden
- Verteilung für fehlertolerante Übertragungen:
benachbarte Bits werden weit auseinander verteilt, so daß eine Gruppen-Störung, die mehrere benachbarte Bits beschädigt, weit verteilt wird und dann mit fehlerkorrigierenden Methoden ganz entfernt werden kann.

Um ein Bit innerhalb eines Bytes zu adressieren werden 3 Bit für die Adresse benötigt. Zusätzliche Bits adressieren Bytes innerhalb des Zielstrings. Von 8 Adreßbits bleiben z.B. 5 übrig, um Bytes zu Adressieren, d.h. mit 8-Bit-Adressen kann die Funktion BIT_MIX\$ einen String mit 32 Bytes adressieren. Den Aufbau der Bit-Zieladressen verdeutlicht folgende Darstellung, zunächst für 1 Byte pro Adresse:

Anzahl Adreßbits	Bit	7	6	5	4	3	2	1	0
3							Bit	Bit	Bit
4						Byte	Bit	Bit	Bit
5					Byte	Byte	Bit	Bit	Bit
6				Byte	Byte	Byte	Bit	Bit	Bit
7			Byte	Byte	Byte	Byte	Bit	Bit	Bit
8		Byte	Byte	Byte	Byte	Byte	Bit	Bit	Bit

Bildlich gesehen kann eine solche Bit-Verteilung innerhalb eines Bytes stattfinden:

Source Data



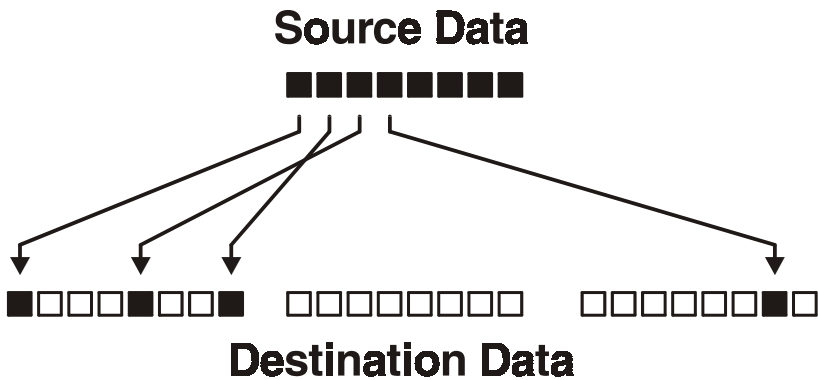
Destination Data

Und so geht es weiter, wenn WORD-Adressen vorliegen, jeweils mit low-Byte zuerst:

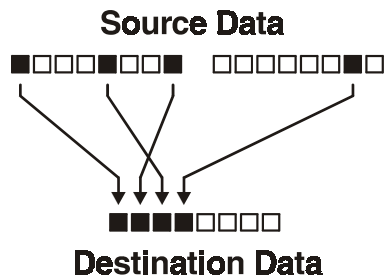
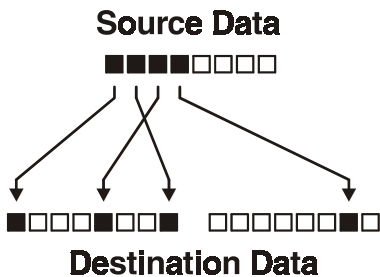
Anzahl Adreßbits	Bit	1. Byte = low Byte								2. Byte = high Byte							
		7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
8		B	B	B	B	B	b	b	b								
9		B	B	B	B	B	b	b	b								B
10		B	B	B	B	B	b	b	b							B	B
11		B	B	B	B	B	b	b	b						B	B	B
12		B	B	B	B	B	b	b	b					B	B	B	B
13		B	B	B	B	B	b	b	b				B	B	B	B	B
14		B	B	B	B	B	b	b	b			B	B	B	B	B	B
15		B	B	B	B	B	b	b	b		B	B	B	B	B	B	B
16		B	B	B	B	B	b	b	b	B	B	B	B	B	B	B	B

BIT_MIX\$

Eine Verteilung über mehrere Bytes könnte so aussehen:



Die Inverse Funktion zu BIT_MIX\$ ist BIT_DEMIX\$. Der Indexstring zum Entmischen ist der gleiche, wie er zum Mixen verwendet wurde.



Das folgende Beispiel zeigt einen Mix- und Demixvorgang mit zunächst nur einem Zeichen. Gemixt wird nur Bit-0 und Bit-1, die vertauscht werden:

Programmbeispiel:

```

-----
' Name: BIT_MIX$1.TIG
' gleichzeitig Beispiel fuer BIT_DEMIX$
' Einfaches Beispiel: mixt Bit-0 und Bit-1
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING SRC$, IDX$, MIXED$              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "A"
  IDX$ = &                              ' Mix-Indizes
"01 00 02 03 04 05 06 07"%              ' 0-te Bit an 1, 1.Bit an 0
  PRINT #1, "      SRC$=";SRC$           ' zeige SRC$
  MIXED$ = BIT_MIX$ ( SRC$, IDX$, 3 )    ' mixen
  PRINT #1, "      MIXED$=";MIXED$       ' zeige den gemixten String
  DEMIXED$ = BIT_DEMIX$ ( MIXED$, IDX$, 3 )' de-mixen
  PRINT #1, "DEMIXED$=";DEMIXED$         ' zeige den entmixten String
END                                        ' Ende Task MAIN

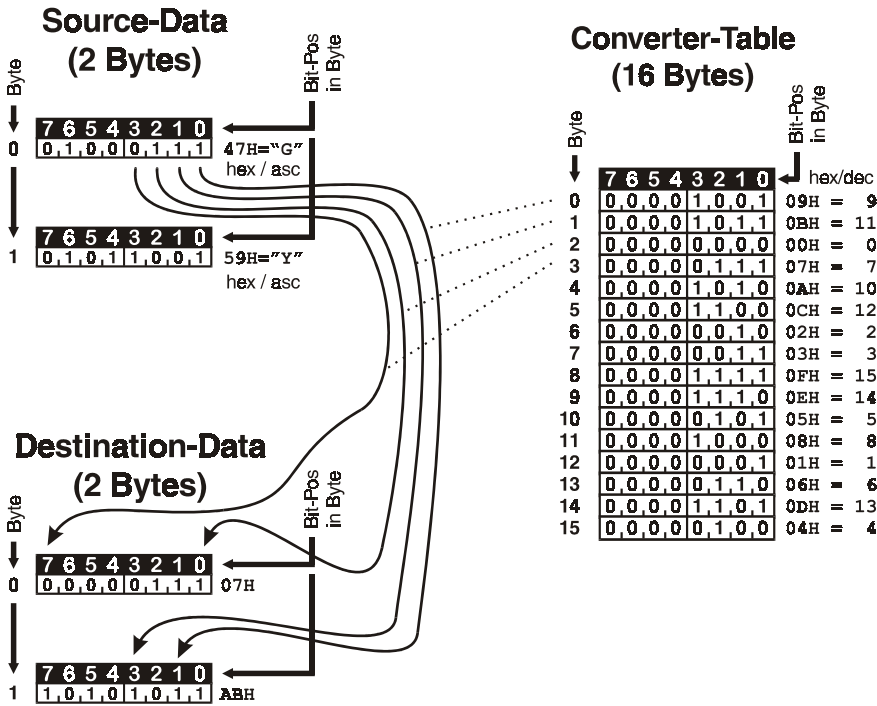
```

3

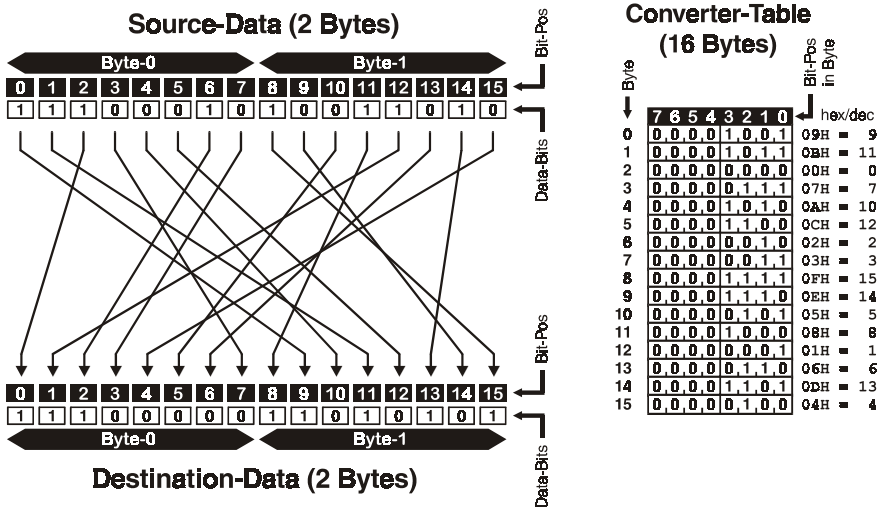
Das nächste Beispiel zeigt, wie Bits aus einem Byte in ein anderes Byte gemixt werden. Es werden alle Bits über zwei Bytes hinweg gespiegelt und in Zweiergruppen ausgetauscht.

Beachten Sie, daß im String IDX\$ vom Bit 0 aufwärts zählend die Bits der Reihe nach aufgeführt werden, in der Darstellung jedoch gemäß der üblichen Darstellungsweise von Zahlen (also auch Binärzahlen) die niederwertigen Bits rechts innerhalb des Bytes stehen:

BIT_MIX\$



Die komplette Darstellung sieht recht verwirrend aus:



Programmbeispiel:

```

'-----
' Name: BIT_MIX$2.TIG
' gleichzeitig Beispiel fuer BIT_DEMIX$
' Vertauscht alle Bits ueber 2 Bytes in 2er-Gruppen
' und stellt sie binaer dar
'-----
user_var_strict                                ' Vars deklarieren!
TASK MAIN                                     ' Beginn Task MAIN
  STRING SRC$, IDX$, MIXED$, DEMIXED$         ' STRING-Variable deklarieren
  LONG L
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "<0AAh><66h>"                        ' 00001111 11110000
  CALL SHOW_BINARY_2BYTES ( SRC$ )            ' zeige SRC$ binaer

  IDX$ = &                                     ' Mix-Indizes
  "09 0B 00 07 0A 0C 02 03 0F 0E 05 08 01 06 13 04"%

  MIXED$ = "<0><0>"
  MIXED$ = BIT_MIX$ ( SRC$, IDX$, 4 )         ' mixen, 4 Adressbits
  CALL SHOW_BINARY_2BYTES ( MIXED$ )         ' zeige MIXED$ binaer

  DEMIXED$ = "<0><0>"
  DEMIXED$ = BIT_DEMIX$ ( MIXED$, IDX$, 4 )   ' de-mixen
  CALL SHOW_BINARY_2BYTES ( DEMIXED$ )       ' zeige den entmixten String
END                                           ' Ende Task MAIN

'-----
' Name: SHOW_BINARY_2BYTES
' zeigt 2 Bytes aus String binaer an
'-----
SUB SHOW_BINARY_2BYTES ( STRING BB$ )
  STRING B$
  LONG L

  IF LEN ( BB$ ) > 1 THEN
    L = ASC(MID$(BB$,0,1))                    ' erstes Byte
    CALL CVT_BINARY ( L, 8, B$ )              ' binaer wandeln
    PRINT #1, B$;" ";                        ' zeige an
    L = ASC(MID$(BB$,1,1))                    ' zweites Byte
    CALL CVT_BINARY ( L, 8, B$ )              ' binaer wandeln
    PRINT #1, B$                              ' zeige an
  ENDIF
END

'-----
' Name: CVT_BINARY
' converts a number into binary representation in a string
'-----
SUB cvt_binary ( &
  LONG bin_no; &                             ' Zahl zu konvertieren
  BYTE n; &                                   ' Anzahl Bits zu konvertieren
  VAR STRING b$ )                             ' in diesen String

```

BIT_MIX\$

3

```
BYTE i
LONG b

b = bin_no
b$=""
for i = 1 to n
    if bit ( b, 0) = 1 THEN
        b$ = "1" + b$
    else
        b$ = "0" + b$
    endif
    b = b shr 1
next
END ' end cvt_binary
```

' beginne mit b\$ = leer
' soviele Bits
' bei Division mit Rest
' '1' nach vorn
' wenn kein Rest
' '0' nach vorn
' naechstes Bit ranschieben

Siehe auch: BIT_DEMIX\$, BYTE_MIX\$, BYTE_DEMIX\$, DISTRIBUTES\$, SCRAMBLES\$

BITNOT

BITNOT (Wert)

BITNOT: Invertiert alle Bits eines numerischen Wertes

Funktion: Liefert **Wert** mit invertierten Bits zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

BITNOT setzt alle 1-Bits auf 0 und alle 0-Bits auf 1.

Programmbeispiel:

```

'-----
' Name: BITNOT.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I                                ' I ist BYTE-Variable
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2> k 0.0.0.0"              ' Ausgabe-Format ist HEX
  PRINT USING #1, "BITNOT(66H) = "; BITNOT(66H) ' Ausgabe BITNOT
END                                        ' Ende Task MAIN

```

Siehe auch: BIT_MIRR, SET_BIT, RES_BIT, und Operatoren BITAND, BITOR, BITXOR

BYTE_ANIB\$

BYTE_ANIB\$

BYTE_ANIB\$ (*String, Tabelle*)

BYTE_ANIB\$: Komprimiert Bytes zu Nibbles, Datenreduktion

Funktion: Reduziert die Datengröße der Bytes im **String** von 8 Bit pro Byte auf jeweils 4 Bit-Nibbles. Diese Komprimierung erfolgt durch Absolutwert-Zuweisung und ist verlustbehaftet.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu komprimierenden Zeichen.

Tabelle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die der Komprimierung zugrundeliegenden absoluten Schwellenwerte.

Funktionswert ist vom Typ STRING und enthält den komprimierten String.

BYTE_ANIB\$ reduziert die Datengröße der Bytes im String von 8 Bit auf jeweils 4 Bit-Nibbles. Der 8-Bit-Wertebereich wird dazu in 16 Bereiche aufgeteilt, die in einem Nibble von 0 bis 15 adressierbar sind. Das zweite String-Argument der Funktion enthält die Tabelle, die mit genau 16 Bytes die (absoluten) Endwerte der Bereiche festlegt. Die Funktion BYTE_ANIB\$ ordnet jedes Byte des Quellstrings dem entsprechenden Bereich zu und legt im Ergebnis-String das zugehörige Nibble ab. Wenn eine ungerade Anzahl Nibbles entsteht, dann wird das letzte Nibble wiederholt.

Der Aufbau des Tabellen-Strings muß genau diesen Kriterien entsprechen:

- enthält genau 16 Bytes
- das erste Byte ist 0
- es kommen niemals 2 gleiche Werte vor
- Werte sind monoton steigend

Beachte: wenn die Länge des Tabellen-Strings nicht genau 16 Bytes beträgt, dann wird folgende interne Standard-Tabelle verwendet:

"<0><1><2><3><4><6><9><13><20><28><39><53><73><104><141><199>"

Eine Anwendung für diese Funktion ist z.B. die Kompression von aufgenommenen Sprachdaten oder Sounds. Die Datenmenge wird auf diese Weise um 50% reduziert und enthält in der Regel immer noch eine gute Sprachverständlichkeit, die mit Over-sampling bei der PWM-Ausgabe noch zu verbessern ist.

Programmbeispiel:

```

'-----
' Name: BYTE_ANIB$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I
  STRING A$, B$, T$(16)                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  T$ = "00 04 08 10 20 30 40 50&"       ' Kompressionstabelle
        60 70 80 90 A0 B0 C8 E0"%
  A$ = "<3><7><8><9><11H><25H><35H>"       ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO 4
    PRINT_USING #1, ASC(MID$(A$,I,1));   ' erste 5 Bytes
  NEXT
  B$ = BYTE_ANIB$ ( A$, T$ )             ' komprimiere absolut Nibble
  PRINT #1, "<13><10>T$=";               ' zeige Kompressionstabelle
  FOR I = 0 TO 4
    PRINT_USING #1, ASC(MID$(T$,I,1));   ' erste 5 Bytes
  NEXT
  PRINT #1, "<13><10>B$=";               ' zeige Ergebnis in B$
  FOR I = 0 TO LEN(B$)-1
    PRINT_USING #1, ASC(MID$(B$,I,1));   ' hier: 4 Bytes
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                       ' Ende Task MAIN

```

3

Siehe auch: Dekomprimierung ANIB_BYTE\$, Incremental-Conversion
 BYTE_INIB\$

BYTE_DEMIX\$

BYTE_DEMIX\$ (*Src\$, Idx\$*)

BYTE_DEMIX\$ (*Src\$, Flash_adr, Len*)

BYTE_DEMIX\$: Entwürfelt Bytes in String lt. Verwürfelungs-String
(nur in Projektmodell 'PM_FULL')

3

Funktion: Ist die inverse Funktion zu BYTE_MIX\$ und mixt die Bytes in **Src\$** nach Vorgabetabelle in **Idx\$** oder einer Tabelle im Flash.

Src\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu de-mixenden Zeichen.

Idx\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die dem De-mixvorgang zugrundeliegende Tabelle.

Funktionswert ist vom Typ STRING und enthält den gemixten String.

BYTE_DEMIX\$ scannt die Bytes in Idx\$ ab und holt die Quellbytes von der Position, die durch den Indexwert in Idx\$ angegeben ist. Da es sich um Byte-Indizes handelt, ist der Quellstring nicht länger als 256 Bytes. Der Zielstring ist nach dem De-Mixvorgang genauso lang wie der Quellstring und enthält die ent-mixten Bytes. Auch der Indexstring ist genauso lang wie der Quellstring.

Der Indexstring zum Entmischen ist der gleiche, wie er zum Mixen verwendet wurde. In dem Index-String kommen alle Werte von 0 bis zu der Länge des Quellstrings minus 1 vor. Um nicht zwei Quellzeichen auf den gleichen Platz zu mixen, d.h. um Überschreiben zu vermeiden, darf kein Wert doppelt vorkommen. Der Aufbau des Tabellen-Strings muß genau diesen Kriterien entsprechen:

- enthält genau soviele Bytes wie der Quellstring
- ist nicht länger als 256
- es kommen niemals 2 gleiche Werte vor
- alle Werte von 0 bis LEN(Quellstring)-1 kommen vor

Das folgende Beispiel zeigt einen Mix- und Demixvorgang mit 6 Zeichen:

Mixen				De-Mixen			
Pos.	Ouelle	Tue an Pos.	Ziel	Pos.	Ouelle	hole von Pos.	Ziel
0	B	3	I	0	I	3	B
1	E	5	R	1	R	5	E
2	R	1	N	2	N	1	R
3	L	4	B	3	B	4	L
4	I	0	L	4	L	0	I
5	N	2	E	5	E	2	N

3

Programmbeispiel:

```

' -----
' Name: BYTE_MIX$1.TIG
' gleichzeitig Beispiel fuer BYTE_DEMIX$
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING SRC$, IDX$, MIXED$              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "Hello World ABCDE"
  IDX$ = &                                ' Mix-Indizes
  "05 0E 02 08 10 0A 0D 01 0F 06 03 0C 09 04 07 00 0B"%
  PRINT #1, SRC$                          ' zeige SRC$
  MIXED$ = BYTE_MIX$ ( SRC$, IDX$ )        ' mixen
  PRINT #1, MIXED$                        ' zeige den gemixten String
  DEMIXED$ = BYTE_DEMIX$ ( MIXED$, IDX$ ) ' de-mixen
  PRINT #1, DEMIXED$                      ' zeige den entmixten String
END                                         ' Ende Task MAIN

```

Siehe auch: BYTE_MIX\$, BIT_MIX\$, BIT_DEMIX\$, SCRAMBLE\$, DISTRIBUTE\$

BYTE_INIB\$

BYTE_INIB\$

BYTE_INIB\$ (String, Tabelle, Start)

BYTE_INIB\$: Komprimiert Bytes zu Incremental-Nibbles
(nur in Projektmodell 'PM_FULL')

Funktion: Reduziert die Datengröße der Bytes im **String** von 8 Bit pro Byte auf jeweils 4 Bit-Nibbles. Diese Komprimierung erfolgt durch Inkrementalwert-Zuweisung und ist verlustbehaftet.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu komprimierenden Zeichen.

Tabelle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die der Komprimierung zugrundeliegenden Inkrementalwerte.

Start ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 0...255 und bestimmt den Startwert zur Bildung der Differenz zum ersten Byte des Quellstrings.

Funktionswert ist vom Typ STRING und enthält den komprimierten String.

BYTE_INIB\$ reduziert die Datengröße der Bytes im String von 8 Bit auf jeweils 4 Bit-Nibbles. Die Nibbles adressieren einen von 16 Werten in der Inkremental-Tabelle, der der Differenz zwischen zwei aufeinanderfolgenden 8-Bit-Werten des Quellstrings am nächsten kommt. Die Werte im Tabellenstring werden als vorzeichenbehaftete Bytes gesehen (-1 = FFH, -127 = 80H). Die Funktion BYTE_INIB\$ ordnet jede Differenz von einem Byte Quellstring zum nächsten dem entsprechenden Inkrementalwert zu und legt im Ergebnis-String das zugehörige Nibble ab.

Der Aufbau des Inkremental-Strings muß genau diesen Kriterien entsprechen:

- enthält genau 16 Bytes
- Bytes 0...7 enthalten Werte von 0...7FH (positiv, steigend)
- Bytes 8...15 enthalten Werte von FFH...80H (negativ -1...-128 fallend)
- es kommen niemals 2 gleiche Werte vor
- die positiven Werte sind monoton steigend
die negativen Werte sind monoton fallend

Beachte: wenn die Länge des Tabellen-Strings nicht genau 16 Bytes beträgt, dann wird eine interne Standard-Tabelle verwendet.

Eine Anwendung für diese Funktion ist z.B. die Kompression von aufgenommenen Sprachdaten oder Sounds. Die Datenmenge wird auf diese Weise um 50% reduziert und enthält in der Regel immer noch eine gute Sprachverständlichkeit, die mit Oversampling bei der PWM-Ausgabe noch zu verbessern ist.

Programmbeispiel:

```

'-----
' Name: BYTE_INIB$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I
  STRING A$, B$, T$(16)                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  T$ = "00 01 02 04 08 18 40 7F&
      FF FE FC F8 F0 E0 C0 80"%
  A$ = "<7FH><80H><85H><84H><80H><7FH>"   ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  B$ = BYTE_INIB$ ( A$, T$, 7FH )        ' komprimiere increm. Nibble
  PRINT #1, "<13><10>T$=";              ' zeige Kompressionstabelle
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(T$,I,1));
  NEXT
  PRINT #1, "<13><10>B$=";              ' zeige Ergebnis in B$
  FOR I = 0 TO LEN(B$)-1                 ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(B$,I,1));
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                      ' Ende Task MAIN

```

3

Siehe auch: Incremental-Conversion BYTE_ANIB\$, INIB_BYTE\$

BYTE_MIRR

BYTE_MIRR

BYTE_MIRR (Wert, Anzahl)

BYTE_MIRR: Kehrt Reihenfolge der Bits um in numerischen Variablen

Funktion: Spiegelt byteweise die angegebene **Anzahl** Bytes von **Wert**, und setzt nicht gespiegelte Bytes auf 0

3

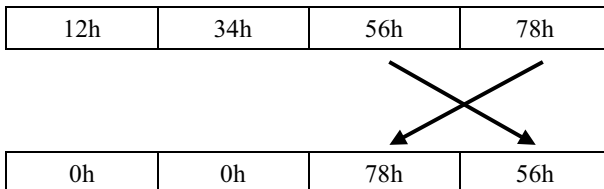
Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 2 bis 4 und gibt die Anzahl der zu spiegelnden Bytes an.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

BYTE_MIRR spiegelt vom niederwertigsten Byte bis zu dem Byte, welches durch Anzahl spezifiziert ist. Die Ergebnisbytes stehen wieder im selben Bereich. Nicht gespiegelte Bytes werden auf 0 gesetzt.

Beispiel: **BYTE_MIRR (12345678h, 2)**



Programmbeispiel:

```

'-----
' Name: BYTE_MIRR.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG I                                ' I ist LONG-Variable
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<8><8> kV2.2.2.2"              ' HEX-Ausgabe bei PRINT USING
  I = 12345678h                          ' Variable besetzen (4 Byte)
  PRINT USING #1, "<1>1.) ";I              ' formatierte Ausgabe auf LCD
  PRINT USING #1, "2.) ";BYTE_MIRR (I,2) ' untere 2 Bytes gespiegelt
END                                       ' Ende Task MAIN

```

3

Siehe auch: BIT_MIRR, BYTE_MIRR

BYTE_MIX\$

BYTE_MIX\$

BYTE_MIX\$ (*Src\$, Idx\$*)

BYTE_MIX\$ (*Src\$, Flash_adr, Len*)

BYTE_MIX\$: Verwürfelt Bytes in String lt. Verwürfelungs-String
(nur in Projektmodell 'PM_FULL')

3

Funktion: Mixt die Bytes in **Src\$** nach Vorgabetabelle in **Idx\$** oder einer Tabelle im Flash.

Src\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu mixenden Zeichen.

Idx\$ ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die dem Mixvorgang zugrundeliegende Tabelle.

Flash_adr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG und enthält die Flash-Adresse der dem Mixvorgang zugrundeliegende Tabelle.

Len ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der Tabelle im Flash an. Dieser Parameter entfällt, wenn ein String übergeben wird.

Funktionswert ist vom Typ STRING und enthält den gemixten String.

BYTE_MIX\$ ordnet die Reihenfolge der Bytes eines Quell-String neu. Jedem Quell-Byte kann eine beliebige neue ZIEL-Position zugewiesen werden. BYTE_MIX kann ein-eindeutige oder auch nicht ein-eindeutige Abbildungen erzeugen. BYTE_MIX\$ verwendet die Bytes in Idx\$ als Indizes für die neue Position der Quellbytes im Zielstring. Da es sich um Byte-Indizes handelt, ist der Quellstring nicht länger als 256 Bytes. Der Zielstring ist nach dem Mixvorgang genauso lang wie der Quellstring und enthält die gemixten Bytes. Auch der Indexstring ist genauso lang wie der Quellstring.

In dem Index-String kommen alle Werte von 0 bis zu der Länge des Quellstrings minus 1 vor. Um nicht zwei Quellzeichen auf den gleichen Platz zu mixen, d.h. um Überschreiben zu vermeiden, darf kein Wert doppelt vorkommen. Der Aufbau des Tabellen-Strings muß genau diesen Kriterien entsprechen:

- enthält genau soviel Bytes wie der Quellstring
- ist nicht länger als 256
- es kommen niemals 2 gleiche Werte vor
- alle Werte von 0 bis LEN(Quellstring)-1 kommen vor

Die Inverse Funktion zu BYTE_MIX\$ ist BYTE_DEMIX\$. Der Indexstring zum Entmischen ist der gleiche, wie er zum Mixen verwendet wurde. Das folgende Beispiel zeigt einen Mix- und Demixvorgang mit 6 Zeichen:

Mixen				De-Mixen			
Pos.	Ouelle	Tue an Pos.	Ziel	Pos.	Ouelle	hole von Pos.	Ziel
0	B	3	I	0	I	3	B
1	E	5	R	1	R	5	E
2	R	1	N	2	N	1	R
3	L	4	B	3	B	4	L
4	I	0	L	4	L	0	I
5	N	2	E	5	E	2	N

Anwendungen für Byte_MIX\$ umfassen:

- Aufbau von Datensätzen in bestimmten Ziel-Formaten
- Daten-Chiffrierung
- Formatierungen

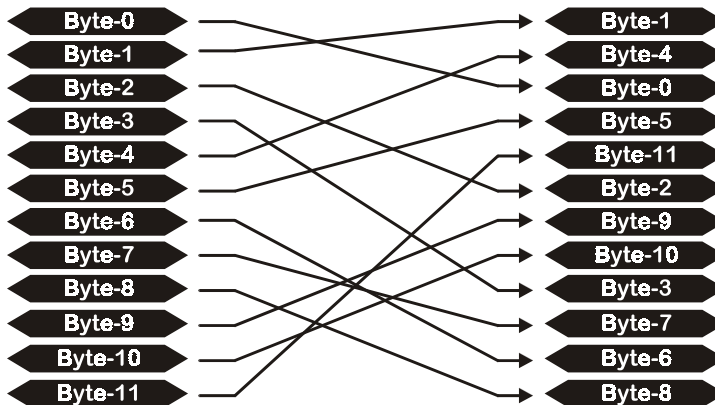
BYTE_MIX\$

Programmbeispiel:

```
-----
' Name: BYTE_MIX$1.TIG
' gleichzeitig Beispiel fuer BYTE_DEMIX$
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING SRC$, IDX$, MIXED$              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "Hello World ABCDE"
  IDX$ = &                                ' Mix-Indizes
  "05 0E 02 08 10 0A 0D 01 0F 06 03 0C 09 04 07 00 0B"%
  PRINT #1, SRC$                          ' zeige SRC$
  MIXED$ = BYTE_MIX$ ( SRC$, IDX$ )      ' mixen
  PRINT #1, MIXED$                        ' zeige den gemixten String
  DEMIXED$ = BYTE_DEMIX$ ( MIXED$, IDX$ ) ' de-mixen
  PRINT #1, DEMIXED$                     ' zeige den entmixten String
END                                       ' Ende Task MAIN
```

Source-Data

Destination-Data



Converter-Table = "02 00 05 08 01 03 0A 09 0B 06 07 04"%

Eine besondere Anwendung für diese Funktion ist z.B. eine Laufschrift. Der Mixvorgang verschiebt dabei den Quellstring jeweils um 1 und setzt das erste Byte an die letzte Stelle.

Programmbeispiel:

```

'-----
' Name: BYTE_MIX$2.TIG
' erzeugt eine rotierende Laufschrift
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I
  STRING SRC$, MIX$                      ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  SRC$ = "Hello World, how are you today? "
  MIX$ = ""
  FOR I = 1 TO LEN ( SRC$) - 2
    MIX$ = MIX$ + CHR$ ( I )            ' Mix-Indizes
  NEXT
  MIX$ = MIX$ + CHR$ ( 0 )              ' erster Buchstabe wird hinten
  LOOP 9999999                          ' viele Loops
  PRINT #1, "<1Bh>A<0><0><0F0h>";LEFT$(SRC$,20);' zeige SRC$
  SRC$ = BYTE_MIX$ ( SRC$, MIX$ )      ' mixen
  WAIT_DURATION 500
ENDLOOP
END                                      ' Ende Task MAIN

```

3

Siehe auch: BYTE_DEMIX\$, BIT_MIX\$, BIT_DEMIX\$, SCRAMBLE\$, DISTRIBUTUTES\$

CALC_CRC

CALC_CRC (*String*, *Position*, *Anzahl*, *Startfaktor*, *Step*)

CALC_CRC: Berechnung einer CRC-Prüfsumme
(nur in Projektmodell 'PM_FULL')

Funktion: Diese Funktion berechnet ab der Position **Pos** über die angegebene **Anzahl** Bytes in **String** eine Prüfsumme nach dem CRC-Verfahren (,cyclic redundant checksum'). Der Startwert und die Schrittweite der Prüfsummenberechnung werden vorgegeben.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Daten, über die die Prüfsumme berechnet werden soll.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition in String an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 1...65535 und legt die Anzahl der zu berücksichtigenden Bytes fest.
Startfaktor	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Startwert des Berechnungsfaktors fest.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Schrittweite des Berechnungsfaktors fest.
Funktionswert	int vom Typ LONG und enthält die Prüfsumme. Eine Typwandlung bei der Zuweisung findet automatisch statt.

CALC_CRC berechnet eine CRC-Prüfsumme aus einem Quellstring. Die Parameter Anzahl und POS geben an, wieviele Bytes ab welcher Position bei der Berechnung berücksichtigt werden. Bei der Berechnung der Prüfsumme werden die Bytes mit einem sich ändernden Faktor multipliziert und dann aufsummiert. Der Startwert des Faktors wird im Argument Start angegeben. Der Wertebereich des Faktors ist 0...65535. Bei jedem neuen Berechnungsschritt wird zu dem Faktor der Wert Step hinzuaddiert. Wenn der Faktor überläuft, dann nimmt er den nächsten kleineren Wert.

1. Byte * (Faktor)	-> aufsummieren
2. Byte * (Faktor + 1 * Step)	-> aufsummieren
3. Byte * (Faktor + 2 * Step)	-> aufsummieren
4. Byte * (Faktor + 3 * Step)	-> aufsummieren
...	-> aufsummieren
n. Byte * (Faktor + (n-1) * Step)	-> aufsummieren

Die Summe wird in einer LONG-Zahl berechnet. Überläufe bleiben unberücksichtigt.

Beachte: Damit die CRC-Prüfsumme einen sinnvollen Schutz bietet, soll der Startwert des Faktors ein ungerader Wert und die Schrittweite ein gerader Wert sein. Damit die maximale Anzahl von Faktoren (=32K) verwendet wird, wählt man die Schrittweite entsprechend. So wiederholen sich die Werte für Faktor, wenn Step=8 gesetzt wird, es entstehen nur 8K Faktoren. Bei Step=6 entstehen 32K Faktoren, da sich die Werte nach dem Überlauf über 65535 nicht wiederholen:

Beispiel: Start = 7, Step = 6

7,13,19,25, ... 65533,3,9,15, ... 65535,5,11,17,...

Programmbeispiel:

```

'-----
' Name: CALC_CRC.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I                                ' FOR loop
  LONG C                                ' Pruefsumme
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
A$ = "<1><2><3><4><5><6><7>"                ' das is A$
PRINT #1, "A$=";                          ' zeige A$
FOR I = 0 TO 4                              ' erste 5 Bytes
  PRINT_USING #1, ASC(MID$(A$,I,1));
NEXT
FOR I = 1 TO 3
  C = CALC_CRC ( A$, 0, I, 6, 7 )          ' Pruefsumme ueber i Bytes
  PRINT #1, "<13><10>CRC=";C;                ' zeige Pruefsumme
NEXT
END                                          ' Ende Task MAIN

```

CALL

CALL

CALL Unterprogrammname [([*Argumentenliste*])]

CALL: Unterprogramm aufrufen

Instruktion: **CALL** ruft ein Unterprogramm auf.

Argumentenliste Es können bis zu 16 Konstanten oder Variable (keine Ausdrücke!) an das Unterprogramm übergeben oder zur Rücklieferung von Variablen benutzt werden. Die Argumente sind durch Kommata voneinander getrennt.

Beachte: Die Argumentliste kann keine Ausdrücke enthalten.

Die Datentypen der übergebenen Argumente müssen mit den im Unterprogramm deklarierten Parametern übereinstimmen (siehe: SUB, Seite 446). Unstimmigkeiten meldet der Compiler als Fehler.

Unterprogramme sind ein sehr leistungsfähiges Werkzeug in Tiger-BASIC®-Programmen. Die systematische Verwendung von Unterprogrammen trägt zu modularer, verständlicher Programmierung bei und reduziert sowohl die resultierende Programmlänge als auch den Debug-Aufwand zum Teil erheblich. Weder die Anzahl von Unterprogrammen, noch ihre Länge sind begrenzt, so lange ausreichend Programm-Speicherplatz (FLASH) zur Verfügung steht. Auch dürfen Unterprogramme grundsätzlich von jedem anderen Unterprogramm und jeder Task aus aufgerufen werden, ohne sich zu behindern. Auch rekursive Konstruktionen - Unterprogramm ruft sich selbst auf - sind erlaubt.

Unterprogramme können mit Variablen unterschiedlichen Geltungsbereichs arbeiten:

Lokale Variable werden im Unterprogramm deklariert. Ihr Geltungsbereich ist begrenzt auf genau den jeweiligen Unterprogramm-Aufruf. Wird ein Unterprogramm von mehreren Tasks gleichzeitig verwendet, so erzeugt jeder Unterprogramm-Aufruf einen eigenen Satz von lokalen Variablen, die nicht miteinander in Beziehung stehen. Keine solcher lokalen Variablen kann von einer anderen Task oder einem anderen Unterprogramm-Aufruf gelesen oder verändert werden. Lokale Variable haben also den Vorteil der „Geheimhaltung“ und „Sicherheit“ nach außen. Lokale Variable werden zur Laufzeit auf dem Stack angelegt (siehe Festlegung der Stack-Größe USER_STACK_SIZE, Seite 40). Sie werden wie alle Variablen im BASIC-Tiger® durch die Deklaration noch nicht initialisiert, besitzen also zunächst einen zufälligen Wert. Die Zahl und Größe von lokalen Variablen, sowie der Verschachtelungsgrad von Unterprogrammen erfordert eine bestimmte Mindestgröße des Stacks. Tritt zur

Laufzeit ein Stack-Überlauf ein, so hat dies einschneidende Wirkung auf den Programm-Ablauf (Task wird gestoppt.). Bis zur Fertigstellung eines Projektes ist die tatsächlich erforderliche Stack-Größe als Worst-Case Betrachtung festzulegen, damit sich in der Anwendung niemals ein Stack-Überlauf ereignen kann.

Will man mit dem erforderlichen Speicherplatz für den Stack haushalten, so führe man sich vor Augen, daß lokale Variablen in Unterprogrammen stets Stack belegen. Insbesondere lokale Strings und Arrays sollten dann möglichst sparsam eingesetzt werden. Strings sollten jeweils explizit auf die wirklich benötigte Größe hin deklariert werden. „STRING A\$ (10) belegt nur 12 Bytes auf dem Stack, während STRING B\$ stets mit der Default-Länge (z.B. 64+2 Bytes) deklariert wird. Bei ARRAYS kann durch geeignete Wahl des Typs, möglichst BYTE oder WORD statt LONG, der Platzbedarf reduziert werden.

Globale Variable können von jedem Unterprogramm und jeder Task aus angesprochen werden. Globale Variable eignen sich zur Kommunikation zwischen Tasks und als universelle Variable, Puffer, Arrays oder Strings mit weitreichender Bedeutung. Werden globale Variable von einem Unterprogramm verändert, ist sorgfältig auf die Plausibilität von Programm-Ablauf und Dateninhalten zu achten! Werden globale Variablen von verschiedenen Tasks aus manipuliert, kann z.B. der Task-Switch-Mechanismus kurzzeitig (!!) angehalten werden, bis die kritische Phase der Manipulation beendet ist (siehe: DISABLE_TSW / ENABLE_TSW).

Ein-/Ausgangs-Parameter

Bis zu 16 Werte und Variablen können als Parameter an ein Unterprogramm übergeben und von ihr zurückgegeben werden. Ein-/Ausgangs-Parameter werden durch Referenz weitergegeben, d.h. das Unterprogramm erhält Zeiger auf den Ort dieser Werte und Variablen. Daher belasten lange Strings, die als Parameter übergeben werden, den Stack nur mit wenigen Referenz-Bytes. Man beachte, daß diese Parameter nicht den Charakter von lokalen Variablen besitzen. Wird eine Ausgangsvariable in einem Unterprogramm verändert, so ändert sich sofort der Wert der entsprechenden Original-Variablen. Reine Eingangs-Variablen sind vor Veränderungen in der Subroutine geschützt.

Bei einem Programm mit intensiver Unterprogrammstruktur, vielen lokalen Variablen, Ein-/Ausgabeparametern, rekursiven Schritten kann es vorkommen, daß der Stack überläuft. Die Folge ist dann ein Runtime-Error, welcher zum Beenden der betroffenen Task führt. Die Task MAIN wird jedoch neu gestartet. (Siehe auch

CALL

‘Fehlerbehandlung während der Laufzeit auf Seite 50 und Compiler-Anweisung USER_STACK_SIZE auf Seite 40).

Anhaltspunkte für die Stackbelegung (ca. Werte):

	Stackbedarf
Unterprogramm-Aufruf:	17 Bytes
BYTE (lokal)	1 Byte
WORD (lokal)	2 Bytes
LONG (lokal)	4 Bytes
REAL (lokal)	8 Bytes
STRING (lokal)	Länge + 2
ARRAY (lokal)	Länge=Anzahl der Elemente * Elementlänge
Ein-Ausgangs-Parameter:	6 Bytes
Globale Variablen:	0 Bytes

Bei einer Stackgröße von 512 Bytes können also z.B. 8 Unterprogramm-Level mit bis zu 64 Byte Stack-Bedarf aktiv sein. Die Stackgröße läßt sich, soweit Speicherplatz vorhanden ist, mit der Compiler-Anweisung USER_STACK_SIZE festlegen. Der Defaultwert ist 2048 Bytes.

Programmbeispiel:

```

'-----
' Name: CALL.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
LONG I                                  ' I ist LONG-Variable
I = 1234                                ' Wert zuweisen
CALL SQUARE (I)                        ' Aufruf der Subroutine
PRINT #1, "I*I =" ; I                  ' Ausgabe von I
END                                     ' Ende Task MAIN

'-----
' Subroutine berechnet Quadrat
'-----
SUB SQUARE ( VAR LONG Z )               ' Definition der Subroutine
Z = Z*Z                                 ' Berechnung
END                                     ' Ende und Ruecksprung

```

Dieses Beispiel (aus dem Verzeichnis 'APPLICATIONS') zeigt den rekursiven Aufruf eines Unterprogramms, d.h. das Unterprogramm ruft sich selbst auf:

Programmbeispiel:

```

'-----
' Name: HANOI.TIG
' Stand: 24.04.1997
' Typ: Tiger-BASIC Source Code
' Zweck: Tuerme von Hanoi, Demonstration fuer rekursiven
'       Unterprogramm-Aufruf, Tastendruck fuer naechsten Schritt.
'-----
ARRAY Disc(4,10) OF BYTE                ' ARRAY fuer Scheiben je Turm
ARRAY Summit(4) OF BYTE                  ' ARRAY fuer Hoehe der Tuerme
BYTE Height, I                           ' globale BYTE-Variablen

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
PRINT #1, "<l>";                          ' Bildschirm loeschen
PRINT #1, "<1Bh>c"; CHR$(0); "<F0h>"; ' Cursor ausschalten
Height = 4                               ' Hoehe des Ausgangs-Turms
FOR I = 1 TO Height                      ' Schleife bis max. Hoehe
  Disc(1,I) = Height-I+1                 ' Scheiben des 1. Turms
  CALL Draw (1,I,Disc(1,I))              ' Scheibe zeichnen
NEXT                                     ' naechste Scheibe
Summit(1)=Height                         ' Hoehe der Spitze von Turm 1
Summit(2)=0                              ' Hoehe der Spitze von Turm 2
Summit(3)=0                              ' Hoehe der Spitze von Turm 3
CALL Move (Height,1,2,3)                 ' Einsprung in Rekursion

```

CALL

3

```

END                                     ' Ende Task MAIN

'-----
' Subroutine zum Bewegen der Scheiben (rekursiv)
'-----
SUB Move (BYTE Depth,From,Over,Towards) ' Definition der Subroutine
  BYTE Temp,Temp2                      ' lokale BYTE-Hilfsvariablen
  STRING A$                            ' lokale STRING-Variable
  IF Depth>0 THEN                      ' Falls Tiefe (bzw. Hoehe) > 0
    Temp=Depth-1                      ' in Hilfsvariable
    CALL Move (Temp,From,Towards,Over) ' rekursiver Aufruf
    FOR Temp = 0 TO 0 STEP 0           ' Endlosschleife bis Temp=1
      RELEASE TASK                     ' Rest der Task-Zeit freigeben
      GET #1, #0, #1, 1, Temp         ' Zeichen in Tastatur-Buffer?
      NEXT                             ' Ende Endlosschleife
      GET #1, 1, A$                   ' Tastatur-Buffer auslesen
      Summit(Towards)=Summit(Towards)+1 ' Hoehe des neuen Turms +1
      Temp2=Summit(From)              ' in Hilfsvariable
      CALL Draw (From,Temp2,0)         ' loesche Scheibe aus Turm/alt
      Temp=Disc(From,Summit(From))     ' in Hilfsvariable
      Temp2=Summit(Towards)            ' in Hilfsvariable
      CALL Draw (Towards,Temp2,Temp)   ' zeichne Scheibe in Turm/neu
      Disc(Towards,Summit(Towards))=Disc(From,Summit(From)) ' Verschieben
      Summit(From)=Summit(From)-1      ' Hoehe des alten Turms -1
      Temp=Depth-1                    ' in Hilfsvariable
      CALL Move (Temp,Over,From,Towards) ' rekursiver Aufruf
    ENDIF
  END
END                                     ' Ende und Ruecksprung

'-----
' Subroutine zur Ausgabe einer Scheibe an bestimmter Position
'-----
SUB Draw (BYTE Tower,Place,Width)      ' Definition der Subroutine
  BYTE Line,Column                     ' lokale BYTE-Variablen

  Column=7*(Tower-1)                  ' Berechnung der Spalte
  Line=4-Place                         ' Berechnung der Zeile
  PRINT #1, "<1Bh>A"; CHR$(Column);& ' Cursor-Positionierung
    CHR$(Line); "<F0h>";
  IF Width=0 THEN                     ' Ausgabe: Scheibe loeschen
    PRINT #1, "<20h><20h><20h><20h><20h><20h>";
  ENDIF
  IF Width=1 THEN                     ' Ausgabe: Scheibe Breite 1
    PRINT #1, "<20h><20h><EFh><EDh><20h><20h>";
  ENDIF
  IF Width=2 THEN                     ' Ausgabe: Scheibe Breite 2
    PRINT #1, "<20h><20h><E9h><E9h><20h><20h>";
  ENDIF
  IF Width=3 THEN                     ' Ausgabe: Scheibe Breite 3
    PRINT #1, "<20h><E9h><E9h><E9h><E9h><20h>";
  ENDIF
  IF Width=4 THEN                     ' Ausgabe: Scheibe Breite 4
    PRINT #1, "<E9h><E9h><E9h><E9h><E9h><E9h>";
  ENDIF
END                                     ' Ende und Ruecksprung

```

CHECK_FLASH

CHECK_FLASH Adresse, Labelname[, Fehlerbehandlung]

CHECK_FLASH: Findet rückwärts erste programmierte Stelle im User-Flash

Instruktion: Kontrolliert einen Block von bis zu 256 Byte im FLASH, bis wohin dieser leer ist, d.h. mit 0FFh Bytes gefüllt ist. Der Test beginnt bei **Adresse**-1 und verläuft absteigend bis maximal zu **Adresse**-1-256. Wird jedoch ein nicht gelöscht Zeichen gefunden, so bleibt **Adresse** auf dem zuletzt gefundenen gelöschten Byte stehen und die Programm-Ausführung verzweigt zu Labelname.

Diese Instruktion trägt den besonderen Eigenschaften des FLASH-Speichers Rechnung. In den FLASH Speicher können jeweils nur 0-Bits geschrieben werden, niemals kann man einen Bitwert 0 durch schreiben (POKE) zum Wert 1 ändern. FLASH Speicher werden daher anders angesprochen, als dies bei normalem Arbeitsspeicher (RAM) der Fall ist. Wird ein FLASH-Speicher z.B. linear mit POKE-Instruktionen beschrieben, und will man nach einem Power-ON erkunden bis zu welcher Adresse ein bestimmter FLASH-Bereich beschrieben ist, so wendet man ggf. wiederholt 'CHECK_FLASH' an, um die höchste noch gelöschte FLASH-Adresse in diesem Bereich zu finden. Ein Beispiel-Programm dazu ist unten abgebildet.

Wird versucht, auf eine nicht existierende FLASH-Adresse zuzugreifen oder ist der FLASH-Speicher z.B. busy, so wird entsprechend der **Fehlerbehandlung** verzweigt. Schreibweise:

ON_ERROR_CALL Unterprogramm

ON_ERROR_GOTO Sprungmarke

Weitere Informationen zur Fehlerbehandlung finden Sie auf Seite 50ff.

Wenn es FLASH-Sektoren gibt, die nicht vom Programm-Code belegt werden, stehen diese einem Tiger-BASIC[®] Programm als Daten-Speicher frei zur Verfügung. Der benutzbare Flash-Speicher beginnt für den Anwender immer bei Adresse 0. Es stehen immer komplette Sektoren zur Verfügung, die byteweise beschrieben und gelesen werden können. Das Löschen erfolgt stets Sektorweise.

Weitere Informationen über den FLASH-Speicher finden Sie im Hardware-Handbuch.

CHECK_FLASH

Programmbeispiel:

3

```
'-----
' Name: CHECK_FLASH.TIG
'-----
' Ermittelt den freien FLASH-Speicher im BASIC-Tiger-Modul
' durch Auffinden der hoechsten noch geloeschten FLASH-Adresse
'-----
#include DEFINE_A.INC          ' allg. Definitionen
USER_VAR_STRICT               ' Variable deklarieren!

TASK MAIN                      ' Start Task MAIN
  LONG GR_SEC, USER_SEC, ADR, FREE  ' LONG-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  GR_SEC = SYSVARN ( FLASH_SSIZE, 0 )  ' Gr. der FLASH-Sektoren
  USER_SEC = SYSVARN ( FLASH_DSEC, 0 )  ' Anz. der FLASH-User-Sektoren
  USER_SEC = USER_SEC - 1
  ERASE_FLASH 0, (USER_SEC-1) * GR_SEC  ' Flash loeschen
  POKE_FLASH 988, 7, 0                  ' beschreibe Flash-Zelle
  ADR = USER_SEC * GR_SEC - 1           ' Adresse berechnen
  LOOP ADR / 256                         ' Soviele 256-Byte-Bloecke
    CHECK_FLASH ADR, NON_FF_EXIT        ' Flash ueberpruefen
  ENDLOOP                               ' naechster Block
  PRINT #1, "Flash is empty"            ' Ausgabe aus LC-Display
  GOTO ENDPR                            ' Sprung zum Programmende

NON_FF_EXIT:                      ' Sprungziel "nicht leer"
  PRINT #1, "NON_FF found at"           ' ab ADR sind Daten <> FF
  PRINT #1, "ADR ="; ADR                ' im FLASH !
  FREE = USER_SEC * GR_SEC - ADR - 1    ' freier Flash-Speicher
  PRINT #1, "Flash free: "; FREE        ' Ausgabe auf LC-Display

ENDPR:                            ' Sprungziel Programmende
  PRINT #1, "--- End ---"              ' Ausgabe auf LC-Display

END                                ' Ende Task MAIN
```


CHR\$

CHR\$ (ASCII-Code)

CHR\$: Typumwandlung: Byte -> 1 Stringzeichen

Funktion: Liefert einen String mit dem Zeichen zurück, welches dem **ASCII-Code** entspricht.

ASCII-Code ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und 255.

Funktionswert ist vom Typ STRING mit der Länge 1.

Programmbeispiel:

```

' -----
' Name: CHR$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  BYTE N                                ' BYTE-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR N = 41h TO 7Fh                      ' Schleife ueber Zeichen
    PRINT #1, CHR$(N);                    ' Zeichen ausgeben
  NEXT                                     ' naechstes Zeichen
END                                         ' Ende Task MAIN

```

Siehe auch: ASC, STR\$, VAL_NUM

CLEAR_FIFO

CLEAR_FIFO

CLEAR_FIFO Name

CLEAR_FIFO: Fifo löschen und in Anfangs-Zustand bringen

Instruktion: Löscht den Inhalt des FIFO-Puffers **Name**. (FIFO=First-In-First-Out).

Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```
' -----
' Name: CLEAR_FIFO.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER (512) OF WORD              ' FIFO-Buffer definieren
  WORD I,J                               ' Variablen definieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 1 TO 377                        ' 377 Werte in den
    PUT_FIFO BUFFER, I                   ' FIFO-Buffer schreiben
  NEXT                                   ' Ende der FOR-NEXT Schleife
  PRINT #1, "<1> Used FIFO =";LEN_FIFO (BUFFER) ' vor Loeschen
  CLEAR_FIFO BUFFER                       ' FIFO-Buffer loeschen
  PRINT #1, "After Clear =";LEN_FIFO (BUFFER) ' nach Loeschen
END                                       ' Ende Task MAIN
```

Siehe auch: GET_FIFO, PUT_FIFO, INTEGRAL_FIFO, READ_FIFO

CODE_1TON\$**CODE_1TON\$ (Code\$, Entry\$, EntryLen)****CODE_1TON\$ (Code\$, Flash_adr, LenFlash, EntryLen)**

CODE_1TON\$: Konvertiert in String: 1 Byte ==> N Bytes
(nur in Projektmodell 'PM_FULL')

Funktion: (Code-One-To-N) Liefert die zu den Indizes gehörenden Strings zurück.

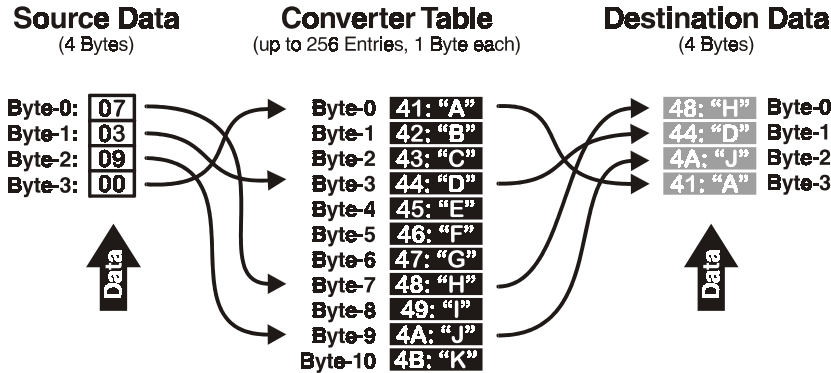
Code\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die umzusetzenden Bytes.
Entry\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu den Codes gehörenden String-Stücke.
Flash_adr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG, WORD oder BYTE und ist eine Adresse im Flash, wo die Tabelle mit den Einträgen beginnt.
LenFlash	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der gesamten Tabelle im Flash an.
Len	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der Einträge in der Entry\$-Tabelle an. Die Länge kann bis zu 16kBytes sein.
Funktionswert	ist vom Typ STRING und enthält die zu den Codes gehörenden Einträge.

CODE_1TON\$ verwendet die Bytes in Idx\$ als Indizes, um im Zielstring die Einträge aus der Tabelle entsprechend den Indizes zusammenzusetzen. Alle Einträge sind gleich lang. Da die Indizes Bytes sind, gibt es maximal 256 Einträge in einer Tabelle.

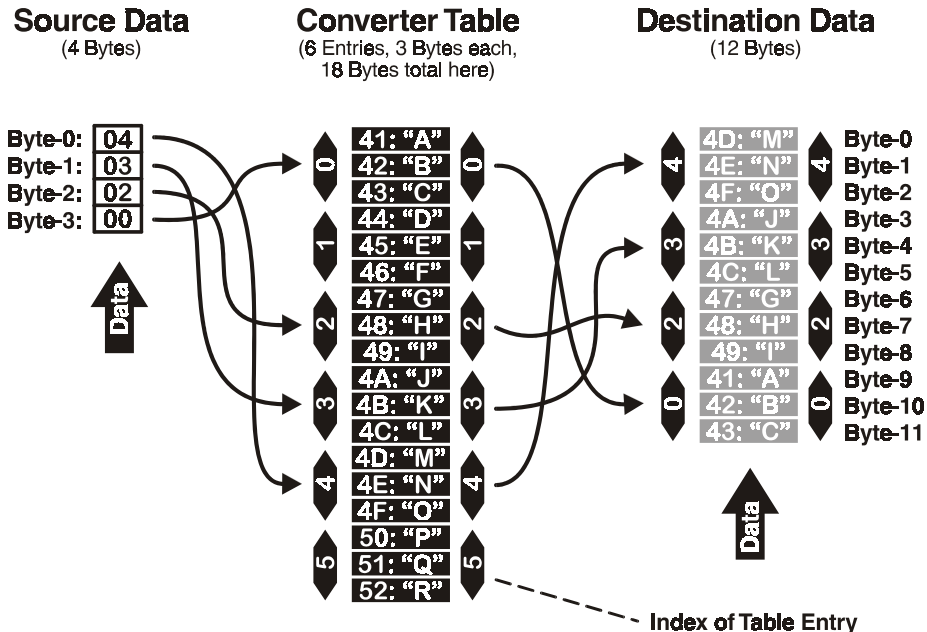
CODE_1TON\$

Das Beispielbild zeigt 10 gleichlange Einträge, aus denen durch Idx\$ mit 4 Codes der Zielstring zusammengebaut wird. Die Einträge könnten z.B. kurze Texte sein, aber auch Soundstücke von mehreren kByte Größe.

Idx\$ = "<7><3><9><0>“:



Eine abgewandelte Darstellung mit Texten statt Indizes. Idx\$ = "<3><2><1><6>“:



Anwendungen:

- Bytes in 2 Bytes wandeln, um einen fehlerkorrigierenden Code zu erhalten
- Zeichen-Generatoren aufbauen
- Sätze aus Indizes zusammenbauen
- Zahlen ausgeschrieben erzeugen
- gesprochene Zahlen-Sounds nach Ziffern zusammensetzen
- komprimierte Datenbestände erzeugen

Das Beispielprogramm baut aus den Codes in einem String einen neuen String auf, indem die Indizes im Indexstring abgearbeitet werden und die zugehörigen Codes aneinandergereiht werden.

3

CODE_1TON\$

Programmbeispiel:

```
-----
' Name: CODE_1TON$.TIG
' haelte die Tabelle in einem String
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING IDX$, CODE$, RESULT$           ' STRING-Variable deklarieren
  LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
                                          ' Eintraglaenge = 5
  CODE$ = "ZERO ONE TWO THREEFOUR FIVE SIX SEVENEIGHTNINE TEN "
  IDX$ = "05 07 08 02"%
                                          ' Indizes
  RESULT$ = CODE_1TON$ ( IDX$, CODE$, 5 ) ' baue neuen String
  PRINT #1, RESULT$                     ' zeige den neuen String
END                                     ' Ende Task MAIN
```

Das nächste Beispiel hat die Codes im Flash.

Programmbeispiel:

```
-----
' Name: CODE_1TON$1.TIG
' haelte die Tabelle im Flash
-----
user_var_strict

DATALABEL CODES, ENDCODES

TASK MAIN                                ' Beginn Task MAIN
CODES::
  DATA "ZERO ONE TWO THREEFOUR FIVE SIX SEVENEIGHTNINE TEN "
ENDCODES::

  STRING IDX$, RESULT$                   ' STRING-Variable deklarieren
  LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
                                          ' Eintraglaenge = 5

  IDX$ = "05 07 08 02"%
                                          ' Indizes: baue neuen String
  RESULT$ = CODE_1TON$ ( IDX$, CODES, ENDCODES-CODES, 5 )
  PRINT #1, RESULT$                     ' zeige den neuen String
END                                     ' Ende Task MAIN
```

CONT_TASK

CONT_TASK Name

CONT_TASK: Setze gestoppte Task wieder fort

Instruktion: Setzt die Task **Name** fort.

Die Ausführung dieser Task wird an der Stelle, an der die Task unterbrochen wurde, weitergeführt. Falls die Task bereits aktiv ist, bleibt diese Instruktion wirkungslos. Die Instruktion CONT_TASK wird in Verbindung mit der Instruktion STOP_TASK eingesetzt.

Programmbeispiel:

```

' -----
' Name: CONT_TASK.TIG
' -----
' Zaehlt die Variablen I (in Task MAIN) und J (in Task T1) hoch.
' Ist I zwischen 300 und 500, ist die Task T1 gestoppt und J
' wird in dieser Zeit nicht weiter hochgezählt.
' -----
LONG I, J                                ' Globale Var. deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
RUN_TASK T1                              ' TASK T1 starten
FOR I = 0 TO 1000                          ' Zaehlschleife fuer I
  IF I > 300 THEN                          ' Ist I > 300 ?
    STOP_TASK T1                          ' Bei I > 300 Task T1 stoppen
  ENDIF                                  ' Ende der IF-Abfrage
  IF I > 500 THEN                          ' Ist I > 500 ?
    CONT_TASK T1                          ' Bei I > 500 Task T1 starten
  ENDIF                                  ' Ende der IF-Abfrage
  PRINT #1, "<1>I ="; I                    ' I anzeigen
  PRINT #1, "J ="; J                      ' J anzeigen
  WAIT DURATION 100                       ' 100 ms warten
NEXT                                       ' naechster Wert
END                                         ' Ende Task MAIN

' -----
' TASK T1 zaehlt globale Variable J hoch
' -----
TASK T1                                  ' Beginn Task T1
FOR J = 0 TO 1000                          ' Zaehlschleife fuer J
  WAIT DURATION 100                       ' 100 ms warten
NEXT                                       ' naechster Wert
END                                         ' Ende Task T1

```

Siehe auch: STOP_TASK, RUN_TASK

CONVERT\$

CONVERT\$

CONVERT\$ (*String1*, *StrTabelle*)

CONVERT\$ (*String1*, *FlashTabelle*, *Länge*)

CONVERT\$: Zeichen-zu-Zeichen Konversion lt. eigener Tabelle

Funktion: Konvertiert alle Zeichen in **String1** nach der in **StrTabelle** enthaltenen Konversionstabelle. Die Zeichencodes des **String1** dienen als Index in Konversionstabelle **StrTabelle**. Liegt die Konversionstabelle im Flash, dann wird die Flash-Adresse und die Länge angegeben.

String1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält den zu konvertierenden String.
StrTabelle	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Konversionstabelle mit bis zu 256 Bytes. Zeichencodes, die keine Entsprechung in der Konversionstabelle haben, weil die Tabelle kürzer ist, werden nicht konvertiert.
FlashTabelle	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG und enthält die Adresse der Konversionstabelle im Flash.
Länge	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält die Länge der Konversionstabelle im Flash. Zeichencodes, die keine Entsprechung in der Konversionstabelle haben, weil die Tabelle kürzer ist, werden nicht konvertiert.
Funktionswert	ist vom Typ STRING und enthält den konvertierten String.

Die ASCII-Codes der Zeichen in String1 dienen als Index in die Konversionstabelle String2. CONVERT\$ konvertiert z.B. ein Zeichen in String1 mit dem Code '16' zu dem Code, der an der Position 16 in String2 steht.

Convert ist ein Byte-zu-Byte Konverter. CONVERT\$ wird eingesetzt:

- zur Zeichensatz-Umsetzung
- zur Maskierung bestimmter Bytes
- zu schnellen byteweisen Berechnungen
- etc.

CONVERT\$ arbeitet schnell und manipuliert mit einer einzigen Instruktion den gesamten String.

Tip

Die Verarbeitungsgeschwindigkeit dieser Funktion ist wesentlich höher, wenn die Zuweisungsvariable, die den Funktionswert aufnimmt, mit der Variablen String1 identisch ist: **A\$ = CONVERT\$ (A\$, B\$)**.

Programmbeispiel:

```

'-----
' Name: CONVERT$.TIG
' Demonstriert Konvertierung mit Tabelle im String und im Flash
'-----
DATALABEL FLASH_CONVERT_TAB

TASK MAIN                                ' Beginn Task MAIN
FLASH_CONVERT_TAB::
  DATA BYTE "&"                        ' Code 0 wird zu 1, etc
01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10&
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20&
21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30&
31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40&
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50&
51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60&
61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70&
71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 00"%

  STRING A$, CODE$(128)                ' STRING-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  CODE$ = "&"                          ' Codeumsetzungsstring zur
0000000000000000000000000000000000&    ' 00..1Fh | Invertierung von
.....&                                ' 20..3Fh | Gross- / Klein-
.abcdefghijkmnopqrstuvwxyz.....&        ' 40..5Fh | Schreibung
.ABCDEFGHIJKLMNOPQRSTUVWXYZ.....&      ' 60..7Fh | (256 Bytes)
.....&                                ' 80..9Fh
.....&                                ' A0..BFh
.....&                                ' C0..DFh
....."                                ' E0..FFh

  A$ = "Hello World"                    ' Original String-Text
  PRINT #1, "<l>"; A$                    ' Ausgabe Original-Text
  PRINT #1, CONVERT$ (A$,CODE$)         ' Ausgabe konvertiert
  PRINT #1, CONVERT$ (CONVERT$ (A$,CODE$),CODE$) ' 2 x konvertiert
'-----
  A$="Hello World 0123456"              ' und mit Tabelle im Flash
  PRINT #1, "<l>"; A$                    ' Original String-Text
  A$=CONVERT$ (A$,FLASH_CONVERT_TAB,65) ' Convert-Tabelle 65 Bytes
  PRINT #1, A$
  A$=CONVERT$ (A$,FLASH_CONVERT_TAB,128) ' Convert-Tabelle 128 Bytes
  PRINT #1, A$
END                                      ' Ende Task MAIN

```

Siehe auch: CONVERT7\$, Unregelmäßige Zahlenfolge auf Seite 296

CONVERT7\$

CONVERT7\$

CONVERT7\$ (*String1*, *String2*)

CONVERT\$ (*String1*, *FlashTabelle*, *Länge*)

CONVERT7\$: Dito, jedoch entfernt ev. gesetztes Bit-7

Funktion: Konvertiert alle Zeichen in **String1** nach der in **StrTabelle** enthaltenen Konversionstabelle. In String1 gesetzte Bit-7 werden ignoriert. Die Zeichencodes des **String1** dienen als Index in Konversionstabelle **StrTabelle**. Liegt die Konversionstabelle im Flash, dann wird die Flash-Adresse und die Länge angegeben.

String1 ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält den zu konvertierenden String.

StrTabelle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Konversionstabelle mit bis zu 128 Bytes (alle Codes zugelassen). Zeichencodes, die keine Entsprechung in der Konversionstabelle haben, weil die Tabelle kürzer ist, werden nicht konvertiert.

FlashTabelle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG und enthält die Adresse der Konversionstabelle im Flash.

Länge ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält die Länge der Konversionstabelle im Flash. Zeichencodes, die keine Entsprechung in der Konversionstabelle haben, weil die Tabelle kürzer ist, werden nicht konvertiert.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die ASCII-Codes der Zeichen in String1 dienen als Index in die Konversionstabelle String2. CONVERT7\$ konvertiert z.B. ein Zeichen in String1 mit dem Code '86H' zunächst zu 06H (Bit 7 zurückgesetzt), und dann zu dem Code, der an der Position 6 in String2 steht.

CONVERT7\$ arbeitet wie CONVERT\$, jedoch werden nur die untersten 7 Bit des Quellstrings ausgewertet. CONVERT7\$ wird in der Regel eingesetzt um Text-Konversionen vorzunehmen, deren Codierung in einem 7-Bit Code vorliegt. Ein eventuell gesetztes 8.Bit wird von CONVERT7\$ ignoriert.

Die Verarbeitungsgeschwindigkeit dieser Funktion ist wesentlich höher, wenn die Zuweisungsvariable, die den Funktionswert aufnimmt, mit der Variablen String1 identisch ist: **A\$ = CONVERT7\$ (A\$, B\$)**.

Tip

Programmbeispiel:

```

-----
' Name: CONVERT7$.TIG
-----

DATALABEL FLASH_CONVERT_TAB

TASK MAIN                                ' Beginn Task MAIN
FLASH_CONVERT_TAB::
    DATA BYTE "&"                        ' Code 0 wird zu 1, etc
01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10&
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20&
21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30&
31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40&
41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50&
51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60&
61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70&
71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 00"%

    STRING A$, CODE$(128)                ' STRING-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
    CODE$ = "&"
000000000000000000000000000000000000&    ' Codeumsetzungsstring zur
000000000000000000000000000000000000&    ' 00..1Fh | Tausche A mit Z,
000000000000000000000000000000000000&    ' 20..3Fh | B mit Y, C mit X
0ZYXWVUTSRQPONMLKJIHGFEDCBA000000&    ' 40..5Fh | etc.
0zyxwvutsrqponmlkjihgfedcba000000"    ' 60..7Fh

    A$ = "C1C2C3C4C5C6"%                ' Eingangs-String: Bit-7 = 1
PRINT #1, "<1>"; A$                      ' Ausgabe Original-Text
PRINT #1, CONVERT7$ (A$,CODE$)          ' Ausgabe konvertiert
' -----
A$="Hello World 0123456"                ' und mit Tabelle im Flash
PRINT #1, "<1>"; A$                      ' Original String-Text
A$=CONVERT$ (A$,FLASH_CONVERT_TAB,65)   ' Convert-Tabelle 65 Bytes
PRINT #1, A$
A$=CONVERT$ (A$,FLASH_CONVERT_TAB,128)   ' Convert-Tabelle 128 Bytes
PRINT #1, A$
END                                        ' Ende Task MAIN
    
```

3

Siehe auch: CONVERT\$

COS

COS

COS (rad)

COS: COSINUS, Real-Werte

Funktion: Liefert den Cosinus von **rad** als zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $-4.5 \cdot 10^{15}$ bis $4.5 \cdot 10^{15}$ und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von -1 bis 1. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
'-----  
' Name: COS.TIG  
'-----  
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren  
  
TASK MAIN                              ' Beginn Task MAIN  
    LONG DEG                            ' LONG-Variable deklarieren  
    REAL RAD                            ' REAL-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
    INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'    INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
    USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"  
    FOR DEG = 0 TO 360                  ' Winkel von 0 bis 360 Grad  
        RAD = DEG * PI_180              ' Winkel: Grad- zu Bogenmass  
        PRINT #1, "<1>COS ("; DEG; " Grad)" ' Ausgabe auf LC-Display  
        PRINT USING #1, "="; COS (RAD) ' Cosinuswert ausgeben  
        WAIT DURATION 1000              ' 1 Sek warten  
    NEXT                               ' naechster Winkelwert  
END                                    ' Ende Task MAIN
```

Siehe auch: SIN, TAN, ASIN, ACOS, ATAN, SINH

COSH

COSH (rad)

COSH: COSINUS hyperbolicus, Real-Werte

Funktion: Liefert den Cosinushyperbolicus von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -709 bis 709 und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
' -----
' Name: COSH.TIG
' -----
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                                ' Beginn Task MAIN
  LONG DEG                              ' LONG-Variable deklarieren
  REAL RAD                              ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                      ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                    ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>COSH ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; COSH (RAD)      ' Cosinus hypercolicus ausgeben
  WAIT DURATION 1000                   ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                  ' Ende Task MAIN
```

Siehe auch: COS, TAN, ASIN, ACOS, ATAN, SINH

COT

COT

COT (rad)

COT: COTANGENS, Real-Werte

Funktion: Liefert den Cotangens von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $-4.5 \cdot 10^{15}$ bis $4.5 \cdot 10^{15}$ und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
'-----
' Name: COT.TIG
'-----
#define PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                             ' Beginn Task MAIN
  LONG DEG                           ' LONG-Variable deklarieren
  REAL RAD                           ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                    ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                  ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>COT ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; COT (RAD)     ' Cotangenswert ausgeben
  WAIT DURATION 1000                 ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                  ' Ende Task MAIN
```

Siehe auch: SIN, COS, TAN, ASIN, ACOS, ATAN, SINH

COTH

COTH (rad)

COTH: COTANGENS hyperbolicus, Real-Werte

Funktion: Liefert den Cotangenshyperbolicus von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -709 bis 709 und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
' -----
' Name: COTH.TIG
' -----
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                             ' Beginn Task MAIN
  LONG DEG                            ' LONG-Variable deklarieren
  REAL RAD                            ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                    ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                  ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>COTH ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; COTH (RAD)    ' Cosinus hypercolicus ausgeben
  WAIT DURATION 1000                 ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                   ' Ende Task MAIN
```

Siehe auch: COS, TAN, ASIN, ACOS, ATAN, SINH

CUT_AND_PASTE

CUT_AND_PASTE

CUT_AND_PASTE (*Wert*, *Resetwert*)

Funktion: Schneidet die Bytes aus **Wert** heraus und ersetzt sie mit dem angegebenen **Resetwert**.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL und ist der Wert, aus dem die Bytes herausgeschnitten werden.

Resetwert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt fest, auf welchen Wert die Argumentvariable nach dem Ausschneiden gesetzt wird.

Funktionswert ist vom Typ LONG bzw. REAL und enthält den ausgeschnittenen Wert. Eine Typwandlung bei der Zuweisung findet automatisch statt.

CUT_AND_PASTE liefert den Wert der Argumentvariablen im Funktionswert zurück und setzt die Argumentvariable auf den Resetwert.

Wenn im Multitaskingsystem auf eine Variable von 2 Tasks quasi gleichzeitig zugegriffen wird, um sie zu verändern, ist ein versetzter Zugriff ‚Lesen‘ und ‚Schreiben‘ von beiden Tasks unerwünscht. Der Vorgang muß vielmehr in einem Befehl ablaufen, was durch CUT_AND_PASTE ermöglicht wird.

Programmbeispiel:

```

'-----
' Name: CUT_AND_PASTE.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  REAL R, RC

' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  L = 12345678h                          ' Original-LONG-Zahl
  C = CUT_AND_PASTE ( L, 5555h )          ' schneide und setze auf 5555h
  USING "UH<8><8>  0 0 0 4 4"
  PRINT_USING #1, "C=";C                  ' Funktionswert zeigen
  PRINT_USING #1, "L=";L                  ' nach dem Schneiden zeigen

  R = 12.34                              ' Original-REAL-Zahl
  RC = CUT_AND_PASTE ( R, 100.0 )         ' schneide aus und setze R auf 100.0
  USING "NF<3><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0"
  PRINT_USING #1, "R=";R
  PRINT_USING #1, "RC=";RC
END

```

3

CUT_AND_PASTE\$

CUT_AND_PASTE\$

CUT_AND_PASTE\$ (*String*, *Position*, *Anzahl*)

CUT_AND_PASTE: Schneidet Wert heraus und kopiert
(nur in Projektmodell 'PM_FULL')

Funktion: Schneidet die **Anzahl** Bytes ab der **Position** aus dem **String** heraus. Der String wird dabei kürzer.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist der String, aus dem die Bytes herausgeschnitten werden.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der auszuscheidenden Bytes fest.
Funktionswert	ist vom Typ STRING und enthält die ausgeschnittenen Bytes ab der Position 0.

CUT_AND_PASTE\$ kopiert die angegebene Anzahl Bytes in String von der angegebenen Byteposition in den Funktionswert ab der niederwertigsten Byteposition. Es werden die Bytes im Quellstring hinter der geschnittenen Stelle an die Anfangsposition der Schnittstelle geschoben, d.h. die Lücke geschlossen und der String entsprechend kürzer.

Programmbeispiel:

```

'-----
' Name: CUT_AND_PASTE$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

A$ = "The quick brown"
PRINT #1, "A$=";A$                        ' A$ vor CUT_AND_PASTE
B$ = CUT_AND_PASTE$ ( A$, 4, 6 )
PRINT #1, "B$=";B$                        ' Ergebnis in B$
PRINT #1, "A$=";A$                        ' A$ nach CUT_AND_PASTE

B$ = CUT_AND_PASTE$ ( A$, 7, 10 )' mehr schneiden als da ist
PRINT #1, "B$=";B$                        ' Ergebnis in B$
END

```

3

Siehe auch: MID\$, LEFT\$, RIGHT\$

CUT_BITS

CUT_BITS

CUT_BITS (*Wert*, *Position*, *Anzahl*)

CUT_BITS: Schneidet Bit-Gruppe aus numerischem Wert aus
(nur in Projektmodell 'PM_FULL')

Funktion: Schneidet die **Anzahl** Bits ab der **Position** aus **Wert** heraus.

3

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält den Wert, aus dem die Bits herausgeschnitten werden.

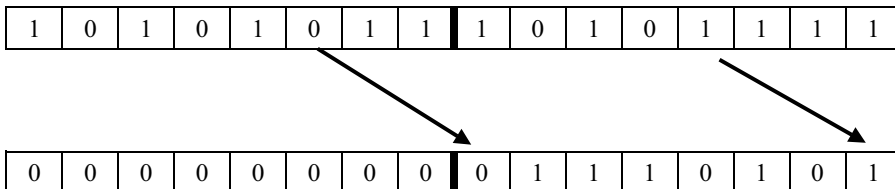
Position ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition gezählt von low Bit aus an.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der zu lesen- den Bytes fest.

Funktionswert ist vom Typ LONG und enthält die ausgeschnittenen Bits ab der Position 0 (low Bit). Eine Typwandlung bei der Zuweisung findet automatisch statt.

CUT_BITS kopiert die angegebene Anzahl Bits in Wert von der angegebenen Bitposition in den Funktionswert ab der niederwertigsten Bitposition. Nicht kopierte Bits werden in Funktionswert auf 0 gesetzt.

Beispiel mit Wert = 0ABAFh vom Typ WORD: **CUT_BITS (Wert, 3, 8)**



Programmbeispiel:

```

'-----
' Name: CUT_BITS.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG A, B
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A = 57H                                ' das ist
  PRINT #1, "A:";                        ' zeige A$
  CALL DISP_BIN ( A, 16 )                ' binaer
  B = CUT_BITS ( A, 3, 7 )                ' schneide 7 Bits ab Pos. 3
  PRINT #1, "CUT_BITS(A,3,7)"
  PRINT #1, "B:";
  CALL DISP_BIN ( B, 16 )                ' zeige B
END                                        ' Ende Task MAIN

SUB DISP_BIN ( VALUE, CNT )              ' CNT Bits binaer aus VALUE
  L = VALUE
  STRING B$
  B$=""
  FOR I = 1 TO CNT
    IF BIT ( L, 0) = 1 THEN               ' B$ zuruecksetzen
      B$ = "1" + B$                      ' bis gewuenschte Bitzahl
    ELSE                                  ' Falls Division mit Rest,
      B$ = "0" + B$                      ' binaere 1 voranstellen.
    ENDIF                                ' Falls kein Rest,
                                          ' binaere 0 voranstellen.
    L = L SHR 1                           ' L eins nach rechts (neues Bit)
  NEXT
  PRINT #1, B$
END                                        ' Ende Unterprogramm

```

3

Siehe auch: BIT_MIRR, AND\$, OR\$, Operatoren BITAND,BIT OR.

DEBOUNCE

DEBOUNCE

DEBOUNCE (*Make, Break, Neutral*)

DEBOUNCE (*Code*)

DEBOUNCE: Entprellt:Tasten, Kontakte, Touch-Screen, Lichtschranken, etc.
(nur in Projektmodell 'PM_FULL')

3

Funktion: ermöglicht auf einfache Weise das Entprellen von Tasten, Schaltern oder auch Touch-Screen-Eingaben.

Make	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, wieviele lückenlos aufeinanderfolgende gleiche Codes einen neuen gültigen Code erzeugen. Wertebereich: 1...8
Break	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, wieviele lückenlos aufeinanderfolgende abweichende Codes erforderlich sind, um einen Code als üngültig zu markieren. Wertebereich: 1...8
Neutral	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welcher Code vorliegt, wenn keine Taste gedrückt ist, keine Eingabe erfolgt.
Code	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der zu entprellende Code.
Funktionswert	ist vom Typ LONG und meldet im Falle der drei Parameter Make, Break, Neutral zurück, ob ein Fehler aufgetreten ist: 0: ok 1: Parameterfehler liefert im Falle von Code als Parameter den gültigen Code dann zurück, wenn die Entprellfunktion diesen Code als gültig erkannt hat, sonst wird der Neutral-Code zurückgeliefert. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Setup von DEBOUNCE mit 3 Parametern:

DEBOUNCE wird zunächst einmal mit den drei Parametern Make, Break und Neutral aufgerufen, um die gewünschten Einstellungen vorzunehmen:

Neutral: Ein bestimmter Code liegt vor, wenn keine Eingabe stattfindet, bei der Tastatur der Code, der gescannt wird, wenn keine Taste gedrückt ist, bei einem Touch-Panel der Code, der entsteht wenn keine Berührung auf der Fläche stattfindet.

Make: Eine gewisse Anzahl gleicher Codes sind erforderlich, um als eine gültige Eingabe zu gelten. Die Codes müssen lückenlos hintereinanderfolgen, so oft, wie im Parameter Make vorgegeben.

Break: Um die einmal gültige Eingabe wieder abubrechen, sind wiederum eine Anzahl ungültige Codes erforderlich. Diese Anzahl kann zwischen 1 und 8 liegen. Es müssen entsprechend viele ungültige Codes lückenlos aufeinanderfolgen, wie in Break angegeben.

Verwenden von DEBOUNCE mit 1 Parameter:

Die spätere Verwendung der Funktion erfordert nur einen Parameter: den Code der entprellt werden soll. Dieser Aufruf von DEBOUNCE steht in der Scan-Schleife der Eingabe und übergibt den Code zur Überprüfung. Als Ergebnis wird entweder der Neutral-Code zurückgeliefert, oder wenn die Entpellung erfolgreich war, der übergebene Code.

DEBOUNCE

Programmbeispiel:

3

```
'-----
' Name: DEBOUNCE.TIG
' Entprellen von Tasten, Kontakten ...
'-----

TASK MAIN
    LONG OK_FLG, WERT

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

    OK_FLG = DEBOUNCE ( 4, 2, 0 )      ' MAKE, BREAK, NEUTRAL
    IF OK_FLG <> 0 THEN
        PRINT #1, "Parameter Error"
        GOTO EXIT
    ENDIF

' die 4. Erscheinung von 2 ergibt einen gueltigen Code
PRINT #1, DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2)
PRINT #1, DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(3)
PRINT #1, DEBOUNCE(3);DEBOUNCE(2);DEBOUNCE(3);DEBOUNCE(3)
PRINT #1, DEBOUNCE(3);DEBOUNCE(3);DEBOUNCE(3);DEBOUNCE(3)
WAIT_DURATION 3000

PRINT #1, "<1>";                        ' loesche Bildschirm
OK_FLG = DEBOUNCE ( 1, 2, 0 )          ' MAKE, BREAK, NEUTRAL
IF OK_FLG <> 0 THEN
    PRINT #1, "Parameter Error"
    GOTO EXIT
ENDIF

' erst in der letzten Zeile bricht die 3 den Code
PRINT #1, DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2)
PRINT #1, DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(3)
PRINT #1, DEBOUNCE(2);DEBOUNCE(3);DEBOUNCE(2);DEBOUNCE(3)
PRINT #1, DEBOUNCE(3);DEBOUNCE(3);DEBOUNCE(3);DEBOUNCE(3)
WAIT_DURATION 3000

PRINT #1, "<1>";                        ' loesche Bildschirm
OK_FLG = DEBOUNCE ( 1, 7, 0 )          ' MAKE, BREAK, NEUTRAL
IF OK_FLG <> 0 THEN
    PRINT #1, "Parameter Error"
    GOTO EXIT
ENDIF

' nach 7 ungueltigen Codes wird die 2 durch eine 4 ersetzt
PRINT #1, DEBOUNCE(2);DEBOUNCE(2);DEBOUNCE(4);DEBOUNCE(3)
PRINT #1, DEBOUNCE(4);DEBOUNCE(3);DEBOUNCE(4);DEBOUNCE(3)
PRINT #1, DEBOUNCE(4);DEBOUNCE(5);DEBOUNCE(4);DEBOUNCE(3)
PRINT #1, DEBOUNCE(1);DEBOUNCE(2);DEBOUNCE(3);DEBOUNCE(3)

EXIT:
END
```


DECODE

DECODE (*Dst\$, DstSiz, Src\$, SrcSiz, Conv\$, ErrFlag\$*)

DECODE (*Dst\$, DstSiz, Src\$, SrcSiz, Conv, ErrFlag*)

DECODE: Decodiert lt. Table, z.B: fehlerkorrigierende Codes
(nur in Projektmodell 'PM_FULL')

Funktion: Ordnet dem Zielstring Dst\$ Byte(s) zu, die durch die Bitgruppen in Src\$ indiziert werden. Zählt gleichzeitig Eigenschaften dieser Zuordnungen, die in der Fehlerflagtabelle in ErrFlag\$ bzw. ErrFlag hinterlegt sind. Anwendung: z.B. Decodierung fehlerkorrigierender Codes.

3

Dst\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und erhält die decodierten Bytes. Die durch den Decodiervorgang entstehenden Bytes werden an vorhandene Bytes in Dst\$ angehängt, d.h. Dst\$ wächst.
DstSiz	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt Anzahl Bytes an, die je Decodiervorgang aus Conv\$ bzw. Conv entnommen werden und an Dst\$ angehängt werden. Entsprechend aufgebaut ist der Konverterstring bzw. die Daten im Flash.
Src\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu decodierenden Daten.
SrcSiz	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Bits an, die im Quellstring Verwendung finden. Im Quellstring Src\$ liegen die Daten in ganzen Bytes vor. SrcSiz signalisiert, wieviele Bits davon jeweils verwendet werden zur Bildung des Konverter-Indexes. String: 1...14 Bits Flash: 1...22 Bits
Conv\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Konversionstabelle.
Conv	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält die Flash-Adresse der Konversionstabelle.
ErrFlag\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält eine Tabelle, die Stellen der Tabelle in Conv\$ markiert. Entsprechend dem Wert in ErrFlag\$ befindet sich an derselben Stelle in Conv\$:

DECODE

- 0: 0: ein gültiger Code.
- 1: ein korrigierter Code, der nun auch einen gültigen Code ergibt.
- 2: ein nicht korrigierbarer Code, der nunzwangsläufig einen ungültigen Code erzeugt.

ErrFlag

ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält die Adresse einer Tabelle, die Stellen der Tabelle in Conv markiert. Entsprechend dem Wert in ErrFlag befindet sich an derselben Stelle in Conv:

- 0: 0: ein gültiger Code.
- 1: ein korrigierter Code, der nun auch einen gültigen Code ergibt.
- 2: ein nicht korrigierbarer Code, der nunzwangsläufig einen ungültigen Code erzeugt.

Funktionswert

ist vom Typ LONG und enthält unterteilt in 2 WORDs folgende Information:

wenn positiv:

high-WORD: Anzahl der nicht korrigierten Fehler.

low-WORD: Anzahl der korrigierten Fehler.

wenn negativ:

- 1: Zielstring Dst\$ ist voll.
- 2: Angabe SrcSiz ungültig (1...14).
- 3: Angabe DstSiz ungültig (1...127).
- 4: ErrFlag-Bereich zu klein.
- 5: Konverterbereich zu klein.
- 6: freie Restlänge in Dst\$ zu klein.
- 7: Konverterbereich ist zu groß (größer als 16kBytes)

Hinweis: Beide Parameter Conv\$ und ErrFlag\$ sind entweder Strings oder beide Conv und ErrFlag liegen im Flash. Gemischte Parameter String und Flash sind nicht möglich.

Im Quellstring **Src\$** liegen die die Codierte Daten zusammen mit den Kontrollbits zur Fehlererkennung und Fehlerkorrektur vor. Um den Dekodiervorgang zu beschleunigen, wird eine Tabelle verwendet, die die Dekodierten Daten enthält (**Conv** bzw. **Conv\$**). Die Codes aus dem Quellstring dienen dabei als Index in die Dekodiertabelle und zeigen direkt auf den dekodierten Code.

Liegt kein Fehler vor, dann zeigt der Quellcode auf einen korrekten Zielcode.

Liegt ein korrigierbarer Fehler vor, dann zeigt der fehlerhafte Quellcode auf eine Stelle in der Konvertertabelle, wo sich wiederum der richtige Code befindet. Die richtigen Codes kommen also mehrfach in der Kodiertabelle vor.

Liegt ein nicht korrigierbarer Fehler vor, dann zeigt der Quellcode auf einen Zielco-

de, der nicht richtig sein kann. Hier kann der Programmierer irgendeinen Code ablegen.

Die Angabe der verwendeten Bits pro Datensatz in Src\$ impliziert zusammen mit DstSiz die Größe der Tabelle Conv\$ und damit auch ErrFlag\$ bzw. den entsprechenden Tabellen im Flash. Während im Flash die Länge einfach vorausgesetzt wird, führt ein zu kleiner String zu einer Fehlermeldung wegen nicht passender Tabellen. Beispiel: SrcSiz ist 7 Bit und DstSiz 2 Byte, dann enthält die Tabelle Conv\$ bzw. Conv 128 mal 2 Bytes, also 256 Bytes. ErrFlag\$ enthält genausoviele Flags wie Datensätze in Dst\$ vorhanden sind, also 128 bytes.

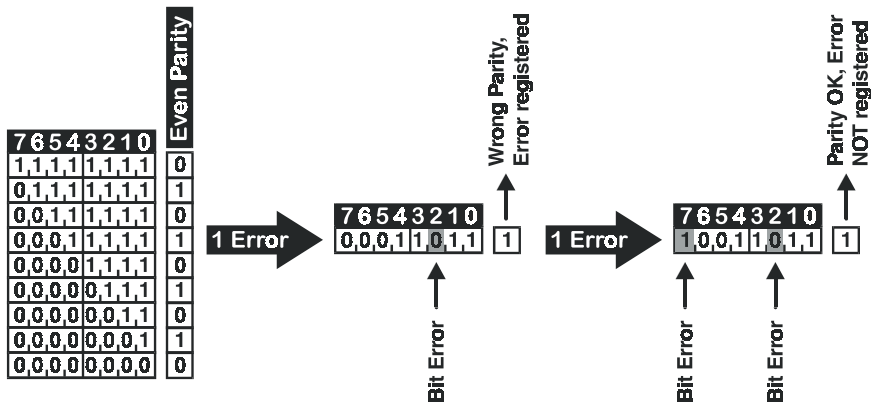
Falsche Tabellengrößen führen zur Rückgabe eines Fehlercodes und der Decodiervorgang wird nicht ausgeführt.

Um nun bei fehlerkorrigierenden Codes die richtigen, die korrigierten und die unkorrigierbaren Codes zu unterscheiden, wird gleichzeitig eine Fehlermarkierungstabelle indiziert (**ErrFlag** bzw. **ErrFlag\$**). In dieser Tabelle liegen Flags, die erkennen lassen, ob mit der Indizierung in die Konversionstabelle ein richtiger, ein korrigierter oder ein nicht-korrigierbarer Code entnommen wurde.

Fehlerkorrigierende Codes

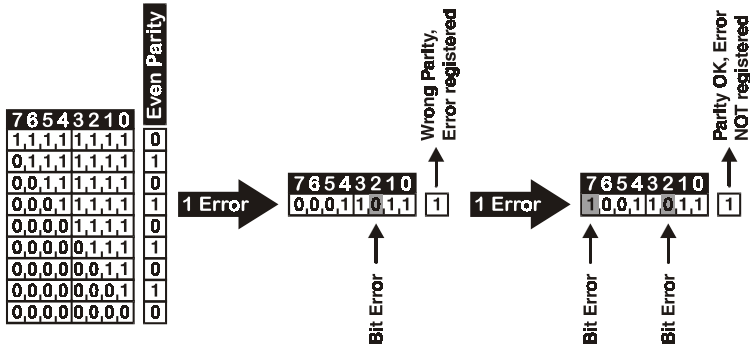
DECODE dient hauptsächlich der Umsetzung von Codierungen, bei denen das Quellformat aus mehr Bits besteht als das Zielformat. Dies können typischerweise fehlererkennende und/oder fehlerkorrigierende Codes sein. Das Prinzip solcher Codierungen basiert darauf, zusätzliche Informationen in eine Bitfolge einzubauen, die zur Erkennung von Fehlern und deren Behebung dienen.

Eine der bekanntesten Methoden zur Fehlererkennung auf Datenwortebene ist das Parity-Bit, das jedem Datenwort angefügt wird. Das Parity-Bit ergänzt die Bits eines Datenwortes so, daß hierdurch eine gerade Anzahl von „1“ Bits (EVEN Parity) oder eine ungerade Anzahl von „1“ Bits (ODD Parity) im Datenwort entsteht. Diese Methode erkennt 1 Bitfehler im Wort zuverlässig, 2 Bitfehler werden jedoch nicht mehr erkannt. Korrekturen sind nicht möglich.

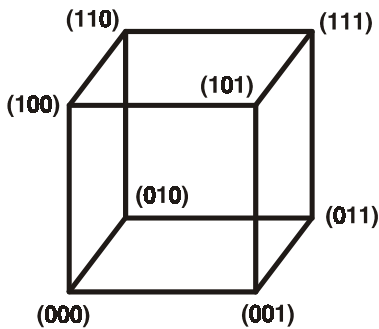


Diese Eigenschaft weist auf eine grundsätzliche Limitierung fehlererkennender und fehlerkorrigierender Codes hin: die Reichweite von Fehlererkennung und Fehlerkorrektur sind begrenzt. Treten mehr Fehler auf als erwartet, kann keine zuverlässiger Erkennung oder gar Korrektur mehr erfolgen. Ein wichtiges Kriterium für die Wahl einer Fehlererkennungs- und Korrekturmethode sind daher Kenntnisse über die Häufigkeit und Art möglicher Fehler. Während in Speicherchips z.B. einzelne Bits defekt sein können, werden in einer Funkübertragung durch Störungen eventuell ganze Bitgruppen gestört.

Um einem Code die Fähigkeit der Fehlerkorrektur zu geben, muß das fehlerhafte Codewort eindeutig dem ursprünglichen Codewort zugeordnet werden können. Folgende Abbildung zeigt ein Codewort von 3 Bit, das zur Übertragung einer 1-Bit Information verwendet wird.

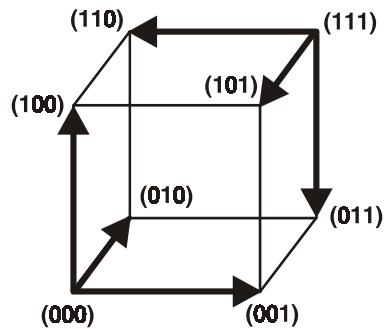


3



2 Messages transmittable:

"0" coded as (000)
 "1" coded as (111)



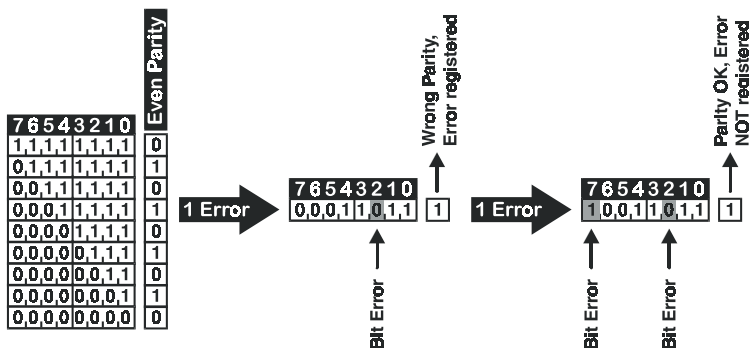
1 Bit Error correctable:

(000) → (100), (010) or (001)
 (111) → (011), (101) or (110)

DECODE

Die Veränderung eines Bits im Codewort kann stets erkannt und korrigiert werden. 2 Fehler werden erkannt, nicht jedoch korrigiert. Der kleinste Abstand zweier beliebiger gültiger Codes zueinander wird als Hamming-Distanz bezeichnet. Je größer dieser Abstand gewählt wird, desto mehr Fehler werden erkannt und korrigiert. Gleichzeitig steigen der Codierungsaufwand und das Datenvolumen, was in realen System unmittelbar mit den entstehenden Kosten verbunden ist.

Hier ein Beispiel für einen 7-Bit Code, der 4 Nutzbits transportiert und **einen** Fehler korrigieren kann. Den 4 Nutzbits werden 3 Kontrollbits hinzugefügt, so daß Worte von 7-Bit Länge entstehen. Die Wahl hier fiel auf 3 Kontroll-Bits, da man mit 3 Bits 8 Zustände darstellen kann: 7 Fehlerzustände sowie einen Zustand für Fehlerfreiheit. Tatsächlich werden die 3 Kontroll-Bits als Parity-Bits gebildet aus verschiedenen Gruppen von jeweils 4 Bits des 7-Bit Wortes. Bei einzelnen Bitfehlern kann festgestellt werden, ob das Datenbit oder das Parity-Bit falsch ist (E):



Es ist leicht ersichtlich, daß 1 Bit-Fehler erkannt und richtig korrigiert werden kann. 2 Fehler hingegen führen bereits zu einem falschen Ergebnis.

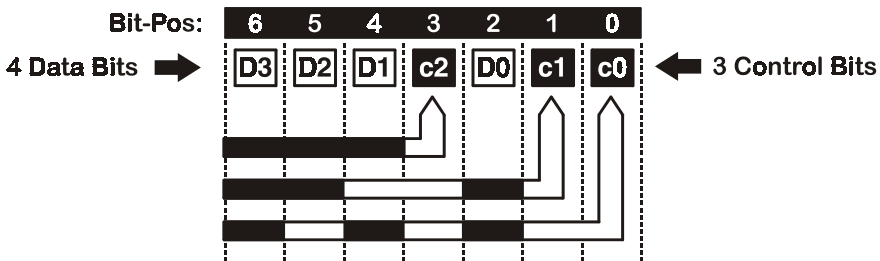
Die Decodierung des oben genannten Codes kann analytisch, entsprechend dem Code-Aufbau erfolgen. Dazu extrahiert man die Kontroll-Bits, prüft ob es einen Fehler gibt, korrigiert ggf. das Datenwort und registriert 1 Fehler + 1 Korrektur für das entsprechende Datenwort und setzt die Datenbits wieder zu einem vollständigen Nibble zusammen. Dies kostet natürlich CPU Zeit, um so mehr, je komplexer die Codierung aufgebaut ist. Will man solche Daten in Echtzeit decodieren, oder zumindest mit kurzen Umsetzungszeiten, bietet sich ein anderes Verfahren an:

- Der Code-Konverter

DECODE\$ ist ein Code-Konverter der speziell für solche Aufgaben ausgelegt ist. Der eigentliche Decodier-Vorgang wird über eine Konverter-Tabelle erledigt. Das führt zu kurzen Konversionszeiten, belegt jedoch Speicherplatz entsprechend der Adressraum-Größe. Für einen 7-Bit-Code wird eine Tabelle mit 128 Byte benötigt, für einen 10-Bit-Code bereits 1024 Byte.

Zusätzlich zur Code-Umsetzung, gibt DECODE\$ Auskunft über die Anzahl von Fehlern und Korrekturen. Dies sind wertvolle Informationen für das Anwendungsprogramm, um die Qualität des Datenübertragungskanals zu beurteilen. Diese Fehler-Informationen werden intern in der Error-Flag-Tabelle aufgezeichnet.

Das nachfolgende Beispiel zeigt alle Schritte einer möglichen Codierung und späteren Decodierung mit Fehlerkorrektur. Es wird der 4/7-Code verwendet wie oben beschrieben. Zusätzlich wird der 7-Bit Datenstrom lückenlos in 8-Bit Bytes gepackt, den Platz im Übertragungskanal optimal auszunutzen:



Each control bit is a **PARITY BIT** for a subword of 3 DATA BITS as shown above.

Any single bit error in a **DATA BIT** or a **CONTROL BIT** is detected and can be corrected as shown below:

c2	c1	c0	
OK	OK	OK	⇒ All 7 bits OK
OK	OK	E	⇒ Bit-0 Error ⇒ correctable (ignore)
OK	E	OK	⇒ Bit-1 Error ⇒ correctable (ignore)
OK	E	E	⇒ Bit-2 Error ⇒ correctable
E	OK	OK	⇒ Bit-3 Error ⇒ correctable (ignore)
E	OK	E	⇒ Bit-4 Error ⇒ correctable
E	E	OK	⇒ Bit-5 Error ⇒ correctable
E	E	E	⇒ Bit-6 Error ⇒ correctable

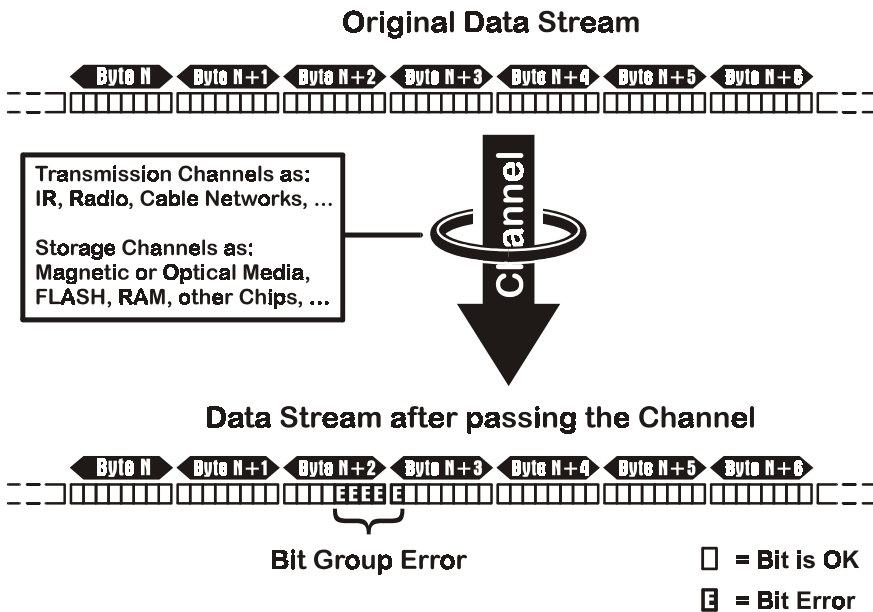
OK = This control bit is a correct **PARITY-BIT**

E = This control bit is **NOT** a correct **PARITY-BIT**

DECODE

Besteht die Wahrscheinlichkeit, daß bei Bit-Fehlern ganze Gruppen von Bits verändert werden, kann eine 4/7 Codierung wie oben beschrieben, nicht mehr zu fehlerfreien Ergebnissen führen. Die im Beispiel gewählte Codierung besitzt eine Korrekturfähigkeit von maximal 1 Fehler pro 7 übertragenen Bits. 2 und mehr Fehler pro 7 übertragenen Bits können nicht mehr behandelt werden. Auch wenn wir uns für einen Code mit stärkerer Korrekturfähigkeit entscheiden, kommen wir bei 3, 4 oder mehr aufeinander folgenden fehlerhaften Bits schnell an eine Grenze, die der Code selbst nicht mehr korrigieren kann.

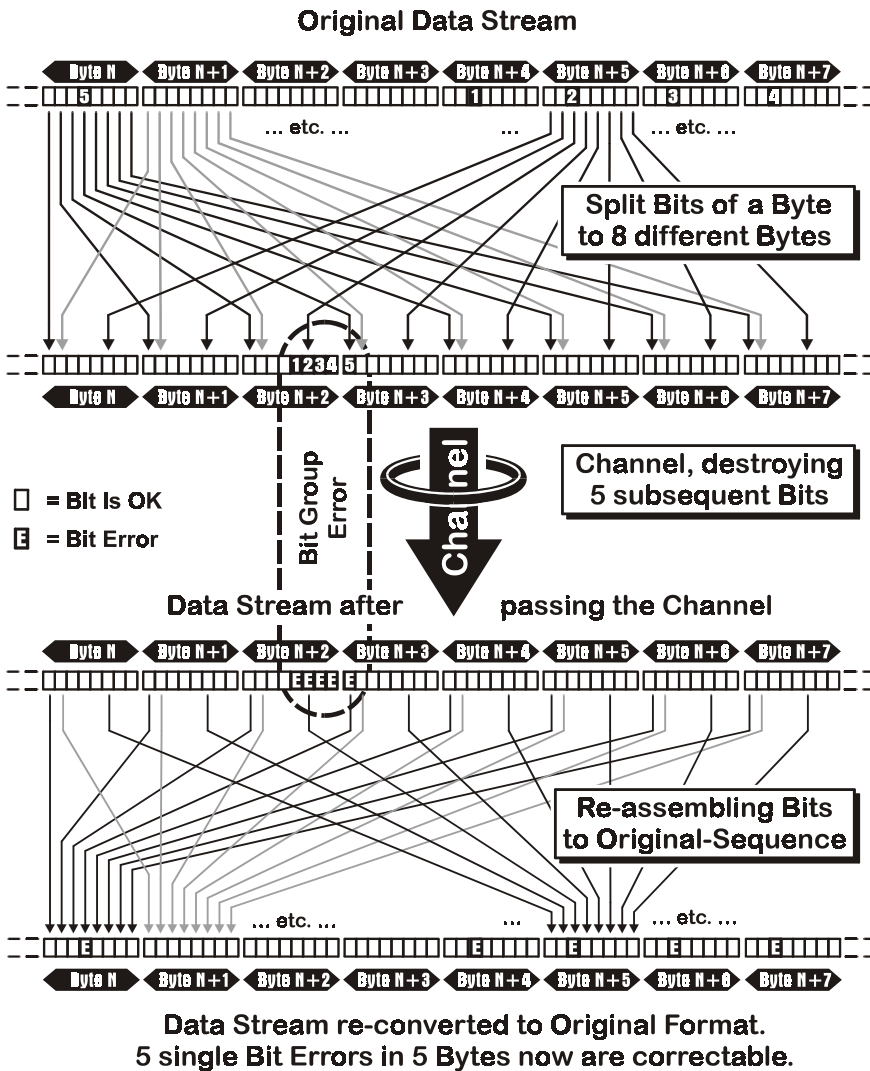
3



Error not correctable with 4/7 Code as described

Eine Methode um dennoch zu guten Ergebnissen zu gelangen besteht darin, die Eigenschaft der Nachbarschaft aufzuheben. Benachbarte Bits werden so verteilt, daß sie im Coderaum relativ weit auseinander liegen. Tritt dann im Coderaum ein Bit-Gruppen Fehler auf, der mehrere benachbarte Bits betrifft, so werden im Originalraum weit auseinander liegende Bits betroffen, die jeweils vom 4/7 Code erkannt und korrigiert werden können.

Das Prinzip dieser Methode zeigt die nachfolgende Abbildung:



Für das gezielte Verteilen einzelner Bits nach eigenen Vorgaben, gibt es in Tiger-BASIC die Funktionen BIT_MIX\$ und BIT_DEMIX\$. Hiermit ist die freie Verwürfelung von Bits in einem Datenstrom möglich, wobei BIT_DEMIX\$ invers zu BIT_MIX\$ arbeitet.

DECODE

Das erste Programmbeispiel zeigt die grundsätzliche Arbeitsweise von DECODE. Es läßt erkennen, wie die Tabellen in Abhängigkeit der Parameter verwendet werden, ohne einen konkreten Zweck zu verfolgen.

Im ersten Fall werden aus den Bitgruppen mit 3 Bits ganze Bytes (Zielgröße 1). Den beiden ungültigen Codes (eine 2 in der ErrFlag\$-Tabelle) werden 78h als Code zugewiesen (78h=ASCII ,x').

Im zweiten Fall liegen die Tabellen im Flash und es werden den Bitgruppen mit 3 Bits jeweils 6 Bytes zugewiesen (Zielgröße 6).

Programmbeispiel:

3

```
'-----
' Name: DECODE1.TIG
' Demonstriert die Arbeitsweise von DECODE
' erst mit Tabellen in Strings, dann Tabellen im Flash
'-----
user var strict                ' unbedingte Var.deklaration

DATALABEL DECO_TAB, DECO_ERR   ' Tabellen im Flash

TASK MAIN                      ' Beginn Task MAIN
DECO_TAB::                     ' 8 Eintraege a 6 Bytes
0   DATA BYTE "Hallo "       ' 0
1   DATA BYTE "111111"       ' 1
2   DATA BYTE ":2222:"       ' 2
3   DATA BYTE ".3333."       ' 3
4   DATA BYTE "-4444-"       ' 4
5   DATA BYTE "+5555+"       ' 5
6   DATA BYTE "=6666="       ' 6
7   DATA BYTE "-7777-"       ' 7

DECO_ERR::                     ' 8 Flags
  DATA BYTE 0,1,2,2,2,0,0

  LONG OK_ERR                  ' ok-error-flag
  STRING SOURCE$, DEST$       ' STRING-Variable deklarieren
  STRING CONV$, ERR_FLAGS$

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

CONV$    = "41 42 43 44 78 78 47 48"%
ERR_FLAGS$ = "00 00 00 00 02 02 00 00"%
SOURCE$   = "00 01 02 03 04 05 06 07"%

DEST$     = ""
OK_ERR = DECODE ( &
  DEST$, 1, &                ' Zielstring, -groesse in Bytes
```

```

SOURCE$, 3, &                                ' Quelldaten, Quellgr.: 1...14 Bits
CONV$, &                                       ' Converter-String
ERR_FLAGS$ )                                  ' Error Flags String

USING "UH<8><8> 0.0.0.4.4"
PRINT USING #1, "<1Bh>A<0><0><0F0h> Err: ";OK_ERR
PRINT #1, "<1Bh>A<0><1><0F0h>DEST$:";DEST$
WAIT_DURATION 3000
'-----
' Nun die Flashtabellen mit 8 Einträgen (3 Quellbits) a 6 Bytes

DEST$="ABC==>"                                ' siehe: es wird angehaengt
SOURCE$= "00 01 03 05 05 00" %

OK_ERR = DECODE ( &
    DEST$, 6, &                                ' Zielstring, diesmal 6 Bytes
    SOURCE$, 3, &                                ' Quelldaten, 3 gueltige Bits
    DECO_TAB, &                                  ' Converter im Flash
    DECO_ERR )                                  ' Error Flags im Flash

USING "UH<8><8> 0.0.0.4.4"
PRINT USING #1, "<1><1Bh>A<0><0><0F0h> Err: ";OK_ERR
PRINT #1, "<1Bh>A<0><1><0F0h>DEST$:";DEST$
END

```

Das zweite Programmbeispiel codiert zunächst einen String in fehlerkorrigierbare Codes. Es können Einzelbitfehler korrigiert werden. Dazu wird zunächst der String in seine Nibbles zerlegt, wobei doppelt so viele Bytes entstehen. Dann werden über eine Tabelle die Nibbles in solche Bytes konvertiert, die zusätzliche Fehlerkontrollbits enthalten. Im Teil des Listings ab 'Übertragung' baut das Programm künstlich ein falsches Bit ein. Nach der Übertragung decodiert DECODE den verfälschten String und beseitigt dabei den Fehler. Die Tabellen Conv und ErrFlag sind 128 Bytes groß, da ein 7-Bit-Code verwendet wird und im Zielstring jeweils 1 Byte bei der Decodierung entsteht. Es gibt nur richtige und korrigierbare Codes im Bereich der Tabelle. Nicht korrigierbar sind Bytes mit 2 oder mehr fehlerhaften Bits. Diese werden jedoch nicht erkannt. Ein eventuell gesetztes höchstes Bit wird ignoriert, da die Bitgruppen laut Vorgabe nur 7 Bits enthalten.

Programmbeispiel:

```

'-----
' Name: DECODE2.TIG
' Codiert einen String in fehler-korrigierbaren Code.
' Decodiert diesen fehler-korrigierenden Code
' Informiert ueber Fehler und Korrekturen
' Nur Einzelbitfehler werden erkannt und korrigiert.
'-----

user var strict                                ' unbedingte Var.deklaration

STRING source$(256)                            ' Quellstring
STRING encoded$(512)                          ' Quellstring in Nibbles
STRING decoded$(512)                          ' decodierte Nibbles

```

DECODE

3

```

STRING dest$(256)                ' Zielstring

DATALABEL encodtab, decodtab, err_cnt ' Tabellen im Flash

TASK MAIN                        ' Beginn Task MAIN
encodtab::                      ' Encodiertabelle 4-Bit-Daten
data byte " &
00 & ' 00000000b code 00h
07 & ' 00000111b code 01h
19 & ' 00011001b code 02h
1E & ' 00011110b code 03h
2A & ' 00101010b code 04h
2D & ' 00101101b code 05h
33 & ' 00110011b code 06h
34 & ' 00110100b code 07h
4B & ' 01001011b code 08h
4C & ' 01001100b code 09h
52 & ' 01010010b code 0ah
55 & ' 01010101b code 0bh
61 & ' 01100001b code 0ch
66 & ' 01100110b code 0dh
78 & ' 01111000b code 0eh
7F & ' 01111111b code 0fh
"%

'0 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
decodtab::
data byte " &                ' Decodiertabelle 4-Bit-Daten
00 00 00 01 00 01 01 01 01 00 02 04 08 09 05 03 01 &
00 02 0a 06 07 0b 03 01 02 02 03 02 03 02 03 03 &
00 0c 04 06 07 05 0d 01 04 05 04 04 05 05 04 05 &
07 06 06 06 07 07 07 06 0e 02 04 06 07 05 03 0f &
00 0c 0a 08 09 0b 0d 01 09 08 08 08 09 09 09 08 &
0a 0b 0a 0a 0b 0b 0a 0b 0e 02 0a 08 09 0b 03 0f &
0c 0c 0d 0c 0d 0c 0d 0d 0e 0c 04 08 09 05 0d 0f &
0e 0c 0a 06 07 0b 0d 0f 0e 0e 0e 0f 0e 0f 0f 0f &
"%

'0 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
err_cnt::                      ' Tabelle zur Fehlerkontrolle
data byte " &
00 01 01 01 01 01 01 01 00 01 01 01 01 01 01 01 &
01 01 01 01 01 01 01 01 01 01 00 01 01 01 01 00 01 &
01 01 01 01 01 01 01 01 01 01 01 00 01 01 00 01 01 &
01 01 01 00 00 01 01 01 01 01 01 01 01 01 01 01 01 &
01 01 01 01 01 01 01 01 01 01 01 01 01 00 00 01 01 01 &
01 01 00 01 01 00 01 01 01 01 01 01 01 01 01 01 01 &
01 00 01 01 01 01 00 01 01 01 01 01 01 01 01 01 01 &
01 01 01 01 01 01 01 01 00 01 01 01 01 01 01 01 00 &
"%

BYTE b
LONG ok_err                ' ok-error-flag
LONG i, y

' LCD-Treiber installieren (BASIC-Tiger)
install_device #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

```

```

source$ = "" ' bilde einen Quellstring
for i = 00h to 0ffh ' Byte-Codes von...bis
    source$ = source$ + chr$ ( i )
next
for y = 0 to len(encoded$)-1 ' ueber alle Bytes
    for i = 0 to 7 ' ueber 8 Bits
        ' setze Nibbles um in Bytes
        encoded$ = pbcd_bcd$ ( source$ )
        ' Codiere zu fehlerkorr. Code
        encoded$ = convert$ ( encoded$, encodtab, 16 )
        ' -----
        ' Uebertragung
        b = nfroms ( encoded$, y, 1 ) ' nimm ein Byte
        b = b bitxor mask ( i ) ' mache ein Bit kaputt
        encoded$ = ntos$ ( encoded$, y, 1, b )
        ' Ende der Uebertragung
        ' -----

    decoded$ = ""
    ok_err = decode ( &
        decoded$, 1, & ' decodiere und zaehle Fehler
        encoded$, 7, & ' Zielstring, -groesse in Bytes
        decodtab, & ' Quelldaten, Quellgr.: 1...14 Bits
        err_cnt ) ' Converter-Tabelle
        ' Fehlerkontrolltabelle

    dest$ = bcd_pbcd$ ( decoded$ ) ' Nibbles wieder zusammenbauen
    if dest$ <> source$ then ' kann nur bei falscher Tabelle
        print #1, "<1bh>A<0><3><0f0h><7>error???" ' auftreten
        wait_duration 2000 ' da das Bsp nur 1 Bit verfaelscht
    endif
    print #1, "<1bh>A<0><0><0f0h>Byte:";y
    print #1, "<1bh>A<0><1><0f0h> Bit:";i
    using "UH<8><8> 0.0.0.4.4"
    print using #1, "<1bh>A<0><2><0f0h> err:";ok_err
    next ' naechstes Bit
next ' naechstes Byte
print #1, "<1bh>A<0><3><0f0h>ready"
END

```

DELETE_PROG

DELETE_PROG

DELETE_PROG (*Select*)

DELETE_PROG: Löscht das aktuelle BASIC-Programm, PC-Mode
(nur in Projektmodell 'PM_FULL')

Funktion: Löscht das Programm im Flash.

3

Select

ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt, was gelöscht werden soll:

0 = alles, Programm- und User-Datenbereich.

1 = nur Programmbereich, User-Daten bleiben erhalten

n = wie 0

DELETE_PROG wird ab Modulversion 1.06n unterstützt.

Die Modulversion ist per Software Laufzeit abfragbar mit der Funktion SYSVARN, über das Kommando Tiger-Status aus dem Menü Anzeige sowie durch einen Aufkleber auf dem Modul gekennzeichnet.

Löscht das Programm im Flash-Speicher und wahlweise den User-Datenbereich. Versetzt das Modul in den PC-Modus. Der PC-Pin braucht nicht auf ‚low‘ gelegt werden. Das Modul ist danach bereit, ein neues Programm zu laden.

DELETE_PROG kann verwendet werden, um ein entferntes Modul via Modem oder ähnlichen Übertragungsstrecken neu zu laden. Das Anwendungsprogramm empfängt ein Kommando, das Programm zu löschen, und verzweigt zu der Instruktion DELETE_PROG. Ohne Programm im Flash befindet sich das Modul automatisch im PC-Modus. Nach dem Herunterladen des neuen Programms wird dieses sofort gestartet und befindet sich wieder im RUN-Modus. Dem Programm auf dem PC (Compiler-IDE bzw. Downloader) fehlt dadurch die letzte Rückmeldung, die normalerweise gesendet wird, wenn das Modul im PC-Modus ist. Dies führt am PC zu einer Fehlermeldung, die jedoch ignoriert werden kann.

Programmbeispiel:

```
'-----  
' Name: DELETE_PROG.TIG  
' loescht das Programm im Flash und geht per Software in den PC-Modus  
' ab Modulversion 1.06n  
'-----  
TASK Main  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  
  PRINT #1, "program will be"  
  PRINT #1, "deleted in 1 second"  
  PRINT #1, "check in PC mode:"  
  PRINT #1, "view Tigerstatus"  
  WAIT_DURATION 1000  
  DELETE_PROG ( 0 )           ' loescht das Programm im Flash  
  PRINT #1, "<1>FAILED<13><10>program NOT deleted"  
END
```

3

DEVEN

DEVEN

DEVEN ()

DEVEN: Meldet letzten Device Error einer Task

Funktion: Liefert die Fehlernummer des zuletzt in dieser Task aufgetretenen Fehlers in einem Device-Treiber zurück.

3

Funktionswert ist vom Typ LONG und immer positiv. Eine Typwandlung bei der Zuweisung findet automatisch statt.

DEVEN wird verwendet, um Fehler, die in einem Device-Treiber auftreten, zu lokalisieren. Die Fehlernummer bezieht sich auf einen Fehler, der in der Task auftrat, in welcher die Instruktion DEVEN gerade ausgeführt wird.

Programmbeispiel:

```
'-----
' Name: DEVEN.TIG
' Liefert die zuletzt aufgetretene Fehlernummer der installierten
' Device-Treiber
'-----
TASK MAIN                                ' Beginn Task MAIN

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

D = DEVEN ( 0, this_task )                ' letzte Device-Fehler-Nummer
PRINT #1, D                               ' Ausgabe ohne Fehler
PUT #1, FILL$( "x", 257 )                 ' zu grosse Datenmenge
D = DEVEN ( 0, main )                     ' letzte Device-Fehler-Nummer
PRINT #1, D                               ' Ausgabe nach Fehler
END                                         ' Ende Task MAIN
```

Siehe auch: Device-Treiber User-Function-Codes

DIFF_TICKS

DIFF_TICKS (Wert)

DIFF_TICKS: Liefert Zeitdifferenz in 1 ms System-Ticks

Funktion: Liefert die Differenz zwischen dem aktuellen Stand des Tickzählers und **Wert** zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Funktionswert ist vom Typ LONG und immer positiv. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Der Tickzähler zeigt die Anzahl der seit dem letzten Reset vergangenen Millisekunden an. Der Tickzähler wird nie negativ (LONG ist vorzeichenbehaftet). Nach 2.147.483.647 Ticks (ca. 24 Tage) beginnt der Tick-Zähler wieder bei 0. Der Befehl DIFF_TICKS liefert die Differenz zwischen dem angegebenen Wert und dem aktuellen Stand des Tickzählers auch im Moment des Überlaufs noch richtig.

Programmbeispiel:

```

' -----
' Name: DIFF_TICKS1.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG T                                ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  T = TICKS ()                          ' Speichert Ticks in T
  LOOP 100000                            ' 100000 leere Loops
  ENDLOOP
  T = DIFF_TICKS(T)                      ' Zeitdifferenz nach LOOP
  PRINT #1, 100000000/T; " Loops/Sec."   ' Anzeige von LOOPS/Sekunde
END                                        ' Ende Task MAIN

```

DIFF_TICKS

Programmbeispiel:

```
' -----
' Name: DIFF_TICKS2.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG N,T                                ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "<l>Measure reaction time"    ' Ausgabe auf LC-Display
  WAIT DURATION 2000                     ' 2 Sek warten
  GET #1, #0, #1, 1, N                   ' N=Zeichen im Tastatur-Buffer
  WHILE N > 0                             ' Solange Zeichen in Buffer,
    GET #1, 1, A$                         ' Zeichen aus Buffer lesen
    GET #1, #0, #1, 1, N                 ' und neue Anzahl bestimmen
  ENDWHILE
  PRINT #1, "Press key"                  ' Ausgabe auf LC-Display
  T = TICKS ()                           ' Speichert Ticks in T
  FOR N = 0 TO 0 STEP 0                   ' Endlosschleife bis N=1(GET!)
    RELEASE_TASK                          ' Rest der Task-Zeit freigeben
    GET #1, #0, #1, 1, N                 ' N=Zeichen im Tastatur-Buffer
  NEXT                                    ' Ende Endlosschleife
  PRINT #1, "Time: ";DIFF_TICKS(T); " ms" ' Anzeige der Reaktionszeit
END                                        ' Ende Task MAIN
```

Siehe auch: TICKS

DIR_PIN

DIR_PIN Log_Portadr., Bitposition, Richtung

DIR_PIN: Setzt die Datenrichtung eines universellen I/O-Pins

Instruktion: Initialisiert die Datenrichtung eines einzelnen Bits eines bit-orientierten internen Ein-/Ausgabe-Ports.

Log_Portadr. ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Bitposition ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position des Bits an.

Richtung ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Datenrichtung der I/O-Leitungen fest.

Richtung	I/O-Pin
0	Ausgang
1	Eingang

Programmbeispiel:

```

' -----
' Name: DIR_PIN.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  DIR_PIN 8,7,0                          ' Port 8, Bit 7 als Ausgang
  LOOP 9999999                          ' sehr viele Loops
    OUT 8,10000000b, 128                 ' Port 8, Bit 7 auf High
    WAIT_DURATION 500                    ' 500 ms warten
    OUT 8,10000000b, 0                   ' Port 8, Bit 7 auf Low
    WAIT_DURATION 500                    ' 500 ms warten
  ENDLOOP                                '
END                                       ' Ende Task MAIN

```

Siehe auch: DIR_PORT

DIR_PORT

DIR_PORT

DIR_PORT Log_Portadr., Richtung

DIR_PORT: Setzt die Datenrichtung eines universellen I/O-Ports

Instruktion: Initialisiert die Datenrichtung aller Bits eines Ports eines internen Ein-/Ausgabe-Ports.

Log_Portadr. ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Richtung ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Datenrichtung der I/O-Leitungen bitweise fest.

Richtung	I/O-Portbit
0	Pin des internen Ports als Ausgang
1	Pin des internen Ports als Eingang

Die Instruktion

DIR_PORT 8, 1

bewirkt also, daß der Pin 0 des Ports ein Eingang ist, alle anderen Pins ein Ausgang:

DIR_PORT 8, 1							
L87	L86	L85	L84	L83	L82	L81	L80
Ausgang	Ausgang	Ausgang	Ausgang	Ausgang	Ausgang	Ausgang	Eingang

Programmbeispiel:

```
'-----  
' Name: DIR_PORT.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
    DIR_PORT 8,0                          ' Port 8 als Ausgang  
    OUT 8,255,01010101b                   ' Bitmuster auf Port 8  
END                                         ' Ende Task MAIN
```

Siehe auch: DIR_PIN

DISABLE_TSW

DISABLE_TSW

DISABLE_TSW: Schaltet Task-Switching Mechanismus AUS

Instruktion: Sperrt das Task-Switching und stellt der aktuellen Task die volle CPU-Leistung zur Verfügung.

Im Anschluß an diese Instruktion werden nur noch Instruktionen der aktuellen Task ausgeführt. Alle anderen Tasks ruhen so lange, bis dieser Zustand durch die Instruktion **ENABLE_TSW** wieder aufgehoben wird. **DISABLE_TSW** gibt einer Task die Möglichkeit für zeitkritische Programmphasen, vorübergehend die maximale CPU-Leistung für diese eine Aufgabe zu reservieren. Sobald die zeitkritische Aufgabe beendet ist sollte man das Task-Switching sofort wieder erlauben.

Soll lediglich die CPU-Leistung für eine Task insgesamt erhöht werden, ist es normalerweise besser, mit **SET_TASK_PRIO** die Priorität dieser Task entsprechend zu erhöhen.

DISABLE_TSW ist ein einschneidender Eingriff in den Ablauf der Task-Bearbeitung und sollte mit großer Umsicht eingesetzt werden. Wird durch fehlerhafte Programmierung der Task-Switch-Mechanismus nicht weder in Gang gesetzt, bleiben alle anderen Tasks stehen!



Alle anderen Tasks bleiben solange stehen, bis ein **ENABLE_TSW erfolgt!**

Das kurzzeitige Sperren des Task-Switch-Mechanismusses kann auch dazu verwendet werden um ungestört Manipulationen an globalen Variablen vorzunehmen, die von verschiedenen Tasks verändert werden. Durch Anhalten aller anderen Tasks werden Konfliktsituationen und nicht plausible Zustände vermieden.

In dem Beispielprogramm läßt Task 1 ein Lauflicht auf Port 8 laufen. Die globale Lauflichtvariable **I** wird auch von Task 2 benutzt. Probieren Sie das Programm ohne **DISABLE_TSW**: Das Lauflicht läuft schneller, aber nach ca. 20 Durchläufen wird es durch Task 1 gestört.

DISABLE_TSW

Programmbeispiel:

```
' -----
' Name: DISABLE_TSW.TIG
' -----
' Erzeugt ein Lauflicht auf Port 8. Sobald Bit 4 = HIGH ist
' (Diode L84 leuchtet), wird die Task T1 fuer 400ms nicht verlassen.
' -----
BYTE I                                ' I ist globale BYTE-Variable
TASK MAIN                            ' Beginn Task MAIN
  I = 128                            ' Bit 7 vorbesetzen
  DIR_PORT 8,0                      ' Port 8 als Ausgang
  RUN_TASK T1                        ' TASK T1 starten
  LOOP 9999999                      ' sehr viele Loops
    I = I ROL 1                     ' Bitweise links rotieren
    OUT 8, 255, I                   ' Port 8, Bit I anzeigen
    WAIT DURATION 100               ' 100 ms warten
  ENDLLOOP                          '
END                                  ' Ende Task MAIN

' -----
' Task T1 verhindert Taskwechsel fuer 400 ms, falls Bit 4 = HIGH
' -----
TASK T1                              ' Beginn Task T1
  LOOP 9999999                      ' sehr viele Loops
    IF I=16 THEN                   ' Falls Bit 4 = HIGH
      DISABLE_TSW                  ' Taskswitching stoppen
      WAIT DURATION 400             ' 400 ms warten
      ENABLE_TSW                    ' Taskswitching aktivieren
    ENDIF                          '
  ENDLLOOP                          '
END                                  ' Ende Task T1
```

Siehe auch ENABLE_TSW sowie das Beispiel dieser Instruktion.

DISTRIBUTE\$

**DISTRIBUTE\$ (SRC\$, S_Offs, S_Bits, S_Gap, NG,
D_Offs, D_Bits, D_Gap, GapFill, DLF)**

DISTRIBUTE\$: Verteilt Bits in einem String (Bitstream)
(nur in Projektmodell 'PM_FULL')

Funktion: Verteilt Bitgruppen aus dem Quellstring zu Bitgruppen im Zielstring.

Src\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu verteilenden Bits.
S_Offs, D_Offs	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt einen Start-Offset im Quellstring (S_Offs) bzw. Zielstring (D_Offs) an. Bits vor dem Offset bleiben unberücksichtigt. Wertebereich: 0...2.147.483.647
S_Bits, D_Bits	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der Bitgruppen im Quellstring (S_Bits) bzw. Zielstring (D_Bits) an. Wertebereich: 0...255
S_Gap, D_Gap	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, wieviel Bits nach jeder Bitgruppe im Quellstring (S_Gap) bzw. Zielstring (D_Gap) als Lücke behandelt werden. Wertebereich: 0...255
NG	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Bitgruppen an. Wertebereich: 0...65535
GapFill	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, ob die Bits der Lücke mit ,0' oder mit ,1' gefüllt werden oder unverändert bleiben. 0: fülle mit ,0'-Bits 1: fülle mit ,1'-Bits 2: lasse unverändert. Anmerkung: dies gilt nur für die Skip-Bereiche Der Offset-Bereich bleibt immer unberührt.
DLF	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt, wie die Länge des Zielstrings behandelt wird:

DISTRIBUTE\$

- 0: Länge des Zielstrings ergibt sich aus dem Wirken der Funktion DISTRIBUTE\$
- 1: Länge des Zielstrings unverändert lassen, wenn der String länger ist als nötig.
In keinem Fall wird die maximal mögliche Länge des Zielstrings überschritten.

Funktionswert ist vom Typ STRING und enthält die neuverteilten Bits.

3

Beachte: Wenn 1 Parameter einen unzulässigen Wert besitzt, dann ist der Zielstring leer.

DISTRIBUTE\$ überträgt und verteilt die Bitfolge eines Source-Strings in einen Zielstring. Die Reihenfolge der Quellbits bleibt erhalten, jedoch können Lücken in Quell- und/oder Zielstring definiert werden. Typische Anwendungen für DISTRIBUTE sind: Anwendungen für DISTRIBUTE\$:

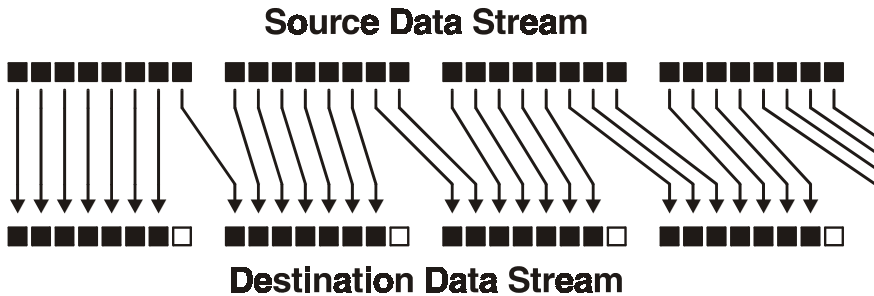
- einen Datenstrom für eine 7-Bit-Übertragung vorbereiten und auf der Empfängerseite dann wieder zu 8-Bit-Daten wandeln
- mehrere Signale (z.B. Stereo-Ton-Aufzeichnungen), bitweise oder byteweise im "Reissverschluss-Verfahren" zusammenfügen
- mehrere Signale (z.B. Stereo-Ton-Aufzeichnungen), bitweise oder byteweise im "Reissverschluss-Verfahren" auseinandernehmen
- als Chiffrierung einsetzen
- Verteilung von Datenbits auf eine Vielzahl von Zielbytes zur Vorbereitung weiterer Codierungs-Schritte, z.B. Code-Converter, Chiffrierung oder fehlerkorrigierender Codes.
- Datenübertragungssicherheit erhöhen. Werden durch eine Störungen viele zusammenliegende Bits zerstört, sind diese mit Fehlersicherungsmethoden nicht mehr zu rekonstruieren. Werden die Bits jedoch verteilt und danach wieder in die ursprüngliche Reihenfolge gebracht, dann finden sich die gestörten Bits nicht mehr an nur einer Stelle.
- Konversion von Datenformaten, zB:
 - 8-Bit ==> 7-Bit
 - 7-Bit ==> 8-Bit
 - 8-Bit ==> 6-Bit
 - 6-Bit ==> 8-Bit
 - 8-Bit ==> 4-Bit
 - etc.
- Restrukturierung von Pixel-Grafik

In den folgenden Beispielen sind als Argumente keine feste Zahlen als Parameter eingesetzt, so wie sonst in einfachen Beispielen üblich. Wird z.B. für Anzahl der Bitgruppen 6 statt 5 angegeben, also mehr Bitgruppen als der Quellstring enthält, dann ist der Zielstring leer. Es ist daher ratsam, die Werte zu berechnen, wenn sie

sich in der Anwendung verändern können. Aus diesem Grunde hier nochmal deutlich:

Liefert DISTRIBUTE\$ einen Leerstring? Dann stimmt ein Parameter nicht!

Die Abbildung zeigt eine einfache Verteilung der Bits, die aus 8-Bit-Daten als Ergebnis 7-Bit-Daten erzeugt. In der Darstellung folgt ein Bit dem anderen, so daß mit Bit 0 begonnen wird und rechts im Byte das 7. Bit erscheint. Das 7. Bit wird in Bit 0 des nächsten Byte übernommen:



DISTRIBUTE\$

Programmbeispiel:

3

```
'-----
' Name: DISTRIBUTE$1.TIG
' Bitverteilung im String
' Es werden alle Bits aus dem Quellstring
'   auf jeweils 7 Bits im Zielstring verteilt
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE NBITS                             ' Anzahl Bits pro Gruppe
  WORD GROUPS                             ' Anzahl Bitgruppen
  LONG S_OFFS, S_SKIP                     ' Start-Offset und Skip-Bits
  STRING SRC$, DEST$                     ' STRING-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
SRC$ = "FF FF"%

S_OFFS = 0                               ' kein Offset
NBITS = 8                                ' 8 Bits pro Gruppe im Src
S_SKIP = 0                               ' 0 im Src ueberspringen
GROUPS = (LEN(SRC$)*8)-S_OFFS             ' bearb. Sourcelaenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP)          ' Anzahl Gruppen

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( SRC$ )        ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( SRC$ )         ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(SRC$):";LEN ( SRC$ )
DEST$ = DISTRIBUTE$ ( &                  ' verteile Bits
  SRC$, &                                ' Quellstring
  S_OFFS, &                               ' Start-Offset in Quellstring
  NBITS, &                                ' Bit/Gruppe im Quellstring
  S_SKIP, &                               ' ueberspringen in Quellstring
  GROUPS, &                               ' Anzahl Gruppen
  0, &                                    ' Start-Offset in Zielstring
  7, &                                    ' Bitgruppenlaenge Zielstring
  1, &                                    ' ueberspringen in Zielstring
  0, &                                    ' uebersprunge Bits sind...
  0 )                                     ' Laenge Zielstring normal

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( DEST$ )        ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( DEST$ )         ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(DEST$):";LEN ( DEST$ )
END                                         ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2"              ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1             ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                             ' neue Zeile
END
```

```

' -----
' Name: SHOW_BINARY_2BYTES
' zeigt 2 Bytes aus String binaer an
' -----
SUB SHOW_BINARY_2BYTES ( STRING BB$ )
  STRING B$
  LONG L

  IF LEN ( BB$ ) > 1 THEN
    L = ASC(MID$(BB$,0,1))          ' erstes Byte
    CALL CVT_BINARY ( L, 8, B$ )   ' binaer wandeln
    PRINT #1, B$;" ";              ' zeige an
    L = ASC(MID$(BB$,1,1))          ' zweites Byte
    CALL CVT_BINARY ( L, 8, B$ )   ' binaer wandeln
    PRINT #1, B$                   ' zeige an
  ENDIF
END

' -----
' Name: CVT_BINARY
' converts a number into binary representation in a string
' -----
SUB cvt_binary ( &
  LONG bin_no; &                  ' Zahl zu konvertieren
  BYTE n; &                        ' Anzahl Bits zu konvertieren
  VAR STRING b$ )                 ' in diesen String

  BYTE i
  LONG b

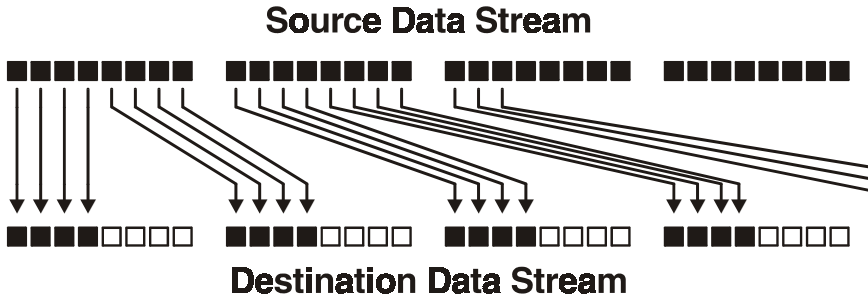
  b = bin_no
  b$=""
  for i = 1 to n
    if bit ( b, 0) = 1 THEN        ' beginne mit b$ = leer
      b$ = "1" + b$               ' soviele Bits
    else                           ' bei Division mit Rest
      b$ = "0" + b$               ' '1' nach vorn
    endif                         ' wenn kein Rest
    b = b shr 1                   ' '0' nach vorn
  next                            ' naechstes Bit ranschieben
END ' end cvt_binary

```

Siehe auch: BIT_MIX\$, BYTE_MIX\$, SCRAMBLE\$

DISTRIBUTE\$

In diesem Beispiel werden aus 8-Bit-Daten jeweils nur 4 Bit in ein Byte übernommen. In der Darstellung folgt ein Bit dem anderen, so daß mit Bit 0 begonnen wird und rechts im Byte das 7. Bit erscheint. Die 4 höherwertigen Bits werden in die niederwertigen Bits des nächsten Byte übernommen:



3

Programmbeispiel:

```

'-----
' Name: DISTRIBUTE$2.TIG
' Bitverteilung im String
' Es werden alle Bits aus dem Quellstring
'   auf jeweils 4 Bits im Zielstring verteilt
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE NBITS                             ' Anzahl Bits pro Gruppe
  WORD GROUPS                             ' Anzahl Bitgruppen
  LONG S OFFS, S SKIP                     ' Start-Offset und Skip-Bits
  STRING SRC$, DEST$                      ' STRING-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
SRC$ = "FF FF"

S OFFS = 0                               ' kein Offset
NBITS = 8                                ' 8 Bits pro Gruppe im Src
S_SKIP = 0                               ' 0 im Src ueberspringen
GROUPS = (LEN(SRC$)*8)-S OFFS             ' bearb. Source-laenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP)         ' Anzahl Gruppen

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( SRC$ )        ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( SRC$ )        ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(SRC$):";LEN ( SRC$ )
DEST$ = DISTRIBUTE$ ( &
  SRC$, &                                ' verteile Bits
  S OFFS, &                              ' Quellstring
  NBITS, &                               ' Start-Offset in Quellstring
  S_SKIP, &                              ' Bit/Gruppe im Quellstring
  GROUPS, &                              ' ueberspringen in Quellstring
  0, &                                   ' Anzahl Gruppen
  4, &                                   ' Start-Offset in Zielstring
  4, &                                   ' Bitgruppenlaenge Zielstring
  0, &                                   ' ueberspringen in Zielstring
  0 )                                     ' uebersprunge Bits sind...
                                           ' Laenge Zielstring normal

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( DEST$ )       ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( DEST$ )       ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(DEST$):";LEN ( DEST$ )
END                                       ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2"             ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1           ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                           ' neue Zeile
END

```

DISTRIBUTE\$

3

```
'-----
' Name: SHOW_BINARY_2BYTES
' zeigt 2 Bytes aus String binaer an
'-----
SUB SHOW_BINARY_2BYTES ( STRING BB$ )
  STRING B$
  LONG L

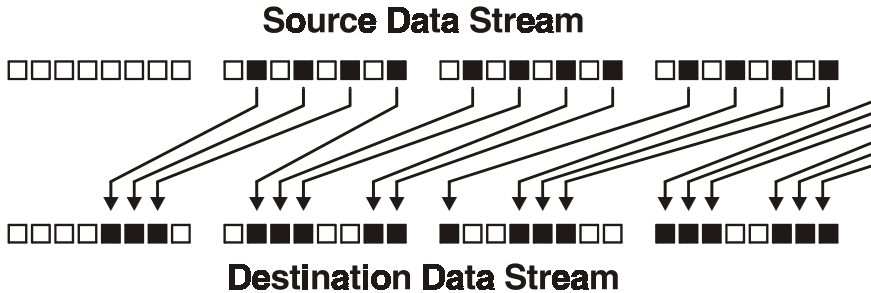
  IF LEN ( BB$ ) > 1 THEN
    L = ASC(MID$(BB$,0,1))      ' erstes Byte
    CALL CVT_BINARY ( L, 8, B$ ) ' binaer wandeln
    PRINT #1, B$;" ";          ' zeige an
    L = ASC(MID$(BB$,1,1))      ' zweites Byte
    CALL CVT_BINARY ( L, 8, B$ ) ' binaer wandeln
    PRINT #1, B$               ' zeige an
  ENDIF
END

'-----
' Name: CVT_BINARY
' converts a number into binary representation in a string
'-----
SUB cvt_binary ( &
  LONG bin_no; &              ' Zahl zu konvertieren
  BYTE n; &                   ' Anzahl Bits zu konvertieren
  VAR STRING b$ )              ' in diesen String

  BYTE i
  LONG b

  b = bin_no
  b$=""
  for i = 1 to n
    if bit ( b, 0) = 1 THEN     ' beginne mit b$ = leer
      b$ = "1" + b$           ' soviele Bits
    else                       ' bei Division mit Rest
      b$ = "0" + b$           ' '1' nach vorn
    endif                     ' wenn kein Rest
    b = b shr 1                ' '0' nach vorn
  next                         ' naechstes Bit ranschieben
END ' end cvt_binary
```

Wie die Abbildung zeigt, werden hier zunächst 9 Bits im Quellbereich übersprungen. Dann beginnt eine Bitverteilung, die nur die geraden Bits berücksichtigt. Im Zielstring werden 4 Bits übersprungen, die auch nie mit Füllbits verändert werden. Dann werden die verteilten Bits in Dreiergruppen aufgesammelt und jeweils zwei Bits übersprungen, die mit dem angegebenen Füllbit (0) aufgefüllt werden.



DISTRIBUTE\$

Programmbeispiel:

3

```

'-----
' Name: DISTRIBUTE$3.TIG
' Bitverteilung im String
' Es werden ab dem 2. Byte alle geraden Bits aus dem Quellstring
'   auf jeweils 3 Bits mit einem Offset von 4 im Zielstring verteilt
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE NBITS                             ' Anzahl Bits pro Gruppe
  WORD GROUPS                             ' Anzahl Bitgruppen
  LONG S_OFFS, S_SKIP                     ' Start-Offset und Skip-Bits
  STRING SRC$, DEST$                     ' STRING-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
SRC$ = "FF FF FF"%

S_OFFS = 9                               ' Offset 9
NBITS = 1                                ' 1 Bit pro Gruppe im Src
S_SKIP = 1                               ' 1 im Src ueberspringen
GROUPS = (LEN(SRC$)*8)-S_OFFS             ' bearb. Sourcelaenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP)          ' Anzahl Gruppen

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( SRC$ )        ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( SRC$ )         ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(SRC$):";LEN ( SRC$ )
DEST$ = "03 FF 7F"%                      ' Bytes im Offsetbereich werden
                                          ' nicht 0, aber len angepasst
DEST$ = DISTRIBUTE$ ( &                 ' verteile Bits
  SRC$, &                                ' Quellstring
  S_OFFS, &                              ' Start-Offset in Quellstring
  NBITS, &                               ' Bit/Gruppe im Quellstring
  S_SKIP, &                              ' ueberspringen in Quellstring
  GROUPS, &                              ' Anzahl Gruppen
  4, &                                   ' Start-Offset in Zielstring
  3, &                                   ' Bitgruppenlaenge Zielstring
  2, &                                   ' ueberspringen in Zielstring
  0, &                                   ' uebersprunge Bits sind...
  0 )                                    ' Laenge Zielstring normal

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( DEST$ )       ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( DEST$ )        ' zeige 2 Bytes aus Zielstring
PRINT #1, "LEN(DEST$):";LEN ( DEST$ )    ' oberstes Bit ist unberuehrt
END                                        ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2>  0.0.0.0.2"             ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1             ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT

```



```

PRINT #1, "" ' neue Zeile
END

' -----
' Name: SHOW_BINARY_2BYTES
' zeigt 2 Bytes aus String binaer an
' -----
SUB SHOW_BINARY_2BYTES ( STRING BB$ )
  STRING B$
  LONG L

  IF LEN ( BB$ ) > 1 THEN
    L = ASC(MID$(BB$,0,1)) ' erstes Byte
    CALL CVT_BINARY ( L, 8, B$ ) ' binaer wandeln
    PRINT #1, B$;" "; ' zeige an
    L = ASC(MID$(BB$,1,1)) ' zweites Byte
    CALL CVT_BINARY ( L, 8, B$ ) ' binaer wandeln
    PRINT #1, B$ ' zeige an
  ENDIF
END

' -----
' Name: CVT_BINARY
' converts a number into binary representation in a string
' -----
SUB cvt_binary ( &
  LONG bin_no; & ' Zahl zu konvertieren
  BYTE n; & ' Anzahl Bits zu konvertieren
  VAR STRING b$ ) ' in diesen String

  BYTE i
  LONG b

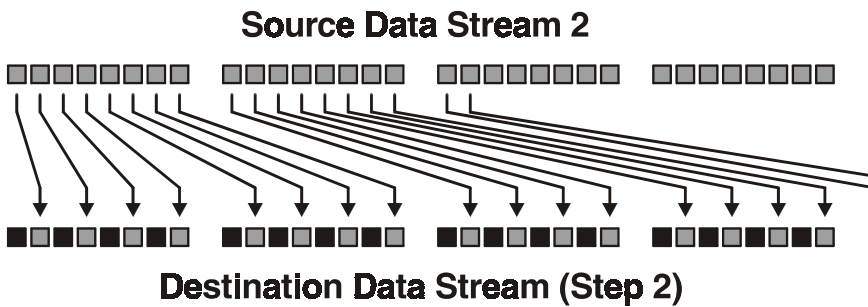
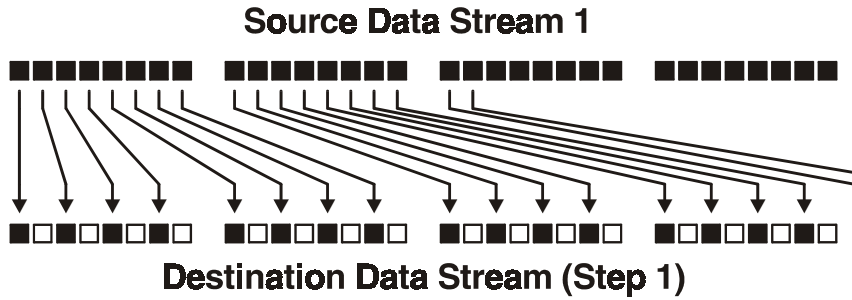
  b = bin_no
  b$=""
  for i = 1 to n
    if bit ( b, 0) = 1 THEN ' beginne mit b$ = leer
      b$ = "1" + b$ ' soviele Bits
    else ' bei Division mit Rest
      b$ = "0" + b$ ' '1' nach vorn
    endif ' wenn kein Rest
    b = b shr 1 ' '0' nach vorn
  next ' naechstes Bit ranschieben
END ' end cvt_binary

```

DISTRIBUTE\$

Es werden in zwei Schritten die Bits aus zwei verschiedenen Strings in einem String zusammengesetzt. Dabei werden die Bits des einen Strings zu den geraden Bits, die Bits des anderen Strings zu den ungeraden Bits im Zielstring.

3



Programmbeispiel:

```

'-----
' Name: DISTRIBUTE$4.TIG
' Bitverteilung im String
' Es werden in zwei Schritten die Bits aus zwei verschiedenen Strings
' in einem String zusammengesetzt. Dabei werden die Bits des einen
' Strings zu den geraden Bits, die Bits des anderen Strings zu den
' ungeraden Bits im Zielstring.
'-----
user var strict          ' unbedingte Var.deklaration
TASK MAIN                ' Beginn Task MAIN
  BYTE NBITS              ' Anzahl Bits pro Gruppe
  WORD GROUPS             ' Anzahl Bitgruppen
  LONG S_OFFS, S_SKIP     ' Start-Offset und Skip-Bits
  STRING SRC1$, SRC2$, DEST$ ' STRING-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
SRC1$ = "FF 0F"%
SRC2$ = "0F 00"%

S_OFFS = 0                ' Im Source-String Offset 0
NBITS = 1                 ' 1 Bit pro Gruppe im Src
S_SKIP = 0                ' 0 im Src ueberspringen
GROUPS = (LEN(SRC1$)*8)-S_OFFS ' bearb. Source-laenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP) ' Anzahl Gruppen

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( SRC1$ ) ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( SRC1$ ) ' zeige 2 Bytes aus Zielstring
' CALL SHOW_STRING_AS_HEX ( SRC2$ ) ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( SRC2$ ) ' zeige 2 Bytes aus Zielstring
DEST$ = DISTRIBUTE$ ( &
  SRC1$, &                ' verteile Bits
  S_OFFS, &               ' Quellstring 1
  NBITS, &                ' Start-Offset in Quellstring
  S_SKIP, &               ' Bit/Gruppe im Quellstring
  GROUPS, &               ' ueberspringen in Quellstring
  0, &                    ' Anzahl Gruppen
  1, &                    ' Start-Offset in Zielstring
  1, &                    ' Bitgruppenlaenge Zielstring
  0, &                    ' ueberspringen in Zielstring
  0, &                    ' uebersprungene Bits sind...
  0 )                     ' Laenge Zielstring normal

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( DEST$ ) ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( DEST$ ) ' zeige 2 Bytes aus Zielstring

S_OFFS = 0                ' Im Source-String Offset 0
NBITS = 1                 ' 1 Bit pro Gruppe im Src
S_SKIP = 0                ' 0 im Src ueberspringen
GROUPS = (LEN(SRC2$)*8)-S_OFFS ' bearb. Source-laenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP) ' Anzahl Gruppen

DEST$ = DISTRIBUTE$ ( &
  SRC2$, &                ' verteile Bits
  S_OFFS, &               ' Quellstring 2
  S_SKIP, &               ' Start-Offset in Quellstring

```

DISTRIBUTE\$

3

```

NBITS, &                                ' Bit/Gruppe im Quellstring
S_SKIP, &                              ' ueberspringen in Quellstring
GROUPS, &                              ' Anzahl Gruppen
1, &                                    ' Start-Offset in Zielstring
1, &                                    ' Bitgruppenlaenge Zielstring
1, &                                    ' ueberspringen in Zielstring
2, &                                    ' uebersprunge Bits bleiben!
0 )                                     ' Laenge Zielstring normal

' fuer Experimente mit laengeren Strings
' CALL SHOW_STRING_AS_HEX ( DEST$ )    ' zeige Quellstring an
CALL SHOW_BINARY_2BYTES ( DEST$ )     ' zeige 2 Bytes aus Zielstring
END                                     ' Ende Task MAIN

' -----
' zeigt den String als HEX-Bytes
' -----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2>  0.0.0.0.2"          ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1          ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                          ' neue Zeile
END

' -----
' Name: SHOW_BINARY_2BYTES
' zeigt 2 Bytes aus String binaer an
' -----
SUB SHOW_BINARY_2BYTES ( STRING BB$ )
  STRING B$
  LONG L

  IF LEN ( BB$ ) > 1 THEN
    L = ASC(MID$(BB$,0,1))              ' erstes Byte
    CALL CVT_BINARY ( L, 8, B$ )        ' binaer wandeln
    PRINT #1, B$;" ";                  ' zeige an
    L = ASC(MID$(BB$,1,1))              ' zweites Byte
    CALL CVT_BINARY ( L, 8, B$ )        ' binaer wandeln
    PRINT #1, B$                        ' zeige an
  ENDIF
END

' -----
' Name: CVT_BINARY
' converts a number into binary representation in a string
' -----
SUB cvt_binary ( &
  LONG bin_no; &                        ' Zahl zu konvertieren
  BYTE n; &                             ' Anzahl Bits zu konvertieren
  VAR STRING b$ )                       ' in diesen String

  BYTE i
  LONG b

  b = bin_no

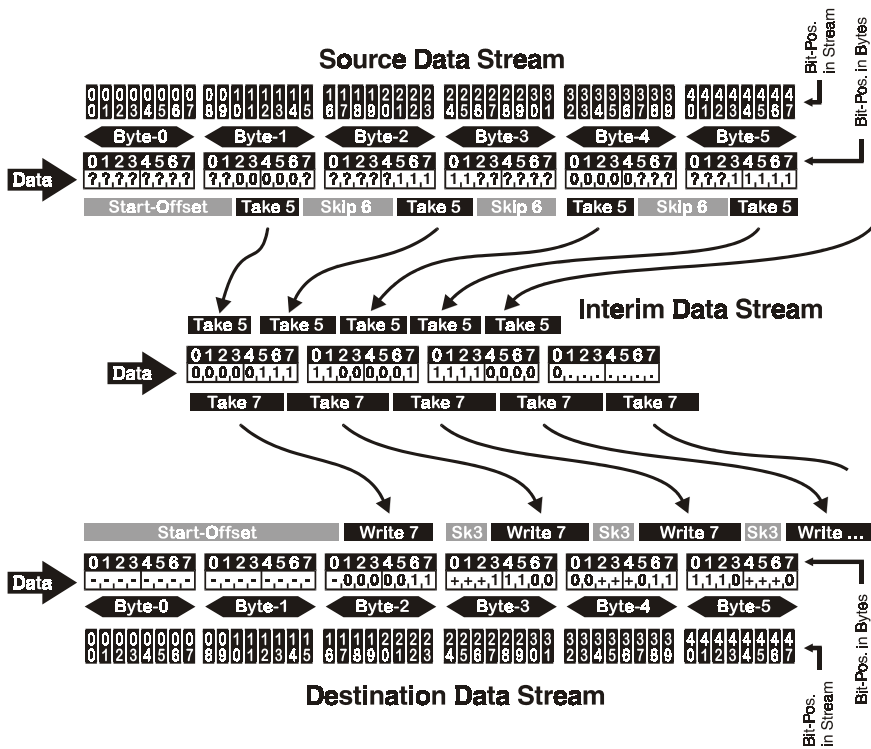
```

```

b$=""
for i = 1 to n
  if bit ( b, 0) = 1 THEN
    b$ = "1" + b$
  else
    b$ = "0" + b$
  endif
  b = b shr 1
next
END ' end cvt_binary

```

' beginne mit b\$ = leer
' soviele Bits
' bei Division mit Rest
' '1' nach vorn
' wenn kein Rest
' '0' nach vorn
' naechstes Bit ranschieben



DISTRIBUTE\$

Programmbeispiel:

3

```
'-----
' Name: DISTRIBUTE$5.TIG
' Bitverteilung im String
'-----
user var strict          ' unbedingte Var.deklaration
TASK MAIN                ' Beginn Task MAIN
  BYTE NBITS              ' Anzahl Bits pro Gruppe
  WORD GROUPS             ' Anzahl Bitgruppen
  LONG S_OFFSETS, S_SKIP  ' Start-Offset und Skip-Bits
  STRING SRC$, INTERIM$, DEST$ ' STRING-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
SRC$ = "55 83 FF FF E0 FF 00 00 00"%

S_OFFSETS = 10            ' Im Source-String Offset 10
NBITS = 5                 ' 5 Bit pro Gruppe im Src
S_SKIP = 6                ' 6 im Src ueberspringen
GROUPS = (LEN(SRC$)*8)-S_OFFSETS ' bearb. Sourcelaenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP) ' Anzahl Gruppen

CALL SHOW_STRING_AS_HEX ( SRC$ ) ' zeige Quellstring an
INTERIM$ = DISTRIBUTE$ ( &      ' verteile Bits
  SRC$, &                      ' Quellstring
  S_OFFSETS, &                 ' Start-Offset in Quellstring
  NBITS, &                     ' Bit/Gruppe im Quellstring
  S_SKIP, &                    ' ueberspringen in Quellstring
  GROUPS, &                    ' Anzahl Gruppen
  0, &                          ' Start-Offset in Zielstring
  5, &                          ' Bitgruppenlaenge Zielstring
  0, &                          ' ueberspringen in Zielstring
  0, &                          ' uebersprunge Bits sind...
  0 )                          ' Laenge Zielstring normal

CALL SHOW_STRING_AS_HEX ( INTERIM$ ) ' zeige Quellstring an

S_OFFSETS = 0              ' Im Source-String Offset 0
NBITS = 7                  ' 1 Bit pro Gruppe im Src
S_SKIP = 0                 ' 0 im Src ueberspringen
GROUPS = (LEN(INTERIM$)*8)-S_OFFSETS ' bearb. Sourcelaenge in Bits
GROUPS = GROUPS / (NBITS+S_SKIP) ' Anzahl Gruppen

DEST$ = DISTRIBUTE$ ( &      ' verteile Bits
  INTERIM$, &                ' Quellstring 2
  S_OFFSETS, &                ' Start-Offset in Quellstring
  NBITS, &                    ' Bit/Gruppe im Quellstring
  S_SKIP, &                    ' ueberspringen in Quellstring
  GROUPS, &                    ' Anzahl Gruppen
  17, &                        ' Start-Offset in Zielstring
  7, &                          ' Bitgruppenlaenge Zielstring
  3, &                          ' ueberspringen in Zielstring
  2, &                          ' uebersprunge Bits bleiben!
  0 )                          ' Laenge Zielstring normal

CALL SHOW_STRING_AS_HEX ( DEST$ ) ' zeige Quellstring an
END                                ' Ende Task MAIN
```

```
'-----  
' zeigt den String als HEX-Bytes  
'-----  
SUB SHOW_STRING_AS_HEX ( STRING S$ )  
  BYTE I  
  USING "UH<2><2>  0.0.0.0.2"      ' Format HEX-Ausgabe  
  FOR I = 0 TO LEN ( S$ ) - 1      ' alle Bytes des Strings  
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX  
  NEXT  
  PRINT #1, ""                     ' neue Zeile  
END
```

Siehe auch: SCRAMBLE\$

ENABLE_TSW

ENABLE_TSW

ENABLE_TSW

ENABLE_TSW: Schaltet Task-Switching Mechanismus EIN

Instruktion: Hebt die Sperrung des Task-Switchings auf. Alle Tasks laufen entsprechend Ihrer Priorität weiter.

Programmbeispiel:

3

```
-----
' Name: ENABLE.TIG
-----
' Erzeugt ein Lauflicht auf Port 8. Sobald Bit 4 = HIGH ist
' (Diode L84 leuchtet), wird die Task T1 fuer 400ms nicht verlassen.
-----
BYTE I                                ' I ist globale BYTE-Variable
TASK MAIN                             ' Beginn Task MAIN
  I = 128                             ' Bit 7 vorbesetzen
  DIR_PORT 8,0                       ' Port 8 als Ausgang
  RUN_TASK T1                        ' TASK T1 starten
  LOOP 9999999                      ' sehr viele Loops
    I = I ROL 1                      ' Bitweise links rotieren
    OUT 8, 255, I                   ' Port 8, Bit I anzeigen
    WAIT DURATION 100               ' 100 ms warten
  ENDL00P
END                                  ' Ende Task MAIN

-----
' Task T1 verhindert Taskwechsel fuer 400 ms, falls Bit 4 = HIGH
-----
TASK T1                               ' Beginn Task T1
  LOOP 9999999                      ' sehr viele Loops
    IF I=16 THEN                   ' Falls Bit 4 = HIGH
      DISABLE_TSW                  ' Taskswitching stoppen
      WAIT DURATION 400            ' 400 ms warten
      ENABLE_TSW                   ' Taskswitching aktivieren
    ENDIF
  ENDL00P
END                                  ' Ende Task T1
```

Siehe auch DISABLE_TSW sowie das Beispiel dieser Instruktion.

END**END**

END: Beendet ein Unterprogramm (führt Return aus) oder eine Task.

Instruktion: Beendet die Definition einer Task oder eines Unterprogramms.

Unterprogramm Steht END am Ende eines Unterprogramms, so hat diese Instruktion zusätzlich die Wirkung eines RETURNS.

Task Steht END am Ende einer Task, wird diese Task beendet und die CPU-Zeit dieser Task freigegeben.

Programmbeispiel:

```
' -----
' Name: END.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "Hello World"                ' Auf LC-Display ausgeben
END                                       ' Ende Task MAIN
```

ERASE_FLASH

ERASE_FLASH

ERASE_FLASH Startadresse, Länge[, Fehlerbehandlung]

ERASE_FLASH: Löscht FLASH-Sektor(en), nur DATEN-FLASH Bereich

Instruktion: Löscht einen oder mehrere Sektoren des FLASH-Speichers.

3

Tiger-BASIC®-Programme können FLASH-Bereiche, die nicht vom Programm-Code belegt sind, auch zur Speicherung von Daten benutzen. Dazu wird die entsprechende Anzahl freier Sektoren zur Verfügung gestellt. Die erste zur Datenspeicherung verwendbare FLASH-Adresse ist 0, die letzte verwendbare Adresse ergibt sich aus der Länge des freien FLASH-Bereichs. Genaue Werte erhält man durch Abfrage der System-Variablen mit der Funktion SYSVARx (siehe ab Seite 458).

Mit ERASE_FLASH kann man, entsprechend der Organisation von FLASH-Speichern, eine bestimmte Anzahl von Sektoren löschen. Damit diese Instruktion ausgeführt wird, muß man die genaue Startadresse des Sektors kennen und die Lösch-Länge muß genau einem ganzen Vielfachen der Sektor-Länge entsprechen. Sonst wird dieses Kommando zur Laufzeit nicht ausgeführt (erzeugt Runtime-Fehler). Wird das ERASE_FLASH Kommando erfolgreich abgesetzt, ist der FLASH für eine kurze Zeit busy und kann nicht mehr angesprochen werden (typisch 0,5...5 Sekunden). In Modulen, die nicht über einen getrennten Daten- und Programm-Flash verfügen, führt dies zu einer Unterbrechung des Programmablaufs aller Tasks während der Löschdauer!

Startadresse ist die FLASH-Adresse, ab der der Löschvorgang beginnen soll. Sie muß genau auf einem Sektoranfang liegen.

Länge ist die Anzahl Bytes, die gelöscht werden soll. Länge muß stets ein ganzes Vielfaches der Sektor-Länge betragen, andernfalls wird kein Löschvorgang gestartet.

Diese Instruktion kann eine eigene **Fehlerbehandlung** in Form eines Unterprogramms oder einer Verzweigung verwenden. Schreibweise:

ON_ERROR_CALL *Unterprogramm*

ON_ERROR_GOTO *Sprungmarke*



ERASE_FLASH wartet solange bis der Flash mit dem löschen fertig ist. Wird zum Beispiel ein Datensektor gelöscht, so bleiben in Modulen mit einem Flash für Daten und Programm solange wie der Löschvorgang dauert (einige hunder msec bis zu 15 sec)
alle Tasks stehen
die I/O-Aktivität frieren ein
die System-Ticks zählen nicht weiter
das stellen der Uhr wird gestört (das-Stellen der Uhr dauert einige Sekunden)
und die Device-Treiber stellen Ihre Aktivitäten ein
Alle zeitkritischen Vorgänge müssen also vor Löschen eines oder mehrerer Flashsektoren in einen kontrollierten Zustand gebracht werden.

3

Weitere Informationen über den Flash-Speicher finden Sie im Hardware-Handbuch.

ERASE_FLASH

Programmbeispiel:

```
'-----
' Name: ERASE_FLASH.TIG
'-----
' Schreibt Werte in den FLASH-Speicher und gibt diesen dann 2 mal
' aus. Vor der zweiten Ausgabe wird der FLASH-Speicher geloescht.
'-----
USER_VAR_STRICT
TASK MAIN                                ' Beginn Task MAIN
  BYTE I,J,K                             ' Variablendeklaration
  LONG S
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
FOR I = 0 TO 10                          ' For...Next Schleife
  POKE_FLASH I,2*i,0                     ' 2*i ins Flash schreiben
NEXT
FOR J = 1 TO 2                           ' 2 Durchlaeufer fuer Ausgabe
  IF J=2 THEN                            ' Falls zweiter Durchlauf,
    S = SYSVARN ( 35, 1 )                ' Flash-Sektorgroesse in Bytes
    ERASE_FLASH 0,S                      ' loesche Flash-Speicher
  ENDIF
  FOR I = 0 TO 20                         ' For...Next Schleife
    PEEK_FLASH I,K,1                     ' Jeweils 1 Byte aus Flash
    PRINT #1, "<1>Value"; I; " ="; K      ' auf LC-Display ausgeben
    WAIT DURATION 1500                   ' 2 Sek warten
  NEXT
NEXT
END                                       ' Ende Task MAIN
```

3

EXIT_TASK

EXIT_TASK *Name*

EXIT_TASK: Beendet Ausführung einer Task

Instruktion: Beendet die Task **Name**.

Eine Task kann sich selbst oder jede andere Task beenden.

Eine mit EXIT_TASK beendete Task kann nicht mehr mit CONT_TASK weitergeführt werden

Statische und gepufferte I/Os (z.B. serielle Ein/Ausgabe) laufen ggf. noch weiter.

Lokale Variable gehen verloren.

3

EXIT_TASK

Programmbeispiel:

```
' -----
' Name: EXIT_TASK.TIG
' -----
' Zaehlt die Variablen I (in Task MAIN) und J (in Task T1) hoch.
' Ist I groesser als 300, wird die Ausfuehrung der Task T1
' gestoppt und J nicht weiter hochgezaehlt.
' -----
LONG I, J                                ' Globale Var. deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
RUN_TASK T1                              ' Task T1 starten
FOR I = 0 TO 1000                          ' Zaehlschleife fuer I
  IF I > 300 THEN                          ' Ist I > 300 ?
    EXIT_TASK T1                          ' dann Task T1 beenden
  ENDIF
  PRINT #1, "<1>I =";I                    ' I anzeigen
  PRINT #1, "J =";J                      ' J anzeigen
  WAIT DURATION 100                      ' 100 ms warten
NEXT                                      ' naechster Wert
END                                       ' Ende Task MAIN

' -----
' Task T1 zaehlt globale Variable J hoch
' -----
TASK T1                                  ' Beginn Task T1
FOR J = 0 TO 1000                          ' Zaehlschleife fuer J
  WAIT DURATION 100                      ' 100 ms warten
NEXT                                      ' naechster Wert
END                                       ' Ende Task T1
```

Siehe auch: RUN_TASK, CONT_TASK, STOP_TASK

EXP

EXP (Basis, Exp)

EXP: Exponential-Funktion für Integer-Werte

Funktion: Liefert die angegebene Potenz „**Basis** hoch **Exp**“ zurück.

Basis ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Exp ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Negative Exponenten liefern als Ergebnis 0, ein Überlauf wird ignoriert und liefert ein falsches Ergebnis.

Programmbeispiel:

```

' -----
' Name: EXP.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  BYTE X,Y                              ' Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  FOR X = 0 TO 7                          ' FOR..NEXT-Schleife
    Y = EXP (2,X)                         ' Potenz bilden
    PRINT #1, "<1>2 power"; X; " ="; Y    ' Ausgabe auf LC-Display
    WAIT DURATION 1000                    ' 1 Sek warten
  NEXT                                    ' Ende FOR...NEXT
END                                        ' Ende Task MAIN

```

Siehe auch: LN, LOG, LD

EXPE

EXPE

EXPE (Wert)

EXPE: Exponential-Funktion zur Basis e, Real-Werte

Funktion: Liefert die durch **Wert** angegebene Potenz der Zahl e, wobei e die Basis des natürlichen Logarithmus ist.

3

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -709 bis 709. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
'-----
' Name: EXPE.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG value                             ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
  FOR value = -50 TO 50 STEP 2            ' Wert von -50 bis 50
    PRINT #1, "<1>EXPE ("; value; ")" = " ' Zeige Argument
    PRINT USING #1, EXPE (value/10.0)     ' Exponentialwert ausgeben
    WAIT DURATION 1000                   ' 1 sek warten
  NEXT                                    ' naechster Wert
END                                        ' Ende Task MAIN
```

Siehe auch: LN, LOG, LD

FIFO

FIFO Name (Anzahl) OF Datentyp

FIFO: Deklaration einer Variablen vom Typ FIFO

(FIFO=First-In-First-Out) Instruktion: Deklariert einen FIFO-Puffer **Name** mit der angegebenen **Anzahl** Elemente des angegebenen **Datentyps**.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG. Anzahl ist minimal 64 und immer eine Potenz von 2.

Datentyp kann BYTE, WORD oder LONG sein.

3



FIFO-Puffer werden stets innerhalb der MAIN Task deklariert. Dennoch sind sie per Definition stets global. Die FIFO-Deklaration erzeugt Code, der zur Initialisierung des FIFO führt. Der Code muß vor der ersten FIFO-Operation ausgeführt sein.

FIFO-Puffer werden eingesetzt um serielle Datenströme zu puffern.

Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```

' -----
' Name: FIFO.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER(512) OF WORD                ' FIFO-Buffer definieren
  WORD I,J                                ' WORD-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 511                          ' 512 Werte in den
    PUT_FIFO BUFFER, 2*I                    ' FIFO-Buffer schreiben
  NEXT I
  FOR I = 0 TO 18                            ' Die ersten 19 Werte
    GET_FIFO BUFFER, J                      ' aus dem FIFO-Buffer lesen
    PRINT #1, "<1>Buffer"; I; " = "; J      ' Wert auf LC-Display ausgeben
    WAIT_DURATION 1000                     ' 1 Sek warten
  NEXT I
END                                          ' Ende Task MAIN

```

Siehe auch: GET_FIFO, PUT_FIFO, INTEGRAL_FIFO

FILL\$

FILL\$

FILL\$ (*FillString*, *Len*)

FILL\$: Füllt String mit bel. Muster-String

Funktion: Liefert einen String mit der Länge **Len**, der mit **FillString** gefüllt ist.

FillString ist vom Typ STRING und enthält den String, mit dem der Ergebnisstring gefüllt wird.

Len ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Funktionswert ist vom Typ STRING und hat die angegebene Länge oder die maximale Stringlänge.

FILL\$ liefert als Ergebnis einen String, der über die gesamte Länge mit dem angegebenen Füllstring gefüllt ist. Die maximale Stringlänge wird nicht überschritten.

Programmbeispiel:

```
'-----
' Name: FILL$1.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  B$ = "Hello"                          ' Vorbesetzung von B$
  A$ = FILL$ ( B$, 40 )                  ' A$ bis Len=40 mit B$ füllen
  PRINT #1, "Length of A$:"; LEN (A$)    ' Laenge von A$ ausgeben
  PRINT #1, A$                          ' A$ auf LC-Display ausgeben
END                                       ' Ende Task MAIN
```

Siehe auch: NTO\$\$, RTO\$\$, STOSS

Ein weiteres Beispielprogramm zeigt, wie mit der Funktion FILL\$ eine berechnete Textgrafik ausgegeben wird. Das Ergebnis finden Sie auf den folgenden Seiten.

Programmbeispiel:

```

' -----
' Name: FILL$2.TIG
' -----
#INCLUDE DEFINE_A.INC           ' allg. Symbol-Definitionen
#DEFINE PI_180 0.0174532925199433 ' Konstante PI/180 definieren

TASK MAIN                       ' Start Task MAIN
  STRING A$                     ' String-Variable deklarieren
  LONG DEG,N                   ' LONG-Variablen deklarieren
  REAL RAD                     ' REAL-Variable deklarieren
  INSTALL DEVICE #2, "SER1B_K1.TDD",& ' ser. Treiber installieren
  BD_19_200, DP_8N, JA,&        ' Einstellungen Kanal 0
  BD_38_400, DP_8E, JA        ' Einstellungen Kanal 1

  A$ = "The quick brown fox jumps over the lazy dog. "

  FOR DEG = 0 TO 720 STEP 5      ' Winkel von 0 bis 720 Grad
    RAD = DEG * PI_180          ' Winkel: Grad- zu Bogenmass
    N = RTL (31 + 30 * SIN(RAD)) ' Anzahl Zeichen fuer Ausgabe
    PRINT #2, #0, FILL$ (A$, N) ' Ausgabe mit FILL$
  NEXT                          ' naechster Winkelwert
END                             ' Ende Task MAIN

```

FILL\$

3

●	The quick brown fox jumps over	●
	The quick brown fox jumps over th	
	The quick brown fox jumps over the l	
●	The quick brown fox jumps over the laz	●
	The quick brown fox jumps over the lazy d	
●	The quick brown fox jumps over the lazy dog	●
	The quick brown fox jumps over the lazy dog. T	
●	The quick brown fox jumps over the lazy dog. The	●
	The quick brown fox jumps over the lazy dog. The q	
●	The quick brown fox jumps over the lazy dog. The qui	●
	The quick brown fox jumps over the lazy dog. The quick	
●	The quick brown fox jumps over the lazy dog. The quick b	●
	The quick brown fox jumps over the lazy dog. The quick bro	
●	The quick brown fox jumps over the lazy dog. The quick brow	●
	The quick brown fox jumps over the lazy dog. The quick brow	
●	The quick brown fox jumps over the lazy dog. The quick brown	●
	The quick brown fox jumps over the lazy dog. The quick brown	
●	The quick brown fox jumps over the lazy dog. The quick brown	●
	The quick brown fox jumps over the lazy dog. The quick brown	
●	The quick brown fox jumps over the lazy dog. The quick brown	●
	The quick brown fox jumps over the lazy dog. The quick brown	
●	The quick brown fox jumps over the lazy dog. The quick brow	●
	The quick brown fox jumps over the lazy dog. The quick bro	
●	The quick brown fox jumps over the lazy dog. The quick b	●
	The quick brown fox jumps over the lazy dog. The quick	
●	The quick brown fox jumps over the lazy dog. The quic	●
	The quick brown fox jumps over the lazy dog. The qui	
●	The quick brown fox jumps over the lazy dog. The q	●
	The quick brown fox jumps over the lazy dog. The	
●	The quick brown fox jumps over the lazy dog.	●
	The quick brown fox jumps over the lazy dog	
●	The quick brown fox jumps over the lazy d	●
	The quick brown fox jumps over the laz	
●	The quick brown fox jumps over the l	●
	The quick brown fox jumps over th	
●	The quick brown fox jumps over	●
	The quick brown fox jumps ov	
●	The quick brown fox jumps	●
	The quick brown fox jum	
●	The quick brown fox	●
	The quick brown fo	
●	The quick brown	●
	The quick bro	
●	The quick b	●
	The quick	
●	The quic	●
	The qu	
●	The q	●
	The	
●	Th	●

FIX

FIX (Wert)

FIX: Schneidet Nachkommastellen eines REAL-Wertes ab

Funktion: Liefert den Wert als Ganzzahl ohne Nachkommastellen zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Wertebereich der Zuweisungsvariablen. Nachkommastellen gehen dabei verloren.

Funktionswert ist vom Typ REAL. Eine Typwandlung bei der Zuweisung findet automatisch statt.

FIX schneidet Nachkommastellen ab, so daß zu '-3.8' der Funktionswert '-3.0' zurückgegeben wird. Die Funktion INT liefert als Funktionswert den nächstkleineren Wert, würde also bei dem Argument '-3.8' den Funktionswert '-4' liefern.

Programmbeispiel:

```

' -----
' Name: FIX.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  L = FIX (-2.7)                          ' Nachkommastellen entfernen
  PRINT #1, "FIX (-2.7) ="; L              ' Ausgabe auf LC-Display
  L = FIX (-2.3)                          ' Nachkommastellen entfernen
  PRINT #1, "FIX (-2.3) ="; L              ' Ausgabe auf LC-Display
  L = FIX (2.3)                           ' Nachkommastellen entfernen
  PRINT #1, "FIX ( 2.3) ="; L              ' Ausgabe auf LC-Display
  L = FIX (2.7)                           ' Nachkommastellen entfernen
  PRINT #1, "FIX ( 2.7) ="; L              ' Ausgabe auf LC-Display
END                                        ' Ende Task MAIN

```

Siehe auch: INT

FOR...NEXT

FOR...NEXT

FOR Zähler = Start TO Ende [STEP Step]

[Anweisungsblock].

NEXT

FOR...NEXT: Schleifen mit Laufvariable: Start, Stopp, Schrittweite

3

Instruktion: FOR...NEXT wiederholt einen Anweisungsblock solange, bis der Wert **Ende** überschritten, bzw. bei negativem Step unterschritten ist.

Die Schleife kann durch einen Sprung vorzeitig verlassen werden. Ebenso ist es auch zulässig die Schleifenvariable in der Schleife zu verändern. Dies kann dazu verwendet werden um bestimmte Werte zuüberspringen, zu wiederholen, die Schleifen-Anzahl zu verkürzen, zu verlängern oder einen vorzeitigen Exit zu erzielen.

Die NEXT Instruktion ist stets einem zuvor genannten FOR zugeordnet. Verschachtelte FOR-Loops bis zu 12 Ebenen sind zulässig.

Für **Zähler** sind folgende Datentypen zulässig:

Datentyp	Wertebereich
BYTE	0...255
WORD	0...65535
LONG	-2147483648...2147483647

Start ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Startwert der Laufvariablen im oben angeführten Wertebereiche.

Ende ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Endwert der Laufvariablen. Ist dieser erreicht oder überschritten, dann wird die FOR...NEXT Schleife beendet.

Step ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Schrittweite der FOR...NEXT Schleife. STEP 0 führt zu einer Endlosschleife falls überhaupt Schleifen ausgeführt werden. Negative Step Werte vermindern jeweils den aktuellen Wert der Schleifenvariablen. Der FOR-Loop ist dann bei Unterschreitung des Endwertes beendet. Der Wert von STEP ist nachträglich nicht mehr veränderbar.



Da der FOR-Loop am Schleifen-Anfang prüft ob es einen weiteren Loop gibt, können FOR-Loops bei entsprechenden Werten auch 0 Durchläufe ausführen (z.B. **Ende** ist kleiner als **Start** bei positiven **Step**). Die Datentypen für **Start**, **Ende** und **Step** dürfen auch gemischt verwendet werden, man beachte lediglich, daß negative Zahlen nur beim Typ LONG möglich sind. Die maximale Verschachtelungstiefe von FOR...NEXT-Schleifen und/oder LOOP...ENDLOOP-Schleifen beträgt 12 pro Task. Bei der Ermittlung der Verschachtelungstiefe zählen alle zur Zeit aktiven Loops einer Task mit, also auch die in aktiven Loops in z.Zt. laufenden Unterprogrammen.

Programmbeispiel:

3

```

' -----
' Name: FOR_NEXT.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG I,SUM                            ' SUM ist LONG-Variable
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  SUM = 0                                ' SUM vorbesetzen
  FOR I = 0 TO 25                          ' FOR-NEXT Schleife
    SUM = SUM + I                          ' Summe erhoehen
    PRINT #1, "<1>SUM (";I;" ) =";SUM      ' Ausgabe auf LC-Display
    WAIT DURATION 1000                    ' 1 Sek warten
  NEXT                                    ' naechster Wert
END                                        ' Ende Task MAIN

```

Die folgenden Zeilen demonstrieren eine Endlosschleife:

```

...
FOR I = 0 TO 0 STEP 0
...
NEXT

```

Siehe auch: LOOP, IF...THEN, WHILE

FRAME

FRAME

FRAME [Taskname] String

FRAME: Erzeugt einen String als Ausgaberahmen für PRINT_USING

Instruktion: Legt einen Standard-Ausgabertext für die PRINT_USING Instruktion fest. Dieser Standard-Text wird wie ein Rahmen jeder Ausgabe-Zeile hinterlegt. Die Festlegung gilt jeweils nur für die aktuelle Task bzw. für die mit **Taskname** angegebene Task. Jede Task kann daher ihre eigene FRAME Festlegung treffen. Wird **String** als Leerstring angegeben, so erscheint kein hinterlegter Text in der Ausgabe-Zeile. Die FRAME Instruktion erleichtert die Ausgabe übersichtlicher Tabellen mit hinterlegten Linien und Texten.

- Taskname** gibt an, für welche Task der Rahmen gesetzt wird. Jede Task kann eigene FRAME-Festlegungen treffen, ohne daß es eine Beeinflussung gibt oder zur Laufzeit ständig zwischen verschiedenen Einstellungen gewechselt werden muß. Wird Taskname nicht angegeben, wird die aktuelle Task angenommen.
- String** ist ein beliebiger String, der den hinterlegten Ausgabe-Rahmen definiert.

Die Instruktion PRINT_USING hinterlegt die Ausgabezeile mit dem zuvor durch FRAME festgelegten Rahmen. Alle weiteren Ausgaben in dieser Zeile werden anschliessend an die jeweiligen Positionen in dieser Zeile geschrieben. Ggf. werden dabei auch Textteile des FRAME-Strings überschrieben. Die FRAME-Zuweisung bleibt so lange aktiv, bis sie durch eine neue FRAME-Instruktion überschrieben bzw. mit Leerstring deaktiviert wird.

Mit der Instruktion TABS kann die Tabulation innerhalb einer PRINT_USING Instruktion individuell festgelegt werden.

	! Pos.	! Stück	! Artikel		! Preis	!
FRAME #1, “	!	!	!		!	!”
Ausgabestring			Modul X		149,00	

Programmbeispiel:

```

'-----
' Name: FRAME.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
FRAME " .... km, ###,## DM"            ' Frame-String definieren
USING "UD<4><0>  0.0.0.0.4UD<5><0>*"    0.0.0.3,2" ' Format-String
TABS "<1>L<10>L"                        ' Tabulatoren: Pos 1, 10
LOOP 10                                ' Loop-Schleife (10 mal)
PRINT #1,"<1><7>";                      ' Clear Screen + BEL
FOR N=1 TO 4                          ' 4 Zeilen:
  KM = 10 + RND(0) / 100              ' Anzahl km (Zufallswert)
  DM = KM * 18                       ' 1 km --> DM -,18
  PRINT USING #1,,KM,DM              ' Werte anzeigen
NEXT                                  '
WAIT DURATION 3000                   ' 3 Sek warten
ENDLOOP                              '
END                                  ' Ende Task MAIN

```

3

Siehe auch: TABS und USING

FREE_FIFO

FREE_FIFO

FREE_FIFO (Name)

FREE_FIFO: Anzahl freier Fifo-Einträge melden

Funktion: Liefert die Anzahl der augenblicklich im FIFO-Puffer **Name** freien Plätze für Datenelemente zurück. (FIFO=First-In-First-Out)

Funktionswert ist vom Typ LONG.

Programmbeispiel:

```
'-----
' Name: FREE_FIFO.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUF(512) OF WORD                  ' FIFO-Buffer definieren
  WORD I                                ' WORD-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 411                        ' 412 Werte in den
    PUT_FIFO BUF, I                      ' FIFO-Buffer schreiben
  NEXT
  PRINT #1, "FIFO free: ";FREE_FIFO(BUF) ' Ausgabe auf LC-Display
END                                       ' Ende Task MAIN
```

Siehe auch: LEN_FIFO, PUT_FIFO, GET_FIFO, PUTU_FIFO

GET

GET #Dev_Nr. [, #Sek_Adr][, #Func_Nr], Anzahl, Variable

GET: Unformatierte Daten-Eingabe von Device-Treiber

Instruktion: Get ist eine Eingabe-Instruktion, die stets mit einem DEVICE-Treiber zusammenarbeitet.

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc., können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

GET liest eine **Anzahl** Bytes von einem byteorientierten Gerät. **Dev_Nr** spezifiziert ein bestimmtes Gerät, das zu Beginn des Programms mit INSTALL_DEVICE unter dieser Device-Nummer definiert wurde. Als Ergebnis dieser Eingabe-Instruktion wird die **Variable** entsprechend gesetzt. GET liefert immer ein Ergebnis und wartet nicht auf das Vorhandensein tatsächlicher Eingabe-Daten. Liegen keine, oder zu wenige Eingabe-Daten vor, so wird die Ergebnis-Variable auf 0 bzw. Leerstring gesetzt. Es ist möglich sowohl Eingaben von fester Länge, als auch von variabler Länge zu verlangen.

#Dev_Nr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-

GET

Gerätes an, wie er zu Beginn des Programms mit IN-STALL_DEVICE definiert worden ist.

#Sek_Adr

Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.

#Func_Nr

Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.

Variable

ist vom Datentypen BYTE, WORD, LONG oder STRING.

Die Instruktion GET liefert abhängig von der angegebenen Anzahl und vom Datentyp der Variablen unterschiedliche Ergebnisse:

Wenn die Variable vom Datentyp String ist:

Anzahl = 0

Es werden so viele Bytes eingelesen, wie im Eingangsgerät vorhanden sind, jedoch höchstens so viele, wie in den String passen.

Anzahl = n

Feste Länge. Es werden genau n Bytes gelesen. Gibt es bislang noch nicht genug Zeichen im Puffer, ist das Resultat ein Leerstring. Falls **Anzahl** größer angegeben wird als die maximale Länge der **Variablen**, so wird automatisch die Variablenlänge als feste Länge verwendet. Die maximale Anzahl von gelesenen Bytes wird nur bestimmt durch: max. String-Länge (kann bis zu 65535 sein), Eingangspuffer Größe (abhängig vom Gerätetreiber: 16 Byte ... 4 MByte)

Bei numerischen Variablen:

Anzahl = 0 Variable Länge. Es werden so viele Bytes gelesen wie es der Größe des Variablen Typs entspricht (BYTE=1, WORD=2, LONG=4, REAL=8). Sind nicht genug Daten im Eingangspuffer, wird der Wert „0“ übergeben und kein Zeichen aus dem Eingangspuffer entnommen.

Anzahl = n Feste Länge. Damit können kleinere Datenmengen in größere Variablentypen eingelesen werden (z.B. 3 Byte in eine LONG-Variable einlesen). Unplausible Angaben werden entsprechend korrigiert.

Programmbeispiel:

```

'-----
' Name: GET.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE N                                ' BYTE-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "<1>Wait for key"              ' Ausgabe auf LC-Display
  FOR N = 0 TO 0 STEP 0                  ' Endlosschleife bis N=1(GET!)
    RELEASE TASK                          ' Rest der Task-Zeit freigeben
    GET #1, #0, #1, 1, N                  ' N=Zeichen im Tastatur-Buffer
  NEXT                                    ' Ende Endlosschleife
  PRINT #1, "Key pressed"                 ' Ausgabe auf LC-Display
END                                        ' Ende Task MAIN

```

GET_FIFO

GET_FIFO

GET_FIFO Name, Variable[, Fehlerbehandlung]

GET_FIFO: Hole Eintrag aus Fifo (wenn vorhanden)

Instruktion: Liest ein Element aus dem FIFO-Puffer **Name** in die **Variable**.
(FIFO=First-In-First-Out)

3

Variable kann vom Typ BYTE, WORD oder LONG sein. Eine Typwandlung findet automatisch statt.

Fehlerbehandlung wird versucht, aus einem leeren FIFO-Puffer zu lesen, so wird entsprechend der Fehlerbehandlung verzweigt. Schreibweise:

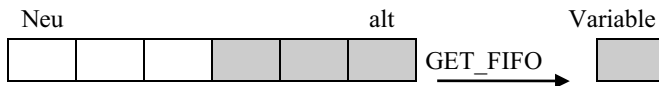
ON_ERROR_CALL Unterprogramm

ON_ERROR_GOTO Sprungmarke

Volle Schreibweise mit Fehlerbehandlung:

GET_FIFO Name, Variable, ON_ERROR_CALL Unterprogramm

Weitere Informationen zur Fehlerbehandlung finden Sie auf Seite 50ff.



Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```

'-----
' Name: GET_FIFO.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER(512) OF WORD                ' FIFO-Buffer definieren
  WORD I,J                               ' WORD-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 511                        ' 512 Werte in den
    PUT_FIFO BUFFER, 2*I                  ' FIFO-Buffer schreiben
  NEXT I                                  '
  FOR I = 0 TO 18                         ' Die ersten 19 Werte
    GET_FIFO BUFFER, J                   ' aus dem FIFO-Buffer lesen
    PRINT #1,"<1>Buffer";I;" ="; J      ' Wert auf LC-Display ausgeben
    WAIT_DURATION 1000                  ' 1 Sek warten
  NEXT I                                  '
END                                        ' Ende Task MAIN

```

3

Siehe auch: UNGET_FIFO, Funktion READ_FIFO

GOTO

GOTO

GOTO *Labelname*

GOTO: Unbedingte Programmverzweigung

Instruktion: Die Programmausführung wird an der mit **Labelname** markierten Stelle weitergeführt.

3

Sprünge in eine andere Task, in eine FOR...NEXT-Schleife oder in eine LOOP...ENDLOOP-Schleife sind nicht erlaubt

Es ist jedoch stets möglich aus einer Schleife herauszuspringen.

Ein sparsamer Gebrauch von GOTO-Instruktionen führt zu besser strukturierten Programmen. Andere Anweisungen wie LOOP, FOR...NEXT oder WHILE eignen sich eventuell besser zur Programmablaufkontrolle.

Programmbeispiel:

```
'-----
' Name: GOTO.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE X                                ' BYTE-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

START:                                  ' Sprungmarke setzen
  FOR X=0 TO 9                          ' Zaechlschleife fuer X
    PRINT #1, "<1>X ="; X                ' Ausgabe des Zaehlers
    WAIT DURATION 1000                  ' 1 Sek warten
    IF X=6 THEN                         ' Falls X=6:
      GOTO START                       ' Ruecksprung auf Marke START
    ENDIF
  NEXT                                  ' naechster Wert
END                                    ' Ende Task MAIN
```


HREAL

HREAL (Real)

HREAL: High 32-Bits of REAL-Wert

Funktion: Liefert als Ergebnis in einer LONG-Zahl die höherwertigen 4 Byte der REAL-Zahl.

Real ist vom Typ REAL.

Funktionswert ist vom Typ LONG.

Mit den Funktionen LREAL und HREAL läßt sich eine 8-Byte-REAL-Zahl in zwei 4-Byte-LONG-Zahlen zerlegen. Die Funktion LLTOR fügt die beiden 4-Byte-Stücke wieder zusammen.

Diese Funktionen ermöglichen das Weiterverarbeiten von REAL-Zahlen in byte- oder bit-orientierter Form, was sonst mit REAL-Zahlen nicht möglich wäre. Nach der Aufspaltung in zwei LONG-Zahlen können nun auch REAL-Zahlen mit der Funktion NTOSS in Strukturen oder Records eingebaut werden.

Programmbeispiel:

```

' -----
' Name: HREAL.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  REAL R1,R2                            ' REAL-Variablen deklarieren
  LONG Hi,Lo                            ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  R1 = 12345.6789                        ' R1 vorbesetzen
  Lo = LREAL (R1)                        ' Low 32 Bit von R1 --> Lo
  Hi = HREAL (R1)                        ' High 32 Bit von R1 --> Hi
  R2 = LLTOR (Lo,Hi)                     ' Erzeuge R2 aus Lo und Hi
  USING "EF<0><1> <8>V0.0.0.0.0.0.1,8.0.0.0.0.0.0"
  PRINT USING #1, "<1>R1="; R1           ' zeige R1 im Original
  PRINT #1, "Lo="; Lo                    ' zeige Low 32 Bit von R1
  PRINT #1, "Hi="; Hi                    ' zeige High 32 Bit von R1
  PRINT USING #1, "R2="; R2              ' zeige R2 (Lo + Hi --> REAL)
END                                      ' Ende Task MAIN

```

Siehe auch: LREAL, LLTOR

I2C_ACK_POLL

I2C_ACK_POLL

I2C_ACK_POLL (*Chip*)

I2C_ACK_POLL: ACK-Polling auf I2C-Bus ob Device bereit ist

Funktion: Pollt ein I²C-Device mit der Adresse **Chip** bis dieses mit einem Acknowledge-Puls antwortet.

3

Chip

ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG und gibt die Adresse des I²C-Chips an. Das Schreib-Lese-Bit wird von den I²C-Funktionen automatisch richtig gesetzt.

I2C_ACK_POLL schreibt die Adresse des Chips wiederholt auf den Bus, um zum Beispiel bei EEPROMs festzustellen, wann der Programmierzyklus beendet ist.

Programmbeispiel:

```

'-----
' Name: I2C_ACK_POLL.TIG
'-----
#define M24LC16 0A0h                ' Adresse EEPROM 2416

TASK MAIN                          ' Beginn Task MAIN
  STRING D$
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  ' I2C_SETUP (pin, din, pout, dout, clout, spcnt)
  I2C_SETUP ( 7, 0, 7, 1, 2, 0)

  D$ = "<00><01><02>"
  ' I2C_WRITE (i2c_sladr, i2c_adr, nob, var$) ' schreibe
  I2C_WRITE ( M24LC16, 1, 3, D$) ' schreibe Daten

  FOR I = 0 TO 100                  ' bei 100 Time-out
    A = I2C_ACK_POLL ( M24LC16 ) ' polle EEPROM
    IF A = 0 THEN
      GOTO READY
    ENDIF
  NEXT
  PRINT #1, "Time out"
  GOTO FIN
READY:
  PRINT #1, "Ready: ";I;"loops"
FIN:
END                                ' Ende Task MAIN

```

3

Siehe auch: I2C_SETUP, I2C_WRITE, I2C_READ\$, I2C_RESULT sowie Beispiele im Verzeichnis APPLICATIONS

I2C_READ\$

I2C_READ\$

I2C_READ\$ (*Chip, Adresse, Anzahl, Variable*)

I2C_READ\$: Liest Bytes von I2C-Bus

Funktion: Liest von einem I²C-Device mit der Adresse **Chip** ab der angegebenen **Adresse** eine **Anzahl** Bytes in die **Variable** ein.

3

Chip	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG und gibt die Adresse des I ² C-Chips an. Das Schreib-Lese-Bit wird von den I ² C-Funktionen automatisch richtig gesetzt.
Adresse	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG und gibt die Adresse oder Registernummer in dem I ² C-Chip an, ab der gelesen wird.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 1 bis 31 und gibt an, wieviele Bytes gelesen werden.
Variable	ist eine Variable vom Datentyp STRING und erhält die gelesenen Bytes.

I2C_READ\$ schreibt zunächst die Adresse des Chips zusammen mit der zu lesenden auf den I²C-Bus. Dann wird das Lesekommando erzeugt und soviele Bytes gelesen, wie durch Anzahl vorgegeben ist. Der Status des Lesevorgangs kann danach mit der Funktion I2C_RESULT abgefragt werden.

Die Anzahl der zu lesenden Bytes ist auf 31 beschränkt.

Programmbeispiel:

```

'-----
' Name: I2C_READ$.TIG
'-----
#define PCF8583 0A0h                ' Adresse der Uhr PCF8583
                                     ' ab Adr. 1: Sek., Min., Std.
TASK MAIN                          ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

' I2C_SETUP (pin, din, pout, dout, clout, spcnt)
I2C_SETUP ( 7, 0, 7, 1, 2, 0)

USING "UH<2><2> 0.0.0.0.2"          ' Zahlen als HEX ausgeben
FOR I = 0 TO 0 STEP 0
'   R$ = I2C_READS$ (i2c_sladr, i2c_adr, nob) ' Lese Byte(s)
'   R$ = I2C_READS$ ( PCF8583, 2, 3) ' Lese SMH
PRINT_USING #1,"Rd=";I2C_RESULT ()

                                     ' tausche Sek./Std.
R$ = RIGHT$ ( R$, 1 ) + MID$ ( R$, 1, 1 ) + LEFT$ ( R$, 1 )
FOR J = 0 TO LEN( R$ ) - 1
PRINT_USING #1,ASC( MID$( R$, J, 1 ) );" ";
NEXT
NEXT
END                                ' Ende Task MAIN

```

3

Siehe auch: I2C_SETUP, I2C_WRITE, I2C_ACK_POLL, I2C_RESULT sowie Beispiele im Verzeichnis APPLICATIONS

I2C_RESULT

I2C_RESULT

I2C_RESULT ()

I2C_RESULT: Liest Status der letzten I2C-Bus Operation

Funktion: Liefert numerisch den Status der zuletzt ausgeführten I²C-Funktion.

Nr	Symbol	Beschreibung
129	I2C_RDOK	Lesen o.k.
241	I2C_RDERR	Lesen fehlerhaft
130	I2C_WROK	Schreiben o.k.
242	I2C_WRERR	Schreiben fehlerhaft
132	I2C_SETOK	Setup o.k.
244	I2C_SETERR	Setup fehlerhaft
250	I2C_PARERR	Parameterfehler
255	E2C_ERR	I ² C-Fehler (allgemein)

Programmbeispiel:

```

'-----
' Name: I2C_RESULT.TIG
'-----
#define PCF8583 0A0h                ' Adresse der Uhr PCF8583
                                     ' ab Adr. 1: Sek., Min., Std.
TASK MAIN                          ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

' I2C_SETUP (pin, din, pout, dout, clout, spcnt)
I2C_SETUP ( 7, 0, 7, 1, 2, 0)

USING "UH<2><2> 0.0.0.0.2"          ' Zahlen als HEX ausgeben
FOR I = 0 TO 0 STEP 0
'   R$ = I2C_READS$ ( i2c_sladr, i2c_adr, nob ) ' Lese Byte(s)
'   R$ = I2C_READS$ ( PCF8583, 2, 3 ) ' Lese SMH
PRINT_USING #1,"Rd=";I2C_RESULT ()

                                     ' tausche Sek./Std.
R$ = RIGHT$ ( R$, 1 ) + MID$ ( R$, 1, 1 ) + LEFT$ ( R$, 1 )
FOR J = 0 TO LEN( R$ ) - 1
'   PRINT_USING #1,ASC( MID$( R$, J, 1 ) );" ";
NEXT J
NEXT I
END                                ' Ende Task MAIN

```

3

Siehe auch: I2C_SETUP, I2C_READ\$, I2C_WRITE, I2C_ACK_POLL sowie Beispiele im Verzeichnis APPLICATIONS

I2C_SETUP

I2C_SETUP

I2C_SETUP (iP_in, D_in, iP_out, D_out, C_out, Delay)

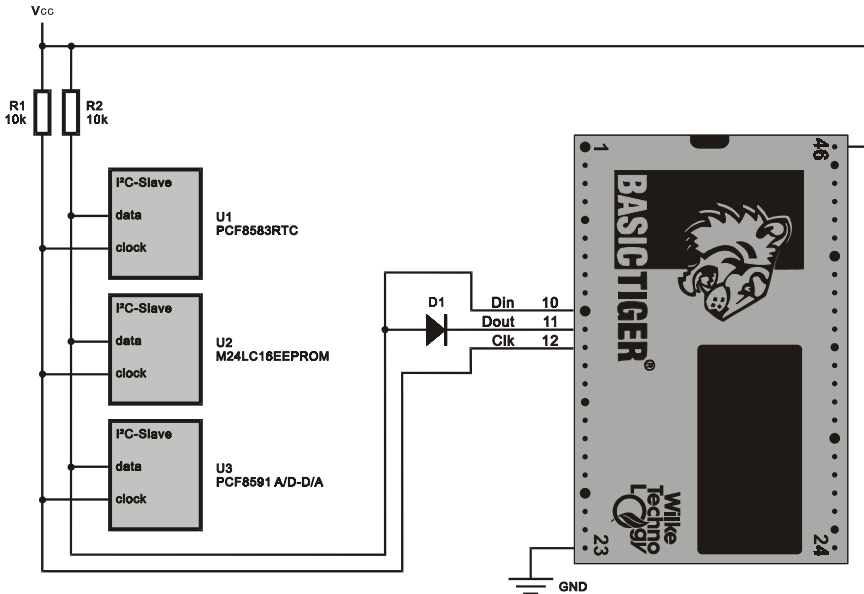
I2C_SETUP: I2C-Bus Einstellungen setzen

Funktion: Initialisiert die I²C-Pins auf einem oder zwei Ports.

iP_in	ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt die Adresse des internen Ports für Input an.
D_in	ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt die Nummer des Pins für Daten-Input an.
iP_out	ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt die Adresse des internen Ports für Output an.
D_out	ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt die Nummer des Pins für Daten-Output an.
C_out	ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt die Nummer des Pins für Clock-Output an.
Delay	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt einen Verzögerungswert an, der die Übertragungsgeschwindigkeit herabsetzt. 0: ca. 200kHz 1: ca. 100kHz 2...10: in Schritten ca. 80kHz...30kHz

I2C_SETUP initialisiert die internen Portpins, die mit der Instruktion I2C_WRITE und der Funktion I2C_READ\$ als I²C-Bus verwendet werden. Außerdem wird ein Verzögerungsparameter gesetzt, der die Geschwindigkeit der Übertragung beeinflusst.

Die Hardware eines I²C-Busses an BASIC-Tiger sieht z.B. so aus:



Der Data-out-Pin wird mit Data_in über eine Diode verbunden. Auf dem Bus ist für die Clock-Leitung und für die Datenleitung je einmal ein Pull-up-Widerstand nötig. Der Wert der Widerstände richtet sich nach den physikalischen Gegebenheiten des Busses. In kleinen Aufbauten mit ein paar Chips auf einer Platine hat sich der Wert von 10k bewährt.

I2C_SETUP

Programmbeispiel:

```
'-----
' Name: I2C_SETUP.TIG
'-----
#define PCF8583 0A0h                                ' Adresse der Uhr PCF8583
                                                    ' Uhr hat ab Adresse 10h RAM
TASK MAIN                                           ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

' I2C_SETUP (pin, din, pout, dout, clout, spcnt)
I2C_SETUP ( 7, 0, 7, 1, 2, 0)

' R$ = I2C_READS$ (i2c_sladr, i2c_adr, nob)      ' Lese Byte(s)
R$ = I2C_READS$ ( PCF8583, 10h, 10)           ' Lese 10 RAM-Bytes
PRINT #1,"Rd=";I2C_RESULT ()
PRINT #1,"R$=";R$
END                                                    ' Ende Task MAIN
```

Siehe auch: I2C_READ\$, I2C_WRITE, I2C_ACK_POLL, I2C_RESULT sowie Beispiele im Verzeichnis APPLICATIONS

I2C_WRITE

I2C_WRITE (*Chip, Adresse, Anzahl, String*)

I2C_WRITE: Schreibt Datenstring auf I2C-Bus

Funktion: Schreibt auf ein I²C-Device mit der Adresse **Chip** ab der angegebenen **Adresse** eine **Anzahl** Bytes aus dem **String**.

Chip	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG und gibt die Adresse des I ² C-Chips an. Das Schreib-Lese-Bit wird von den I ² C-Funktionen automatisch richtig gesetzt.
Adresse	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG und gibt die Adresse oder Registernummer in dem I ² C-Chip an, ab der geschrieben wird.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 1 bis 31 und gibt an, wieviele Bytes geschrieben werden.
Variable	ist eine Variable vom Datentyp STRING und erhält die zu schreibenden Bytes.

3

I2C_WRITE schreibt zunächst die Adresse des Chips zusammen mit der Adresse bzw. Registernummer des Chips auf den I²C-Bus. Dann werden soviele Bytes geschrieben, wie durch die Anzahl vorgegeben ist. Der Status des Schreibvorgangs kann danach mit der Funktion I2C_RESULT abgefragt werden.

Anmerkung: Das IC **PCF8574** von Philips besitzt keine interne Adresse. Anstelle der internen Adresse steht direkt der Wert, der geschrieben wird. Mit einem Trick läßt sich dieser Chip trotzdem anwenden: Es wird grundsätzlich nur die Lesefunktion I2C_READ\$ eingesetzt. Diese schreibt den Wert, der als 'interne Adresse' angegeben ist und liest zugleich. Es wird also immer Lesen und Schreiben zugleich in einer Funktion ausgeführt.

Die Anzahl der zu schreibenden Bytes ist auf 31 beschränkt.

I2C_WRITE

Programmbeispiel:

```
'-----
' Name: I2C_WRITE.TIG
'-----
#define PCF8583 0A0h                ' Adresse der Uhr PCF8583
                                     ' ab Adr. 1: Sek., Min., Std.
TASK MAIN                          ' Beginn Task MAIN
  STRING T$
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  ' I2C_SETUP (pin, din, pout, dout, clout, spcnt)
  I2C_SETUP ( 7, 0, 7, 1, 2, 0)

  T$ = "<00><59><11>"
  ' I2C_WRITE (i2c_sladr, i2c_adr, nob, var$) ' schreibe
  I2C_WRITE ( PCF8583, 1, 3, T$) ' schreibe Zeit
END                                     ' Ende Task MAIN
```

Siehe auch: I2C_SETUP, I2C_READ\$, I2C_ACK_POLL, I2C_RESULT sowie Beispiele im Verzeichnis APPLICATIONS

IF...THEN**IF Bedingung THEN****Anweisungsblock****ELSE****Anweisungsblock****ENDIF****IF..THEN..ELSE..ENDIF: Bedingte Programmausführung, ggf. mit Alternative**

Instruktion: Führt eine Anweisung oder einen Anweisungsblock in Abhängigkeit von der angegebenen Bedingung aus.

Die Bedingung besteht aus mindestens einer Vergleichs-Operation, die als wahr oder falsch ausgewertet werden kann.

Mehrere solcher Bedingungen können durch AND, OR, XOR oder NOT miteinander verknüpft werden. (Siehe Operatoren Seite 34)

Vergleichsoperatoren	
=	gleich
<>	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich

Im Vergleich zwischen einer Konstruktion mit IF...ELSE...ENDIF und SWITCH...CASE...ENDSWITCH ist SWITCH...CASE wesentlich schneller, da hier die Verzweigung abhängig von einer Konstanten erfolgt, wo bei IF...ELSE verknüpfte Ausdrücke als Bedingung möglich sind. Einen weiteren Geschwindigkeitsvorteil bietet die indizierte Verzweigung über eine Tabelle mit der Instruktion SWITCH...CASE.

IF...THEN

Instruktion	Vorteil
IF...THEN...ELSE	Ausdrücke und verknüpfte Bedingungen möglich
SWITCH...CASE	beliebige Konstante hinter CASE, schneller als IF...THEN...ELSE
SWITCHL...CASE	höchste Verzweigungsgeschwindigkeit, da über indizierte Sprungtabelle

3

Programmbeispiel:

```
'-----  
' Name: IF_THEN.TIG  
'-----  
' Erzeugt 10 Zufallszahlen zwischen 0 und 255 und gibt als Text  
' aus, ob diese groesser als 127 oder kleiner als 128 sind.  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
  LONG I                                ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  RANDOMIZE                              ' Startwert ermitteln  
  LOOP 10                                ' 10 Durchlaeufer  
    I = RND(0) BITAND 0000000011111111b ' Zufallszahlen von 0..255  
    PRINT #1, "<1>"; I; " is"           ' Zufallszahl ausgeben  
    IF I > 127 THEN                      ' Falls I > 127:  
      PRINT #1, "greater than 127"      ' Ausgabe "groesser"  
    ELSE                                  ' Sonst:  
      PRINT #1, "less than 128"         ' Ausgabe "kleiner"  
    ENDIF  
    WAIT DURATION 2000                  ' 2 Sek warten  
  ENDLOOP  
END                                      ' Ende Task MAIN
```

Siehe auch: SWITCH...CASE, SWITCHL...CASE

IMASK

IMASK (BitNr)

IMASK: Gibt inverse Maske zurück, ein „0“ Bit

Funktion: Liefert eine Bitmaske zurück, bei der alle Bits gleich 1 sind und nur das Bit an der Position **BitNr** gleich 0 ist. Das niederwertigste Bit ist Bitnummer 0.

BitNr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt das auf 0 gesetzte Bit an. Das niederwertigste Bit ist Bitnummer 0.

Funktionswert ist vom Typ BYTE, WORD oder LONG und enthält das gesetzte Bit. Alle anderen Bits sind 0.

Mit der IMASK.-Funktion kann auf einfache Weise eine Bitmaske erzeugt werden, die in Ausdrücken in Zusammenhang mit AND, OR oder den Instruktionen IN und OUT Verwendung findet.

IMASK (1)

liefert als Funktionswert in einer BYTE-Variablen:

1	1	1	1	1	1	0	1

Programmbeispiel:

```

'-----
' Name: IMASK.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG L, Value                          ' LONG-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  Value = 1234567890                    ' Value vorbesetzen
  FOR L = 0 TO 31                        ' Werte von 0 bis 31
    PRINT #1, "<1>Bit"; L; " is"        ' Ausgabe auf LC-Display
    IF (Value AND IMASK(L)) = Value THEN ' Falls Bit L nicht gesetzt
      PRINT #1, "not ";                ' Ausgabe auf LC-Display
    ENDIF
    PRINT #1, "set."                    ' Ausgabe auf LC-Display
    WAIT DURATION 2000                  ' 2 Sek warten
  NEXT                                  ' naechstes Bit pruefen
END                                     ' Ende Task MAIN

```

Siehe auch: MASK

IN

IN

IN Log_Portadr., Variable

IN: Liest Bits von I/O-Port ein

Instruktion: Liest den momentan anliegenden 8-Bit-Wert von dem internen oder externen Port mit **Log_Portadr.** in **Variable** ein.

Log_Portadr. ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Variable ist vom Datentypen BYTE, WORD oder LONG.

Programmbeispiel:

```
'-----
' Name: IN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE Value,Old                         ' BYTE-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  DIR_PORT 8,255
  Old = 0                                ' Port 8 als Eingang setzen
  Value = 1                              ' Variablen initialisieren
  FOR X=0 TO 0 STEP 0                    ' Endlosschleife
    IN 8,Value                            ' Port einlesen
    IF Value <> Old THEN                  ' Falls Wert geaendert,
      PRINT #1, "<1>Value ="; Value      ' Ausgabe auf LC-Display
      Old = Value                       ' alten Wert sichern
    ENDIF
  NEXT                                    ' Ende Endlosschleife
END                                        ' Ende Task MAIN
```

Sowohl das erweiterte Portsystem als auch Device-Treiber können Portbits gegen Zugriff schützen, so daß diese dann nicht mehr durch IN- und OUT-Instruktionen angesprochen werden können. Siehe auch 'Die Softwareseite der erweiterten I/O-Ports' im Hardware-Handbuch.



Siehe auch: LL_IPORT_IN, XIN, XBUS_INR\$

INDEX

INDEX (*Liste*, *Suchelement*)

INDEX: Sucht Zeichenfolge in einem String, liefert Index

Funktion: INDEX gibt den Index zurück, unter dem das **Suchelement** sich in der **Liste** befindet, oder -1, wenn das **Suchelement** nicht vorhanden ist.

Liste	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält als erstes Zeichen das Trennzeichen zwischen den Stringelementen. Es folgen die Stringelemente solange, bis das Trennzeichen zweimal hintereinander aufgeführt ist.
Suchelement	ist vom Datentyp STRING und stellt den unter den Stringelementen der Liste zu suchenden String dar.
Funktionswert	ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Mit der Funktion INDEX wird festgestellt, an welcher Stelle in der Liste sich der gesuchte String (Suchelement) befindet. Die erste Position ist die Position 0. Befindet sich der gesuchte String nicht in der Liste, liefert INDEX als Ergebnis '-1' zurück. Die Zuweisungsvariable sollte daher eine (vorzeichenbehaftete) LONG-Zahl sein.

Die Liste ist als String aufgebaut, der als erstes Zeichen das gewünschte Trennzeichen für die Listenelemente enthält, welches sich am Ende des Strings in zweifacher Form wiederholt. Die Elemente sind nicht in der Länge begrenzt, jedoch können sie nicht das Trennzeichen enthalten.

Beispiele für Strings, die als Liste aufgebaut sind:

```

B$ = ",John,Mary,Harry,Mathilde,Nobody,, "
X = INDEX ( B$, "Mathilde" )           ` liefert 3 fuer X
C$ = "!0,00DM!1,25DM!2,50DM!3,75DM!5,00DM!6,25DM!!"
BIN$ = ":0000:0001:0010:0011:0100:0101:0110:0111:&
1000:1001:1010:1011:1100:1101:1110:1111:"

```

Anwendungen: Menüselektion, Konverter, Daten-Kompression/-Expansion

INDEX

Programmbeispiel:

```
' -----
' Name: INDEX.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING L$,S$                          ' STRING-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  L$ = ":This:is:one:Test:~:"          ' L$ vorbesetzen
  S$ = "one"                          ' S$ vorbesetzen
  PRINT #1, "<1>"; L$                    ' Ausgabe der Liste
  PRINT #1, S$                        ' Ausgabe des Suchelements
  PRINT #1, "Index:~"; INDEX(L$,S$)    ' Ausgabe des Index
END                                    ' Ende Task MAIN
```

Siehe auch: SELECT\$, INDEX_1D, INDEX_2D

INDEX_1D**INDEX_1D (X1, Ranges\$)****INDEX_1D (X1, Ranges\$, Act, Hyst)**

INDEX_1D: 1D-Wertklassen, Digi-Poti, Widerstands-Tastaturen, Toleranzgruppen
(nur in Projektmodell 'PM_FULL')

Funktion: Prüft in welchem Wertebereich der Wert **X1** liegt und liefert eine passende Indexnummer. **Ranges\$** gibt die Wertebereiche vor.

3

- X1** ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und stellt den aktuellen Eingabe oder Messwert dar. WORD-Variable werden entsprechend vorzeichenbehaftet gesehen.
Wertebereich von 0...+16383(7FFFh)
- Ranges\$** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält eine Tabelle der Wertebereiche als WORD-Zahlen. Die Werte in der Tabelle müssen in aufsteigender Folge vorliegen.
Wertebereich: 0...+16383(7FFFh)
- Act** ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und stellt den aktuellen ‚eingeras-teten‘ Wert dar (in der Regel vorheriges X1). Act wird übergeben, wenn die Hysteresis-Funktion benötigt wird. WORD-Variable werden entsprechend vorzeichenbehaftet gesehen.
Wertebereich von 0...+16383(7FFFh).
- Hyst** ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und den Hysteresiswert an. Hyst wird übergeben, wenn die Hysteresis-Funktion benötigt wird.
- Funktionswert** ist vom Typ LONG und gibt den Index des Tabelleneintrags zurück, in dessen Wertebereich X1 liegt. Die Zählung beginnt mit 0. Eine Typwandlung bei der Zuweisung findet automatisch statt.
- 0...n: Bereich wurde getroffen
-1: X1 lag außerhalb des zulässigen Wertebereichs von - 0...+16383

INDEX_1D

Vergleich zwischen INDEX_1D und INDEX_2D

INDEX_1D arbeitet ähnlich wie Funktion INDEX_2D. Während man dort in der 2-dimensionalen Ebene bestimmte Felder (Boxen) definieren kann und hieraus einen Index bzw. eine Box-Nummer erhalten kann, so wird entsprechendes bei INDEX_1D in der 1-dimensionalen Welt bewerkstelligt. INDEX_1D verwendet ebenfalls über einen Definitions-String, der die Lage von „Boxen“ definiert und ein skalarer Eingangswert, wird nun auf eine der existierenden Boxen verteilt.

3

INDEX_1D wandelt quasi-kontinuierliche Werte (z.B. vom A/D-Wandler) in diskret abgestufte Werte mit größerer Stufung.

Der erste Bereich beginnt immer bei 0 und der letzte endet mit 7FFFh. Beispiel: die Tabelle enthält die Werte 3, 5, 7 (low Byte zuerst):

I = INDEX_1D (9, "03 00 06 00 07 00 09 00"%)

,I' erhält den Wert 3 die Bereiche gelten wie folgt:

- 0...3 = Bereich 0
- 4...6 = Bereich 1
- 7...7 = Bereich 2
- 8...9 = Bereich 3
- 8...16767 = Bereich 4

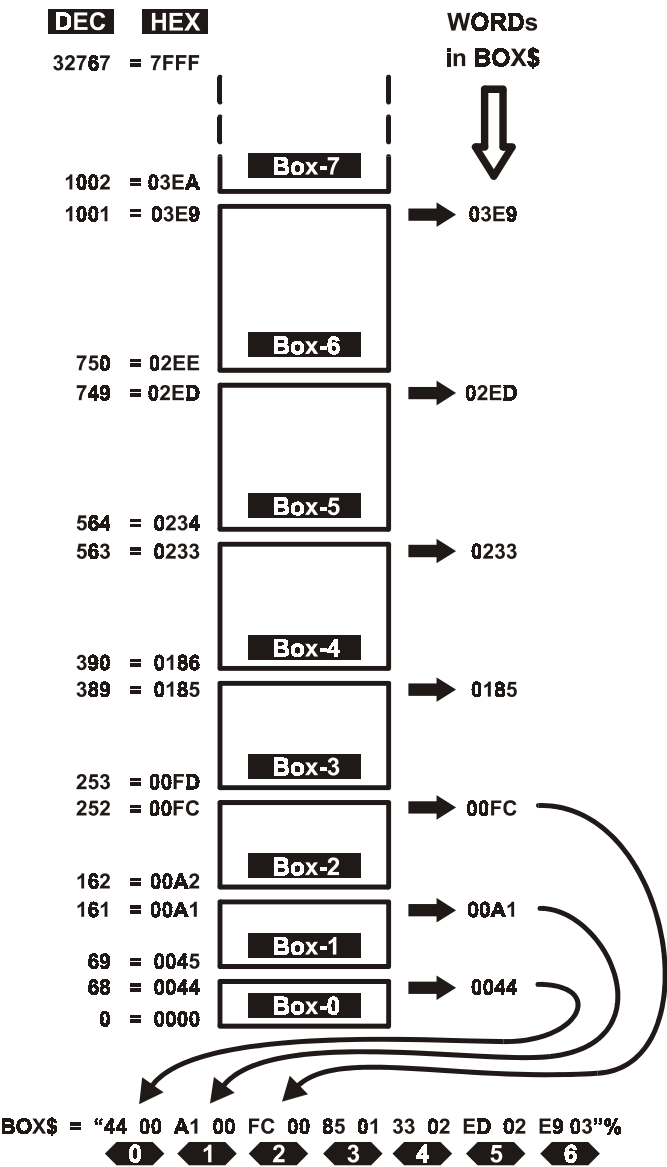
Das folgende kleine Beispiel zeigt die Funktionswerte auf einem kleinen Text-LCD und ist daher überschaubar und leicht nachvollziehbar.

Programmbeispiel:

```
'-----  
' Name: INDEX_1D1.TIG  
'-----  
  
TASK Main  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  
  RANGES$ = "03 00 06 00 07 00 09 00"% ' 5 Bereiche letzter 9...7FFFh  
  
  FOR V = -2 TO 12  
    ' teste einige Werte  
    PRINT #1, "V:";V;" INDEX_1D:";INDEX_1D ( V, RANGES$ )  
    WAIT_DURATION 500  
  NEXT  
  PRINT #1, "1FFFF";" INDEX_1D:";INDEX_1D ( 1FFFFh, RANGES$ )  
END
```

Siehe auch: INDEX_2D, DEBOUNCE, Applikation .TIG

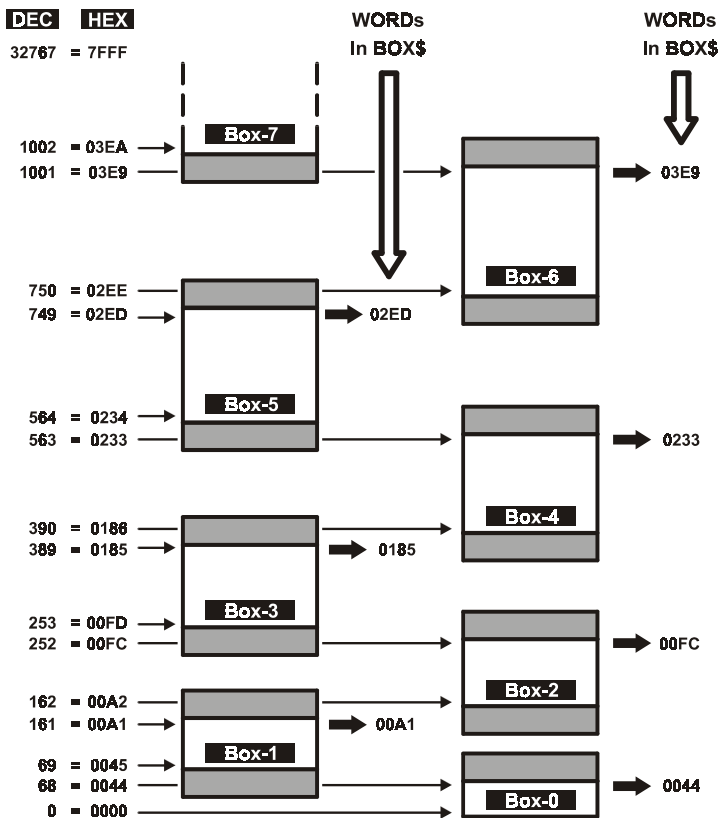
Eine Mögliche Zuordnung von Eingangs-Werten in 8 unterschiedlich große Boxen veranschaulicht folgende Abbildung:



INDEX_1D Boxes

Hysterese

Manchmal soll ein einmal ‚betretener‘ Bereich nicht so leicht durch einzelne Ausreißer der analogen Meßwerte verlassen werden. Mit Hilfe einer eingebauten Hysterese läßt sich dies erreichen. Ist einmal ein Indexwert einer ‚Box‘ gültig, dann muß der Wert um einen bestimmten Betrag außerhalb liegen, um die ‚Box‘ zu verlassen. Das Gleiche gilt jedoch auch für das Betreten einer ‚Box‘. Wenn die Werte nur am Rande im Bereich liegen und von einer benachbarten ‚Box‘ dorthin gewandert sind, dann bleibt auf Grund der Hysterese noch weiterhin die benachbarte ‚Box‘ gültig. Die Abbildung zeigt die Hysteresebereiche als graue Übergänge zwischen den Boxen. Mechanisch entspricht dies einer Rastung, eine gültige Box ist eingerastet und kann nur durch eine etwas größere Kraft verlassen werden.



BOX\$ = "44 00 A1 00 FC 00 85 01 33 02 ED 02 E9 03"%"
0 1 2 3 4 5 6

INDEX_1D Boxes + Hysteresis

INDEX_1D

Das folgende kleine Beispiel zeigt auf einem kleinen Text-LCD die Werte an, die INDEX_1D liefert. Trotz überschreiten der Boxgrenze von 8 springt der Indexwert erst bei 11 von 0 auf 1 um.

Programmbeispiel:

```
' -----
' Name: INDEX_1D2.TIG
' zeigt Hysteresefunktion
' -----
TASK Main
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

'          <--8--><-16--><-24--><-32-->
RANGES$ = "08 00 10 00 18 00 20 00"% ' 5 Bereiche letzter 20h...7FFFh

W = 0
FOR EVER = 0 TO 0 STEP 0          ' Endlosschleife
  FOR V = 8 TO 11                  ' von unten kommend
    W = INDEX_1D ( V, RANGES$, W, 2 )
    PRINT #1, " V:";V;" W:";W      ' 8 ist Ende Bereich 0
    WAIT_DURATION 500              ' aber springt bei 11 um
    NEXT                          ' steigend

    WAIT_DURATION 2000
    PRINT #1, "<1>";                ' loesche LCD

    FOR V = 9 TO 6 STEP -1          ' von oben kommend
      W = INDEX_1D ( V, RANGES$, W, 2 )
      PRINT #1, " V:";V;" W:";W    ' 8 ist Beginn Bereich 0
      WAIT_DURATION 500            ' aber springt bei 6 um
      NEXT                        ' fallend

      WAIT_DURATION 2000
      PRINT #1, "<1>";              ' loesche LCD

      V = 18
      W = INDEX_1D ( V, RANGES$, W, 2 )
      PRINT #1, " V:";V;" W:";W    ' ab 17 beginnt Bereich 2
                                   ' springt sofort um

      WAIT_DURATION 2000
      PRINT #1, "<1>";              ' loesche LCD

    NEXT                          ' Endlosschleife
END
```

Siehe auch: INDEX_2D, DEBOUNCE

Anwendungen für INDEX_1D

Die Anwendungen stellen sich vielfältig und sehr unterschiedlich dar.

Häufig angewendet werden die Boxen von INDEX_1D in jeder Art von Prüfungen, wie bei Endkontrollen in industriellen Produktionen, wo es um ‚gut‘ oder ‚schlecht‘ bzw. um verschiedene Toleranzfelder und Genauigkeitsklassen geht. Zum Beispiel soll der Durchmesser von Gemüsezwiebeln, die Reaktionszeit eines Call-Centers oder die Gewichtszunahme von Freilandhühnern in einer begrenzten Anzahl von Kategorien erfasst werden.

Eine andere Form der Anwendung für INDEX_1D liegt in der gezielten Umsetzung einer linearen Bewegung oder Rotation, in abgestufte Ausgangswerte. Zum Beispiel kann man in einem Oszilloscope Einstellungen wie Eingangs-Empfindlichkeit, Zeitbasis, .etc. mit klassischen Stufenschaltern oder Drehwinkelgebern (Encoder) vornehmen. Eine kostengünstigere Alternative hierzu bietet INDEX_1D: auch solche gestuften Einstellungen könnte man mit einem preiswerten Dreh-Potentiometer vornehmen, dessen kontinuierliche Ausgangswerte über INDEX_1D in feste Stufen umgesetzt werden. Ebenso lassen sich ein softwaregesteuerter Rasten-Click, Menü-Auswahlen und zum Beispiel numerische Eingaben hiermit realisieren.

Die Handhabung einer solchen Potentiometer-gesteuerten „digital“-Eingabe ähnelt der Eingabe mit Drehwinkelgebern, mit dem Unterschied eines limitierten Drehbereichs bei Potentiometern.

Tastatur mit INDEX_1D

Schließlich eröffnet INDEX_1D auch interessante Möglichkeiten wenn es darum geht kleine Tastaturen oder mehrkanalige Abtaster mit minimalem Aufwand zu realisieren. Tasten oder Tastaturen lassen sich auf unterschiedliche Weise in ein Tiger-System integrieren:

- Komplette MFII-Tastaturen über seriellen Anschluß mit Device-Treiber: MF2-xxx.TDD.
- Tasten-Matrix bis 128 Tasten mit Shift, CTRL und Repeat über Device Treiber LCD1.TDD.
- Einzeltasten an digitalen I/O-Pins abfragen (interne I/O oder Erweiterungs-I/O).
- Mit einem einzigen Analogeingang eine Tastatur mit INDEX_1D.

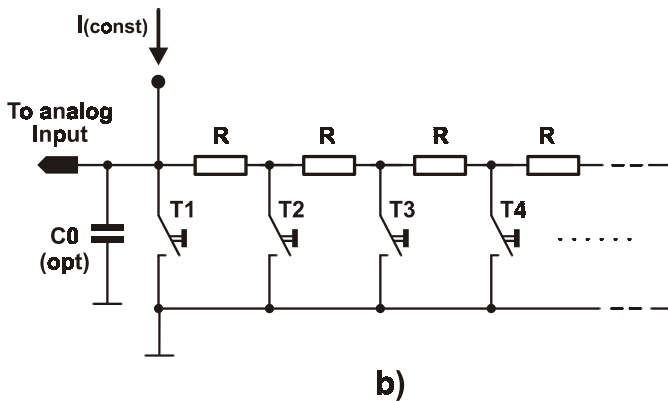
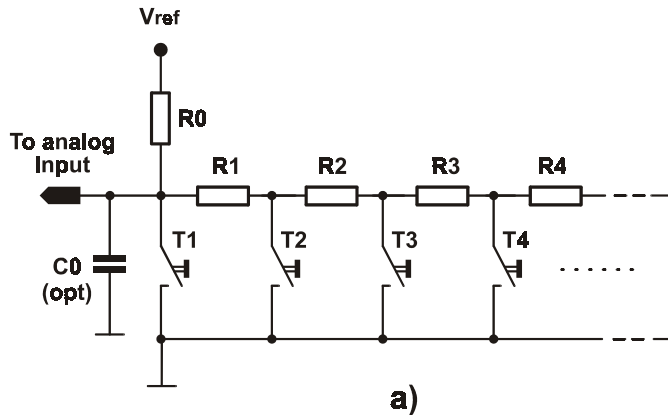
Mit einem einzigen Analogeingang kann eine Tastatur mit INDEX_1D realisiert werden. Die Tasten schalten einen Spannungsteiler, der aus einer Widerstandskette besteht. Jede Taste erzeugt einen anderen analogen Messwert. Da der Messwert naturgemäß schwankt, erzeugt eine Taste keinen konstanten Wert, sondern Werte, die sich in einem bestimmten Bereich bewegen. INDEX_1D weist einer Taste einen Indexwert zu, sofern die gemessenen Analogwerte innerhalb eines Wertebereichs liegen. INDEX_1D erhält im String Ranges\$ die Definitionen der Wertebereiche, indem jeweils die oberen Grenzwerte als WORD-Wert angegeben werden. Die oberen

INDEX_1D

Grenzwerte gehören noch zu dem angegebenen Wertebereich, der folgende Bereich beginnt bei dem nächstgrößeren Wert.

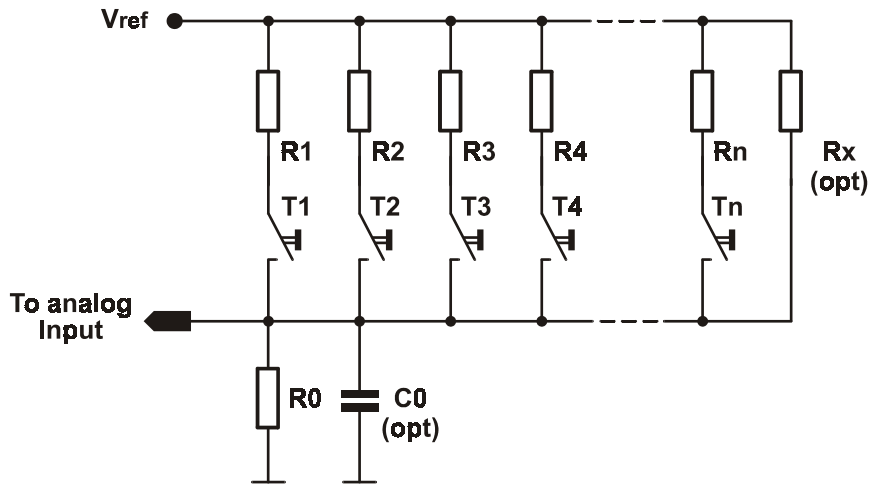
Mit INDEX_1D können mehrere Tasten an einem einzigen I/O-Pin des Tigers angeschlossen und eingelesen werden. Hierzu nachfolgend einige Schaltungs-Beispiele und Überlegungen dazu. In der folgenden Schaltung haben die Tasten weiter links eine Priorität gegenüber den rechten Tasten. Es wird immer nur eine gedrückte Taste erkannt.

3

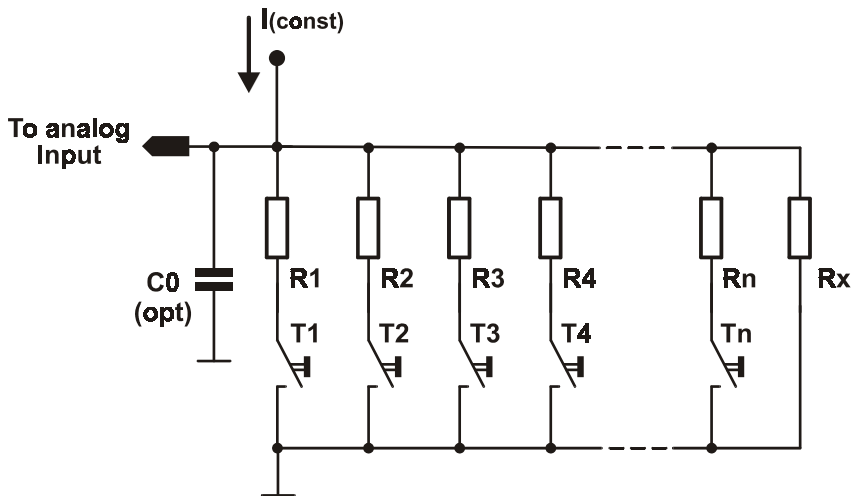


Keyboards using just 1 analog input pin

Diese Schaltungsvarianten nutzen den analogen Meßbereich mit erhältlichen Widerstandswerten besser aus. Es ist möglich, Tastenkombinationen zu erkennen.



a)



b)

Keyboards using just 1 analog input pin

INDEX_1D

In beiden Schaltungen wird von einer Parallel Schaltung Gebrauch gemacht. Die Erkennung welche Taste gedrückt ist, erfolgt durch die unterschiedlichen Widerstandswerte R1 bis Rn die jeweils mit der zugehörigen Taste in Reihe liegen. Im Fall a) entsteht ein Spannungs-Teiler mit der Ausgangs-Spannung:

$$\frac{V_{ref} * R_0}{R_n + R_0}$$

3

Im Fall b) entsteht durch die Einspeisung eines Konstantstromes eine Ausgangs-Spannung, die unmittelbar dem entstehenden Gesamt-Widerstand proportional ist:

Kontinuierlichen Eingangswerten wird eine begrenzte Anzahl fester Stufen zugeordnet. Diese Zuordnung kann linear, logarithmisch oder in jeder anderer Skalierung erfolgen.

Die Notation im BASIC-Programm ist kompakt und enthält nur die beiden Relevanten Angaben: Eingangs-Wert „WERT“ und die Abbildungs-Tabelle in feste Boxen: „BOX\$“:

I = INDEX_1D (WERT, BOX\$)

INDEX_1D baut diese Zuordnung über eine Tabelle mit dem Box-Grenzen auf, so daß man im Aufbau der Zuordnungsfunktion weitgehendst frei ist. Allein die stetig, monoton steigende Charakteristik ist vorgegeben:

Die Realisierung einer Tastatur mit Hilfe eines Analog-Eingangs wird von INDEX_1D ideal unterstützt. Wünschenswert ist, den gesamten analogen Eingangsreich (z.B. von 0...5V) möglichst gleichmäßig auszunutzen.

INDEX_2D**INDEX_2D (X1, Y1, BOXES\$)**

INDEX_2D: 2D-Wertklassen, X/Y-Positionen -> Index, Touch-Screen
(nur in Projektmodell 'PM_FULL')

Funktion: Weist Feldern in einer Fläche Indexnummern zu und liefert zu den Eingangswerten X1 und Y1 den passenden Index. Beispiel: Berührflächen auf einem Touchpanel.

X1, Y1	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG im Bereich von -16383...+16383 und stellen die aktuelle Position innerhalb einer Fläche dar. WORD-Variable werden entsprechend vorzeichenbehaftet gesehen.
Boxes\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält eine Tabelle der aufgespannten Flächen in X-Y-Koordinaten-Paaren. Die zweiten Koordinatenwerte für X und Y müssen größer sein als die ersten. Der Wertebereich ist wie bei den Koordinatenwerten X1, Y1 zu sehen.
Funktionswert	ist vom Typ LONG und gibt den Index des Tabelleneintrags zurück, dessen Fläche die Position X1, Y1 enthält. Die Zählung beginnt mit 1. Der Rückgabewert ist 0, wenn X1, Y1 in keiner Fläche liegt. Eine Typwandlung bei der Zuweisung findet automatisch statt.
	1...n: Bereich wurde getroffen
	0: außerhalb der Bereiche
	-1: X1 oder Y1 lagen außerhalb des zulässigen Wertebereichs von -16383...+16383

Index_2D ist ein spezieller Konverter für Anwendungen mit X-/Y-Eingabe-Geräten, wie z.B.:

- Touch-Screen
- Mouse
- Joystick
- Trackball
- Scanner

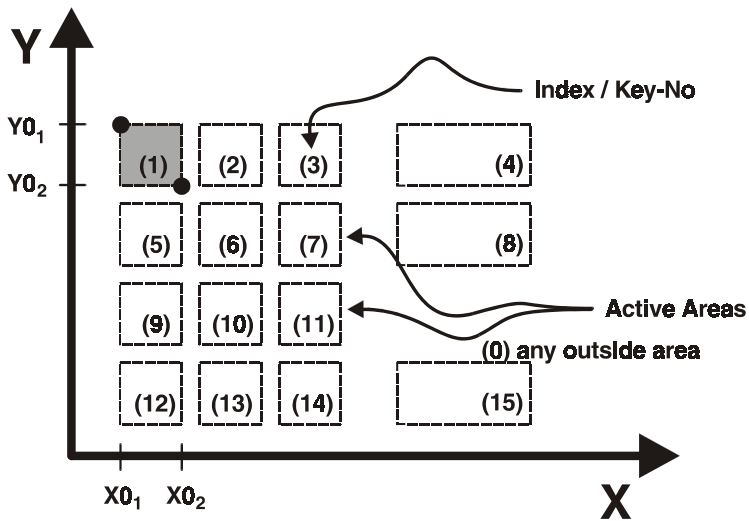
INDEX_2D

Der Konverter setzt X-/Y-Positionsangaben um in vorbestimmte Index-Werte, die bestimmten (Bildschirm-)Fenstern in der X-/Y-Ebene zugeordnet sind:

3



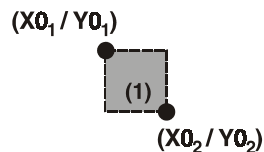
Screen Printout



Touch Areas in X/Y

The active area defined by:

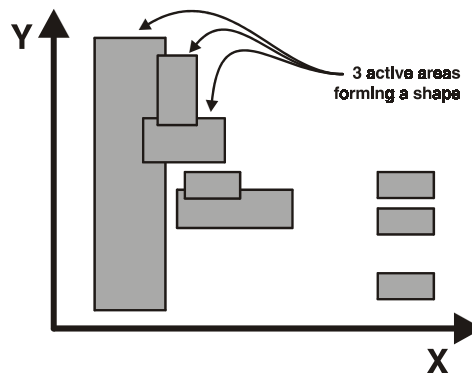
- left-upper corner and
- right lower corner



INDEX_2D ermöglicht es, sehr einfach mittels Touchscreen Tastaturen oder Menü-Auswahlen auf dem Display zu realisieren. Die Berührung einer Tasten-Position auf dem LCD generiert ein X-/Y-Koordinaten-Paar. Diese Position entspricht, wenn sie innerhalb einer vordefinierten Box liegt, dem Antippen einer bestimmten Taste oder eines bestimmten Bildschirm-Fensters. INDEX_2D erkennt nun ob und in welcher von mehreren Boxen bzw. Bildschirmfenstern eine solche X-/Y-Koordinate liegt. Die Darstellung auf dem Bildschirm muß nicht notwendigerweise Tastendarstellungen im üblichen Sinne enthalten:



Screen Printout



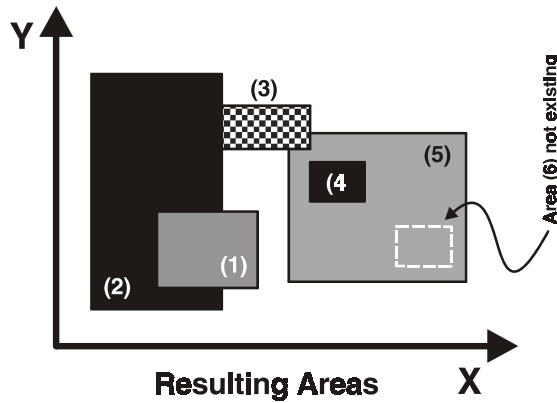
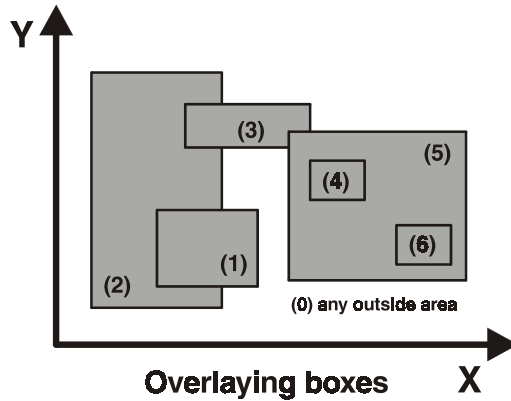
Shapes formed by several boxes

Bei Berührtasten (Touchpanel) wird über der Fläche der Anzeige (LCD, Monitor) der Berührungspunkt ermittelt und einer ‚Taste‘ zugeordnet, sofern die Berührung innerhalb einer Fläche stattfindet, wo sich eine Taste befindet. Die X- und Y-Werte des Berührungspunktes müssen innerhalb eines Bereichs liegen. Wo sich Tasten auf der Fläche befinden, bestimmt die Anwendung, in der Regel sogar dynamisch. Je nach Bedarf befinden sich Tasten oder anklickbare Symbole an verschiedenen Stellen der Fläche.

INDEX_2D

INDEX_2D erhält im String Click_Boxes\$ die Definitionen der Tastenflächen, indem jeweils die linke obere Ecke und die rechte untere Ecke als WORD-Koordinaten angegeben wird. Es werden für eine Fläche 4 WORD-Werte angegeben, die zwei X-/Y-Koordinaten darstellen. Ein Flächeneintrag belegt mit zwei WORD-X-Y-Koordinaten 8 Bytes. Die zweiten Koordinatenwerte für X und Y müssen größer sein als die ersten. Bei Touch-Screens kann sich je nach physikalischem Anschluß die X- und/oder die Y-Richtung umkehren.

3



Das folgende Beispiel läuft gibt den Indexwert auf einem Text-LCD aus. Es kann z.B. auf dem Plug & Play Lab ausprobiert werden und ermöglicht ein Verständnis anhand der Zahlen. Es trifft nur bei der ersten Ausgabe eine Fläche.

Programmbeispiel:

```

' -----
' Name: INDEX_2D.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A1$, A2$, A3$                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

' spanne 3 Felder in einem gedachten LCD 240 x 128 auf
' jeweils 10x10 gross, 10 Pixel vom oberen Rand
' Koordinaten als WORD, niederwertiges Byte zuerst
'   X1=10 Y1=10 X2=20 Y2=20
  A1$ = "<10><0><10><0><20><0><20><0>"
'   X1=30 Y1=10 X2=40 Y2=20
  A2$ = "<30><0><10><0><40><0><20><0>"
'   X1=50 Y1=10 X2=60 Y2=20
  A3$ = "<50><0><10><0><60><0><20><0>"
  A$ = A1$ + A2$ + A3$

  PRINT #1, INDEX_2D ( 33, 12, A$ )    ' Index beginnt mit 1
  PRINT #1, INDEX_2D ( 33, 22, A$ )    ' 2. Feld -> 2
                                         ' ausserhalb -> 0
END                                     ' Ende Task MAIN

```

3

Siehe auch: INDEX_1D, DEBOUNCE, LCD-Kit: TKB_KEY3.TIG

INIB_BYTE\$

INIB_BYTE\$

INIB_BYTE\$ (String, Tabelle, Start)

INIB_BYTE\$: De-Komprimiert I-Nibbles wieder zu Bytes
(nur in Projektmodell 'PM_FULL')

Funktion: Dekomprimiert Incremental-Nibbles aus **String** wieder in Bytes. Die Dekomprimierung ist die inverse Funktion zu BYTE_INIB\$.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu dekomprimierenden Zeichen in Form von Nibbles.
Tabelle	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die der Komprimierung zugrundeliegenden Inkrementalwerte.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 0...255 und bestimmt den Startwert zur Dekomprimierung.
Funktionswert	ist vom Typ STRING und enthält den dekomprimierten String.

INIB_BYTE\$ erweitert die Datengröße der Nibbles im String von 4 Bit auf jeweils ganze Bytes (8 Bit). Die Nibbles adressieren einen von 16 Werten in der Inkremental-Tabelle, der der Differenz zwischen zwei aufeinanderfolgenden 8-Bit-Werten entspricht. Die Werte im Tabellenstring werden als vorzeichenbehaftete Bytes gesehen (-1 = FFH, -127 = 80H). Die Funktion INIB_BYTE\$ ordnet jedem Nibble des Quellstrings den entsprechenden Differenzwert zu und berechnet das Byte im Zielstring aus dem vorherigen Byte plus Differenzwert.

Das zweite String-Argument der Funktion enthält die Tabelle, die mit genau 16 Bytes die Inkrementalwerte festlegt. Die 16 Werte sind in einem Nibble von 0 bis 15 adressierbar. Sinnvollerweise ist der hier verwendete Tabellenstring dem bei der Komprimierung verwendeten angepasst. Der Aufbau des Tabellen-Strings muß genau diesen Kriterien entsprechen:

- enthält genau 16 Bytes
- Bytes 0...7 enthalten Werte von 0...7FH (positiv, steigend)
- Bytes 8...15 enthalten Werte von FFH...80H (negativ -1...-128 fallend)
- es kommen niemals 2 gleiche Werte vor
- die positiven Werte sind monoton steigend
- die negativen Werte sind monoton fallend

Programmbeispiel:

```

'-----
' Name: INIB_BYTE$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I
  STRING A$, B$, T$(16)                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  T$ = "00 01 02 04 08 18 40 7F&
      FF FE FC F8 F0 E0 C0 80"%
  A$ = "<0><28H><A8H>"                   ' das is A$
  PRINT #1, "A$=";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1                  ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  B$ = INIB_BYTE$ ( A$, T$, 7FH )         ' dekomprimiere increm. Nibble
  PRINT #1, "<13><10>T$=";               ' zeige Kompressionstabelle
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(T$,I,1));
  NEXT
  PRINT #1, "<13><10>B$=";                 ' zeige Ergebnis in B$
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(B$,I,1));
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                       ' Ende Task MAIN

```

3

Siehe auch: Komprimierung BYTE_INIB\$, Absolut-Conversion ANIB_BYTE\$,
BYTE_ANIB\$

INPUT

INPUT

INPUT #Dev_Nr. [, #Sek_Adr][, #Func_Nr], Variable

INPUT: Formatierte Daten-Eingabe von Dev.-Treiber, Trennzeichen

Instruktion: INPUT ist eine Eingabe-Instruktion, die stets mit einem DEVICE-Treiber zusammenarbeitet (LCD1-Tastatur, serielle Treiber, paralleler Eingang, MF2-Tastatur).

3

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc, können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

INPUT liest String oder Zahl bis zu einem Begrenzerzeichen von Gerät **Dev_Nr** in die **Variable** ein. Standard-Begrenzerzeichen sind KOMMA sowie alle Steuerzeichen von \$00...\$1F. INPUT wartet darauf bis Eingabedaten vorliegen. Die CPU-Zeit wird so lange anderen Tasks zur Verfügung gestellt.

#Dev_Nr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-Gerätes an, wie er zu Beginn des Programms mit IN-STALL_DEVICE definiert worden ist.

- #Sek_Adr** Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.
- #Func_Nr** Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.
- Variable** ist vom Datentypen BYTE, WORD, LONG oder STRING.

INPUT wartet auf das Begrenzerzeichen. Die folgende Instruktion in dieser Task wird erst ausgeführt, wenn es eine Eingabe gegeben hat. Alle anderen Tasks laufen weiter, die CPU-Zeit der wartenden Task wird den anderen Tasks zusätzlich zur Verfügung gestellt.

Die maximale INPUT-Länge ist begrenzt durch

- maximal 255 Zeichen
- maximale Länge der Variablen
- Größe des Eingangs-Puffers (abhängig vom Device-Treiber).

Programmbeispiel:

```

'-----
' Name: INPUT.TIG
'-----
#INCLUDE KEYB_PP.INC          ' dt. Keyboard Belegung
TASK MAIN                     ' Beginn Task MAIN
  STRING A$                   ' STRING-Variable deklarieren
  LONG L                      ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  CALL INIT_KEYB (1)          ' setze Keyboard + div.
  PRINT #1, "<l>input STRING"   ' Ausgabe auf LC-Display
  INPUT #1, A$                 ' STRING-Input (kein Echo!)
  PRINT #1, "Input = "; A$     ' Ausgabe auf LC-Display
  PRINT #1, "input LONG-Value" ' Ausgabe auf LC-Display
  INPUT #1, L                  ' LONG-Input (kein Echo!)
  PRINT #1, "Input ="; L      ' Ausgabe auf LC-Display
END                             ' Ende Task MAIN

```

INPUT_LINE

INPUT_LINE #Dev_Nr. [, #Sek_Adr][, #Func_Nr], Variable

INPUT_LINE: Formatierte Daten-Eingabe von Dev.-Treiber, Zeilen

Instruktion: INPUT_LINE ist eine Eingabe-Instruktion, die stets mit einem DEVICE-Treiber zusammenarbeitet. (LCD1-Tastatur, serielle Treiber, paralleler Eingang, MF2-Tastatur).

3

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc, können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

INPUT_LINE liest einen String ein von Gerät **Dev_Nr** in die **Variable** ein. Begrenzzeichen ist RETURN. INPUT_LINE wartet darauf bis Eingabedaten vorliegen. Die CPU-Zeit wird so lange anderen Tasks zur Verfügung gestellt. Im Gegensatz zu INPUT Instruktion, werden andere Trennerzeichen wie KOMMA mit eingelesen.

#Dev_Nr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-Gerätes an, wie er zu Beginn des Programms mit IN-STALL_DEVICE definiert worden ist.

#Sek_Adr	Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.
#Func_Nr	Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.
Variable	ist vom Datentyp STRING.

INPUT_LINE wartet auf das Begrenzerzeichen. Die folgende Instruktion in dieser Task wird erst ausgeführt, wenn es eine Eingabe gegeben hat. Alle anderen Tasks laufen weiter, die CPU-Zeit der wartenden Task wird den anderen Tasks zusätzlich zur Verfügung gestellt.

Die maximale INPUT-Länge ist begrenzt durch

- maximal 255 Zeichen
- maximale Länge der Variablen
- Größe des Eingangs-Puffers (abhängig vom Device-Treiber).

Programmbeispiel:

```

'-----
' Name: INPUT_LINE.TIG
'-----
#INCLUDE KEYB_PP.INC           ' dt. Keyboard Belegung
TASK MAIN                     ' Beginn Task MAIN
    STRING A$                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 80h, 8
CALL INIT_KEYB (1)             ' setze Keyboard + div.
PRINT #1, "<l>input STRING"      ' Ausgabe auf LC-Display
INPUT_LINE #1, A$              ' Zeileneingabe (kein Echo!)
PRINT #1, "Input = "; A$       ' Ausgabe auf LC-Display
END                             ' Ende Task MAIN

```

INSTALL_DEVICE

INSTALL_DEVICE #Dev_Nr, „Dateiname“, Optionen

INSTALL_DEVICE: Device Treiber installieren

Instruktion: Bindet den Device-Treiber ein, der durch **Dateiname** spezifiziert ist und weist dem Gerät (Device) die Nummer **Dev_Nr** zu. Alle Ein- und Ausgaben auf dieses Gerät werden im Programm unter dieser Device-Nr. vorgenommen.

3

#Dev_Nr ist bei dieser Instruktion immer eine Konstante und gibt die Nummer des I/O-Gerätes an.

Dateiname steht in Anführungszeichen und besteht aus **Laufwerk, Pfad, Dateiname**. Werden Laufwerk und Pfad nicht angegeben, sucht der Compiler den Treiber in dem Verzeichnis, welches unter **Verzeichnisse** im Menü **Optionen** eingegeben ist.

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc, können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungsbereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

Programmbeispiel:

```
'-----  
' Name: INSTALL_DEVICE.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, "Hello World"                ' Auf LC-Display ausgeben  
END                                       ' Ende Task MAIN
```

3

INSTR

INSTR

INSTR (*String1*, *String2*, *Position*, *Option*)

INSTR: Sucht Zeichenfolge in einem String, liefert Position

Funktion: Sucht auf einer Länge von maximal 255 Bytes in **String2** ab **Position** nach **String1**. Liefert die Position zurück, an der **String1** gefunden wurde, oder -1. Die erste Position ist 0.

3

- String1** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält den zu suchenden String.
- String2** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING. In String2 wird gesucht.
- Position** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition an.
- Option** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG mit Bits der folgenden Bedeutung:

Option binär / dez.	Suchrichtung	7 o. 8-Bit-Modus	Groß- /Kleinschreibung berücksichtigen
0000 / 0	vorwärts	8-Bit	ja
0001 / 1	rückwärts	8-Bit	ja
0010 / 2	vorwärts	7-Bit	ja
0011 / 3	rückwärts	7-Bit	ja
0100 / 4	vorwärts	8-Bit	nein
0101 / 5	rückwärts	8-Bit	nein
0110 / 6	vorwärts	7-Bit	nein
0111 / 7	rückwärts	7-Bit	nein

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

INSTR sucht in String2 über eine Länge von 255 Bytes nach String1. Die Suchlänge der Funktion ist beschränkt, um lange, nicht zu unterbrechende Suchzeiten zu verhindern. Bei erfolgreicher Suche wird die Position von String1 zurückgeliefert, an der das erste Zeichen von String1 gefunden wurde. Wenn String1 nicht gefunden wird, ist das Ergebnis -1. Die Zuweisungsvariable sollte daher eine (vorzeichenbehaftete) LONG-Zahl sein.

Programmbeispiel:

```

'-----
' INSTR.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, INSTR("A T","this is a test",8,4) ' Ausgabe INSTR
END                                      ' Ende Task MAIN

```

3

Siehe auch: MID\$, STOSS

INT

INT

INT (Wert)

INT: Liefert nächst gelegene Ganzzahl eines REAL-Wertes

Funktion: Liefert die größte Ganzzahl zurück, die kleiner oder gleich **Wert** ist.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Wertebereich der Zuweisungsvariablen. Nachkommastellen gehen dabei verloren.

Funktionswert ist vom Typ REAL. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Die Funktion INT liefert als Funktionswert den nächstkleineren Wert Bei dem Argument '-3.8' liefert INT den Funktionswert '-4'. FIX schneidet Nachkommastellen ab, so daß zu '-3.8' der Funktionswert '-3.0' zurückgegeben wird.

Programmbeispiel:

```
'-----  
' Name: INT.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
  LONG L                                ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  L = INT (-2.7)                          ' auf ganze Zahl abrunden  
  PRINT #1, "INT (-2.7) ="; L              ' Ausgabe auf LC-Display  
  L = INT (-2.3)                          ' auf ganze Zahl abrunden  
  PRINT #1, "INT (-2.3) ="; L              ' Ausgabe auf LC-Display  
  L = INT (2.3)                           ' auf ganze Zahl abrunden  
  PRINT #1, "INT ( 2.3) ="; L              ' Ausgabe auf LC-Display  
  L = INT (2.7)                           ' auf ganze Zahl abrunden  
  PRINT #1, "INT ( 2.7) ="; L              ' Ausgabe auf LC-Display  
END                                        ' Ende Task MAIN
```

INTEGRAL_FIFO

INTEGRAL_FIFO Name, Scale, Wert

INTEGRAL_FIFO: Fifo benutzen als Kurzzeit-Integrator (Window)

Instruktion: Fügt den **Wert** in den FIFO-Puffer **Name** und liest anschliessend einen durch Integralbildung bereinigten **Wert** aus dem FIFO-Puffer **Name**. (FIFO=First-In-First-Out)

Es werden keine Elemente entnommen. Sinnvollerweise ist der FIFO-Puffer genauso groß wie der Integralbereich. Die Größe des FIFO-Puffers ist immer eine Potenz von 2.

Nach der Einrichtung des FIFO-Puffers wird durch die Instruktion START_FIFO der Integralbereich sowie der Anfangswert der Integralbildung gesetzt (siehe Seite 200).

Der Startwert mit START_FIFO muß gesetzt werden.

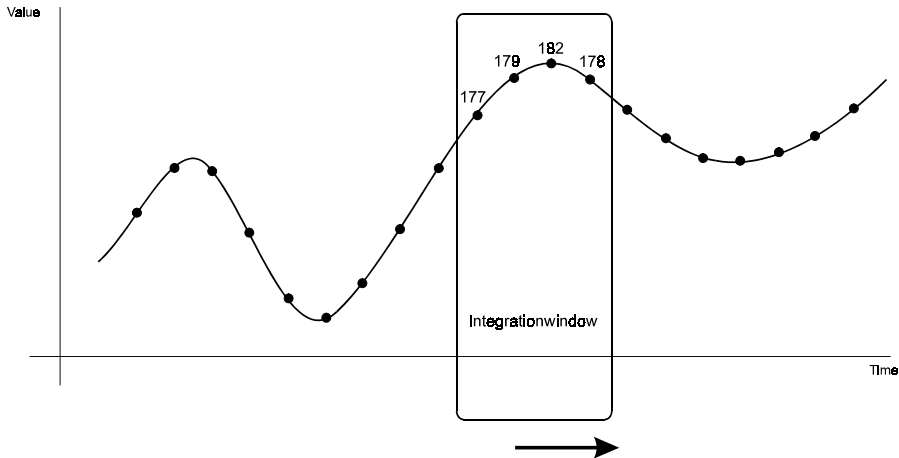
Scale gibt einen Faktor an, mit dem der Wert skaliert wird. Werte für **Scale** sind Potenzen von 2 (1, 2, 4, ...2k, ...2G) oder (-1 -2, -4, -8, ...-2k,...-2G).

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist das neue Element, welches in den FIFO-Puffer eingefügt wird.

Die Integralbildung stellt einen Filter dar, der die höherfrequenten Anteile und Rauschen „herausrechnet“, zum Beispiel aus Meßwerten, die mit dem A/D-Wandler eingelesen wurden.

INTEGRAL_FIFO

Im folgenden Beispiel wird der Wert 181 in dem FIFO-Puffer mit der Länge 4 durch Integralbildung mit Scale=1 berechnet:



	FIFO					Wert
vorher	177	179	182	178	<--	181
Wert einfügen	179	182	178	181		
Summe/Anzahl	(179+182+178+181)/4 =				-->	180
mal Scale (1)	179	182	178	181		180

Um einen zur Integralbildung verwendeten FIFO-Puffer von Anfang an mit halbwegs plausiblen Werten zu erhalten, ist dieser vor der ersten Benutzung mit der Instruktion `START_FIFO` mit Startwerten dem ungefähr erwarteten Wert zu füllen. Geschieht dies nicht, so beginnt der erste Integralwert mit einem zufälligen Wert, der sich erst nach einer Anzahl von Integralwertbildungen auf den richtigen Wert einstellt.

Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```

'-----
' Name: INTEGRAL_FIFO.TIG
'-----
TASK Main
    FIFO BUFFER(32) of LONG
    WORD W1
    LONG L1
    ' LCD-Treiber installieren (BASIC-Tiger)
    INSTALL DEVICE #1, "LCD1.TDD"
    ' LCD-Treiber installieren (TINY-Tiger)
    ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
    CLEAR_FIFO BUFFER
    START_FIFO BUFFER,32,0
    FOR W1 = 0 TO 31
        L1 = RND(1)
        PRINT #1, "<1>Elements: ";W1
        PRINT #1, "          L1: ";L1
        INTEGRAL_FIFO BUFFER,0,L1
        PRINT #1, "Integral: ";L1
        WAIT_DURATION 1000
    NEXT
END
' Beginn Task MAIN
' FIFO-Buffer definieren
' WORD-Variable deklarieren
' LONG-Variable deklarieren
' FIFO-Buffer loeschen
' Startwert setzen
' Einrichten der FOR-Schleife
' Zufallswert generieren
' W1 auf LC-Display ausgeben
' L1 auf LC-Display ausgeben
' Integral bilden
' L1 auf LC-Display ausgeben
' 1 Sek warten
' naechster Wert fuer W1
' Ende Task MAIN

```

3

INV_BIT

INV_BIT

INV_BIT Variable, Bitnr

INV_BIT: Invertiert 1 Bit in numerischer Variable

Instruktion: Invertiert das Bit mit der Nummer **Bitnr** in der **Variablen**.

Variable ist vom Datentypen BYTE, WORD oder LONG.

Bitnr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welches Bit invertiert werden soll. Ist Bitnr größer als die Anzahl Bits der Variablen, dann verändert sich die Variable nicht.

Programmbeispiel:

```
'-----
' Name: INV_BIT.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE A                                ' BYTE-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2> k 0.0.0.0.2UH<2><2> k 0.0.0.0.2"
  A = 37h                                ' Variable besetzen
  INV_BIT A,2                            ' Bit 2 invertieren
  PRINT USING #1, 37h; " --> "; A        ' Ausgabe auf LC-Display
END                                       ' Ende Task MAIN
```


LD

LD (Wert)

LD: Logarithmus Dualis, Basis 2

Funktion: Liefert den dualen Logarithmus von **Wert** zurück.

Wert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Funktionswert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von 0 bis 1023 und . Eine Typwandlung findet automatisch statt.

Programmbeispiel:

```
'-----
' Name: LD.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG value                             ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0"
  FOR value = 0 TO 200                    ' Werte von 0 bis 200
    PRINT #1, "<1>LD ("; value; ")" = "    ' Zeige Argument
    PRINT USING #1, LD(value/10.0)        ' Logarithmus dualis ausgeben
    WAIT DURATION 1000                   ' 1 sek warten
  NEXT                                    ' naechster Wert
END                                        ' Ende Task MAIN
```

Siehe auch: LOG, LN

LEFT\$

LEFT\$

LEFT\$ (String, Länge)

LEFT\$: Linker Teilstring

Funktion: Liefert den linken Teil der angegebenen **Länge** aus der Zeichenfolge **String** zurück.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.

Länge ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Zeichen an, die in dem Zielstring enthalten sind.

Funktionswert ist vom Typ STRING und enthält den linken Teil vom Quellstring mit der angegebenen Länge. Die Länge ist jedoch nie größer als die ursprüngliche Länge des Strings.

Programmbeispiel:

```
'-----
' Name: LEFT$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "Hello, I am the BASIC Tiger"      ' String vorbesetzen
  PRINT #1, LEFT$ (A$,5)                  ' linken Teilstring ausgeben
END                                         ' Ende Task MAIN
```

Siehe auch: MID\$, RIGHT\$, LEN

LEN

LEN (*String*)

LEN: Ermittelt die aktuelle String-Länge

Funktion: Liefert die Anzahl der Zeichen der Zeichenfolge **String** zurück.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```

' -----
' Name: LEN.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "ABCDEFGHIJ"                      ' A$ vorbesetzen
  PRINT #1, "A$ = "; A$                  ' Ausgabe auf LC-Display
  PRINT #1, "Length of A$:"; LEN (A$)    ' Ausgabe der Laenge von A$
END                                       ' Ende Task MAIN

```

Siehe auch: VAL_NUM, VAL_REAL, ASC

LEN_FIFO

LEN_FIFO

LEN_FIFO (*Fifobu*)

LEN_FIFO: Gesamt-Länge des Fifo melden

Funktion: Liefert die Anzahl der augenblicklich im FIFO-Puffer **Fifobu** befindlichen Datenelemente zurück. (FIFO=First-In-First-Out)

Fifobu ist der Name der Variablen vom Typ FIFO.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
'-----
' Name: LEN_FIFO.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER(512) OF WORD                ' FIFO-Buffer deklarieren
  WORD I                                  ' WORD-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 51                          ' 52 Werte in den
    PUT_FIFO BUFFER, I                     ' FIFO-Buffer schreiben
  NEXT                                     '
  PRINT #1, "Values: "; LEN_FIFO (BUFFER) ' Ausgabe auf LC-Display
END                                         ' Ende Task MAIN
```

Siehe auch: FREE_FIFO

LET\$

LET\$ (String)

LET\$: String-Zuweisung, high-speed
(nur in Projektmodell 'PM_FULL')

Funktion: Führt eine String-Zuweisung schneller aus als der Operator „=“.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zuzuweisenden Zeichen.

Funktionswert ist vom Typ STRING und enthält den zugewiesenen String.

Die Zuweisung von großen Strings ab ca. 50 Bytes wird mit LET\$ schneller ausgeführt als durch den Operator „=“. Solche Zuweisungen kommen insbesondere bei der Grafikbearbeitung und Sound-Bearbeitung oft vor.

Programmbeispiel:

```

' -----
' Name: LET$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$(4k), B$(4k)                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'   INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = FILL$ ( "ABCDEFGH IJ", 4k )       ' A$ vorbesetzen
  B$ = LET$ ( A$ )                       ' zuweisen
  PRINT #1, "Length of B$:"; LEN (B$)   ' Ausgabe der Laenge von B$
END                                       ' Ende Task MAIN

```

Siehe auch: Operator „=“

LIMIT

LIMIT

LIMIT (Var, Min, Max)

LIMIT: Begrenzt Integer Wert auf zulässigen Bereich

Funktion: Liefert den Wert **Var** zurück, wenn zwischen **Minimalwert** und **Maximalwert**, jedoch den Minimalwert, wenn Var kleiner ist, und den Maximalwert, wenn Var größer ist.

3

Var	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.
Min	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Minimalwert an. Min ist kleiner als Max
Max	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Maximalwert an. Max ist größer als Min.
Funktionswert	ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

LIMIT weist automatisch einen Funktionswert zu, der die gesetzten Grenzen nicht unter- oder überschreitet.

LIMIT ersetzt die folgende Konstruktion in BASIC, ist schneller als diese und kann außerdem in anderen Ausdrücken direkt verwendet werden.

```
IF VAR < MIN THEN      ' Check for MIN
  VAR=MIN
ELSE
  IF VAR > MAX THEN    ' Check for MAX
    VAR=MAX
  ENDIF
ENDIF
```

Beachte: Falls MIN > MAX, so vertauscht LIMIT die beiden Werte, so daß es dennoch zu einem sinnvollen Ergebnis kommt.

Die nicht vorzeichenbehafteten BYTE und WORD werden in der Funktion wie LONG behandelt und erhalten bei negativen Grenzwerten entsprechend große Werte. Ähnliche Effekte entstehen, wenn der Maximalwert jenseits des Wertebereichs der Variablen für den Funktionswert ist. Siehe Beispielprogramm.

Programmbeispiel:

```

-----
' Name: LIMIT.TIG
-----
TASK MAIN                                     ' Beginn Task MAIN
  BYTE B, Mi, MA
  WORD W
  LONG A, L
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A = 17
  L = LIMIT ( A, -2, 15 )                     ' zwischen -2 und 15
  PRINT #1, "A=17 -> L=";L                     ' Ergebnis: 15
  A = -17
  L = LIMIT ( A, -2, 15 )                     ' zwischen -2 und 15
  PRINT #1, "A=-17 -> L=";L                     ' Ergebnis: -2
  A = -17
  W = LIMIT ( A, -2, 15 )                     ' zwischen -2 und 15
  PRINT #1, "A=-17 -> W=";W                     ' Ergebnis: 65534
  A = -17
  B = LIMIT ( A, -2, 15 )                     ' zwischen -2 und 15
  PRINT #1, "A=-17 -> B=";B                     ' Ergebnis: 254
  WAIT DURATION 2000
  A = 259
  B = LIMIT ( A, -2, 257 )                     ' zwischen -2 und 257
  PRINT #1, "<1>A=259 -> B=";B                     ' Ergebnis: 1
END                                             ' Ende Task MAIN

```

3

Siehe auch: Operatoren

LIN_APPROX

LIN_APPROX (X1, Y1, X2, Y2, Xn)

LIN_APPROX: Zwischenschritte durch lineare Approximation

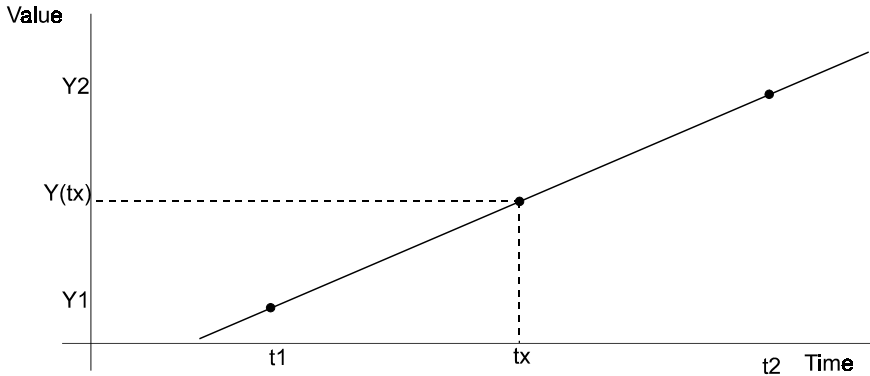
Funktion: Liefert den Y-Wert zu **Xn** durch lineare Approximation. Eine Gerade im zweidimensionalen Koordinatensystem wird durch die zwei Punkte **X1,Y1** und **X2,Y2** beschrieben. Der Y-Wert eines dritten Punktes mit gegebenem X-Wert wird von der Funktion LIN_APPROX geliefert.

3

X1,Y1	sind Konstanten, Variable oder Ausdrücke vom Datentyp BYTE, WORD oder LONG und sind die Koordinaten des ersten Punktes der Geraden.
X2,Y2	sind Konstanten, Variable oder Ausdrücke vom Datentyp BYTE, WORD oder LONG und sind die Koordinaten des zweiten Punktes der Geraden.
Xn	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD oder LONG und ist die X-Koordinate des gesuchten Punktes der Geraden.
Funktionswert	ist vom Typ LONG und ist die Y-Koordinate des Punktes auf der Geraden mit dem X-Wert Xn. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Die Funktion LIN_APPROX liefert sowohl Y-Koordinaten von Punkten, deren X-Koordinaten zwischen X1 und X2 liegen als auch von Punkten, deren X-Koordinaten außerhalb liegen.

Der Versuch, mit den ganz-zahligen LONG-Zahlen die lineare Approximation selbst zu berechnen, scheitert bereits bei der Berechnung der Steigung der Geraden: Beispiel: Bei Delta-X=100 sei Delta-Y=120, die Steigung also 1,2. Die Berechnung liefert aber für $120/100 \rightarrow 1$, da LONG-Zahlen eben immer ganz-zahlig sind.



3

Programmbeispiel:

```

'-----
' Name: LIN_APPROX.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG X1,Y1,X2,Y2,Xn                    ' LONG-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  USING "SD<5><4> V0.0.0.3,3"            ' Format fuer Ausgabe
  X1=1000                                ' 1. X-Koordinate
  Y1=1000                                ' 1. Y-Koordinate
  X2=11000                                ' 2. X-Koordinate
  Y2=3000                                ' 2. Y-Koordinate
  FOR Xn=-10000 TO 20000 STEP 1000      ' Werte fuer Xn
    Yn = LIN_APPROX (X1,Y1,X2,Y2,Xn)    ' Berechnung von Yn
    PRINT USING #1, "<1>Xn =" ;Xn        ' Ausgabe von Xn
    PRINT USING #1, "Yn =" ;Yn          ' Ausgabe von Yn
    WAIT DURATION 1000                  ' 1 Sek warten
  NEXT                                  ' naechster Wert fuer Xn
END                                      ' Ende Task MAIN

```

LL_IPORT_AND

LL_IPORT_AND

LL_IPORT_AND *Log_Portadr, Bitmuster*

LL_IPORT_AND: Bitweise UND-Verknüpfung auf internem I/O-Port

Instruktion: Setzt durch eine logische UND-Operation auf dem internen Port mit der Adresse **Log_Portadr** diejenigen Bits auf 0, die im **Bitmuster** auf 0 stehen.

3

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG.

LL_IPORT_AND ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_AND prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```
' -----  
' Name: LL_IPORT_AND.TIG  
' -----  
  
TASK MAIN                                ' Beginn Task MAIN  
    DIR_PORT 8,0                          ' Port 8 als Ausgang  
    LL_IPORT_OUT 8,11111111b              ' alle Pins von Port 8 high  
    WAIT DURATION 3000                    ' 3 Sek. warten  
    LL_IPORT_AND 8,10000001b               ' nur Pin 0 und 7 high lassen  
END                                        ' Ende Task MAIN
```

Siehe auch: LL_IPORT_OR, LL_IPORT_IN, LL_IPORT_OUT

LL_IPORT_IN

LL_IPORT_IN *Log_Portadr., Variable[, Maske]*

LL_IPORT_IN: Schnelles Einlesen von einem internem I/O-Port mit bitweiser UND-Verknüpfung

Instruktion: Liest den momentan anliegenden 8-Bit-Wert von dem internen Port mit der Adresse **Log_Portadr.** in **Variable** ein.

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Variable ist vom Datentypen BYTE, WORD oder LONG.

Maske ist eine Konstante vom Datentyp BYTE, WORD oder LONG und gibt optional die Bitmaske an, mit der gelesene Bits ausmaskiert werden.

LL_IPORT_IN ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_IN prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```

'-----
' Name: LL_IPORT_IN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE Value,Old                        ' BYTE-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 80h, 8
  DIR_PORT 8,255                        ' Port 8 als Eingang setzen
  Old = 0                               ' Variablen initialisieren
  Value = 0
  FOR X=0 TO 0 STEP 0                   ' Endlosschleife
    LL_IPORT_IN 8,Value,00001111b       ' lese Port 8, untere 4 Bits
    IF Value <> Old THEN                 ' Falls Wert geändert,
      PRINT #1, "<i>Value ="; Value      ' Ausgabe auf LC-Display
      Old = Value                       ' alten Wert sichern
    ENDIF
  NEXT                                  ' Ende Endlosschleife
END                                    ' Ende Task MAIN

```

Siehe auch: LL_IPORT_AND, LL_IPORT_OR, LL_IPORT_OUT, XIN

LL_IPORT_OR

LL_IPORT_OR

LL_IPORT_OR *Log_Portadr.*, *Bitmuster*

LL_IPORT_OR: Bitweise ODER-Verknüpfung auf internem I/O-Port

Instruktion: Setzt durch eine logische OR-Operation auf dem internen Port mit der Adresse **Log_Portadr.** diejenigen Bits auf 1, die im **Bitmuster** auf 1 stehen.

3

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG.

LL_IPORT_OR ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_OR prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```
' -----
' Name: LL_IPORT_OR.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  BYTE X,Y                               ' BYTE-Variablen deklarieren
  DIR_PORT 8,0                           ' Port 8 als Ausgang
  LOOP 9999999                            ' sehr viele Loops
    LL_IPORT_OUT 8,00000000b              ' Port 8, alle Bits low
    WAIT DURATION 500                     ' 500ms warten
    FOR X=0 TO 7                          ' Schleife ueber alle 7 Bits
      Y = EXP (2,X)                       ' berechne Wert von Bit X
      LL_IPORT_OR 8,Y                     ' Bit X zusaetzlich auf high
      WAIT DURATION 500                   ' 500ms warten
    NEXT X                                ' naechstes Bit
  ENDLOOP                                 '
END                                        ' Ende Task MAIN
```

Siehe auch: LL_IPORT_AND, LL_IPORT_IN, LL_IPORT_OUT

LL_IPORT_OUT

LL_IPORT_OUT *Log_Portadr., Wert*

LL_IPORT_OUT: Schnelle Ausgabe auf internem I/O-Port

Instruktion: Gibt **Wert** an dem internen Port mit der Adresse **Log_Portadr.** aus.

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG.

LL_IPORT_OUT ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_OUT prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```

' -----
' Name: LL_IPORT_OUT.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  DIR_PORT 8,0                          ' Port 8 als Ausgang
  LOOP 9999999                          ' sehr viele Loops
    LL_IPORT_OUT 8,10101010b            ' Port 8, ungerade Bits high
    WAIT DURATION 500                  ' 500 ms warten
    LL_IPORT_OUT 8,01010101b            ' Port 8, gerade Bits high
    WAIT DURATION 500                  ' 500 ms warten
  ENDLLOOP                              '
END                                    ' Ende Task MAIN

```

Siehe auch: LL_IPORT_AND, LL_IPORT_OR, LL_IPORT_IN, XOUT

LL_IPORT_PULSE

LL_IPORT_PULSE

LL_IPORT_PULSE *Log_Portadr., Maske, Anzahl*

LL_IPORT_PULSE: Toggelt ein Bit eines internen I/O-Ports

Instruktion: Gibt auf dem internen Port mit der Adresse **Log_Portadr.** auf denjenigen Bits, die in der **Maske** auf 1 stehen, die angegebene **Anzahl** Pulse (bis 255) aus.

3

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Maske ist eine Konstante vom Datentypen BYTE, WORD oder LONG und bestimmt, welche Portpins die Pulse ausgeben.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und gibt die Anzahl der Pulse an. Die Anzahl ist 0...255.

Vor der Pulsausgabe müssen die entsprechenden Pins auf 'Ausgang' gesetzt werden (DIR_PIN Portn,0 oder DIR_PORT Portn,0). Der Ruhepegel wird mit der OUT-Instruktion gesetzt. Die Pulse werden ausgegeben, indem die Pins invertiert werden. Am Ende stellt sich wieder der Ruhezustand ein. Mit einem Zähler kann mit LL_IPORT_PULSE eine einfache Anzeige realisiert werden, wobei nur ein I/O-Pin des Moduls verbraucht wird. Dabei wird der erste Puls verwendet, um den Zähler zurückzusetzen. Auch Schieberegisteranwendungen sind denkbar.

LL_IPORT_PULSE ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_PULSE prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```
'-----  
' Name: LL_IPORT_PULSE.TIG  
'-----  
TASK MAIN  
    DIR_PORT 8,0  
    LL_IPORT_OUT 8,11111111b  
    LL_IPORT_PULSE 8,00000001b,199  
END  
    ' Beginn Task MAIN  
    ' Port 8 als Ausgang  
    ' Ruhepegel festlegen  
    ' gebe 199 Pulse auf Pin 1 aus  
    ' Ende Task MAIN
```

LL_IPORT_XOR

LL_IPORT_XOR Log_Portadr., Bitmuster

LL_IPORT_XOR: Bitweise XOR-Verknüpfung auf internem I/O-Port

Instruktion: Invertiert durch eine logische XOR-Operation auf dem internen Port mit der Adresse **Log_Portadr.** diejenigen Bits, die mit den entsprechenden Bits im **Bitmuster** übereinstimmen. Die übrigen Bits bleiben unberührt.

Log_Portadr. ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG.

LL_IPORT_XOR ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: LL_IPORT_XOR prüft nicht die Zulässigkeit des Portzugriffs. Wenn z.B. auch ein Device-Treiber auf diesen Port zugreift, kann es zur Störung des Device-Treibers kommen. Low-Level-Instruktionen werden eingesetzt, wenn es auf hohe Ausführungsgeschwindigkeit ankommt.

Programmbeispiel:

```
' -----
' Name: LL_IPORT_XOR.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
    DIR_PORT 8,0                          ' Port 8 als Ausgang
    LL_IPORT OUT 8,11110000b              ' Port 8, Bits 4 bis 7 high
    LOOP 9999999                          ' sehr viele Loops
        WAIT DURATION 500                 ' 500 ms warten
        LL_IPORT_XOR 8,10000001b          ' Toggle Bit 0 & Bit 7
    ENDLOOP                               '
END                                        ' Ende Task MAIN
```

Siehe auch: LL_IPORT_AND, LL_IPORT_OR, LL_IPORT_IN, LL_IPORT_OUT

LLTOR

LLTOR

LLTOR (*Long_low*, *Long_high*)

LLTOR: Setzt 2 LONG-Werte zusammen zu 1 REAL-Variablen

Funktion: Fügt die zwei LONG-Zahlen **Long_low** und **Long_high** zu einer REAL-Zahl zusammen (**Long-Long-To-Real**).

Long_low ist eine Variable vom Typ LONG, die zuvor durch die Funktion LREAL erzeugt worden ist.

Long_high ist eine Variable vom Typ LONG, die zuvor durch die Funktion HREAL erzeugt worden ist.

Mit den Funktionen LREAL und HREAL lässt sich eine 8-Byte-REAL-Zahl in zwei 4-Byte-LONG-Zahlen zerlegen. Die Funktion LLTOR fügt nun die beiden 4-Byte-Stücke wieder zusammen.

Diese Funktionen ermöglichen das Weiterverarbeiten von REAL-Zahlen in byte- oder bit-orientierter Form, was sonst mit REAL-Zahlen nicht möglich wäre. Nach der Aufspaltung in zwei LONG-Zahlen können nun auch REAL-Zahlen mit der Funktion NTOSS in Strukturen oder Records eingebaut werden.

Programmbeispiel:

```
'-----
' Name: LLTOR.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  REAL R1,R2                             ' REAL-Variablen deklarieren
  LONG Hi,Lo                             ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
R1 = 12345.6789                           ' R1 vorbesetzen
Lo = LREAL (R1)                           ' Low 32 Bit von R1 --> Lo
Hi = HREAL (R1)                           ' High 32 Bit von R1 --> Hi
R2 = LLTOR (Lo,Hi)                        ' Erzeuge R2 aus Lo und Hi
USING "EF<0><1> <8>V0.0.0.0.0.0.0.1,8.0.0.0.0.0.0"
PRINT USING #1, "<1>R1="; R1              ' zeige R1 im Original
PRINT #1, "Lo="; Lo                      ' zeige Low 32 Bit von R1
PRINT #1, "Hi="; Hi                      ' zeige High 32 Bit von R1
PRINT USING #1, "R2="; R2                ' zeige R2 (Lo + Hi --> REAL)
END                                       ' Ende Task MAIN
```

Siehe auch: LREAL, HREAL

LN

LN (Wert)

LN: Logarithmus Naturalis, Basis e

Funktion: Liefert den natürlichen Logarithmus von **Wert** zurück.

Wert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Funktionswert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von 0 bis 708 und . Eine Typwandlung findet automatisch statt.

Programmbeispiel:

```
'-----
' Name: LN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG value                             ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0"
  FOR value = 0 TO 200                   ' Werte von 0 bis 200
    PRINT #1, "<1>LN ("; value; " ) ="    ' Zeige Argument
    PRINT USING #1, LN(value/10.0)       ' Logarith. naturalis ausgeben
    WAIT DURATION 1000                  ' 1 sek warten
  NEXT                                  ' naechster Wert
END                                     ' Ende Task MAIN
```

Siehe auch: LOG, LD

LOG

LOG

LOG (Wert)

LOG: 10-er Logarithmus

Funktion: Liefert den Zehner-Logarithmus von **Wert** zurück.

Wert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Funktionswert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von 0 bis 308 und . Eine Typwandlung findet automatisch statt.

Programmbeispiel:

```
' -----  
' Name: LOG.TIG  
' -----  
TASK MAIN                                ' Beginn Task MAIN  
    LONG value                            ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
    INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'    INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
    USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"  
    FOR value = 0 TO 200                    ' Werte von 0 bis 200  
        PRINT #1, "<1>LOG ("; value; ") =" ' Zeige Argument  
        PRINT USING #1, LOG(value/10.0)    ' 10er-Logarithmus ausgeben  
        WAIT DURATION 1000                 ' 1 sek warten  
    NEXT                                    ' naechster Wert  
END                                         ' Ende Task MAIN
```

Siehe auch: LN, LD

LOOP

LOOP Wert

[Anweisungsblock]

ENDLOOP

LOOP...ENDLOOP: Anzahl von Schleifen unbedingt ausführen

Instruktion: Führt den Anweisungsblock so oft aus, wie **Wert** vorgibt.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und ist positiv. Ist der Wert 0 oder negativ, wird der Anweisungsblock nicht ausgeführt.

Datentyp	Wertebereich
BYTE	0...255
WORD	0...65535
LONG	0...2147483647

Ein ENDLOOP kann nicht vor einem passenden LOOP ausgeführt werden. Daher ist ein Einsprung in die LOOP...ENDLOOP-Schleife nicht erlaubt, es ist jedoch möglich aus der Schleife herauszuspringen. Die LOOP-Schleife ist schneller als die FOR...NEXT-Schleife.

Die maximale Verschachtelungstiefe von LOOP...ENDLOOP-Schleifen und/oder FOR...NEXT-Schleifen beträgt 12 pro Task. Bei der Ermittlung der Verschachtelungstiefe zählen alle zur Zeit aktiven Loops einer Task mit, also auch die in aktiven Loops in z.Zt. laufenden Unterprogrammen.

LOOP

Programmbeispiel:

```
'-----
' Name: LOOP.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I                                ' BYTE-Variable deklarieren
  LONG J                                ' LONG-Variable deklarieren
  DIR_PORT 8,0                          ' Port 8 als Ausgang
  LOOP 99999999                         ' sehr viele Loops
    I = 1                               ' Variable I initialisieren
    J = 75                              ' Variable J initialisieren
    LOOP 8                              ' 8 Schleifendurchläufe
      OUT 8, 255, I                     ' I auf Port 8 ausgeben
      I = I ROL 1                       ' Bitweise links rotieren
      J = J + 75                        ' Pausenzeit verlängern
      WAIT DURATION J                  ' J ms warten
    '
  ENDLOOP                               '
ENDLOOP                                '
END                                    ' Ende Task MAIN
```

Siehe auch: WHILE, FOR...NEXT

LREAL

LREAL (Real)

LREAL: Low 32-Bits of REAL-Wert

Funktion: Liefert als Ergebnis in einer LONG-Zahl die niederwertigen 4 Byte der REAL-Zahl (**L**ow part of **R**EAL).

Real ist vom Typ REAL.

Funktionswert ist vom Typ LONG.

Mit den Funktionen LREAL und HREAL läßt sich eine 8-Byte-REAL-Zahl in zwei 4-Byte-LONG-Zahlen zerlegen. Die Funktion LLTOR fügt die beiden 4-Byte-Stücke wieder zusammen.

Diese Funktionen ermöglichen das Weiterverarbeiten von REAL-Zahlen in byte- oder bit-orientierter Form, was sonst mit REAL-Zahlen nicht möglich wäre. Nach der Aufspaltung in zwei LONG-Zahlen können nun auch REAL-Zahlen mit der Funktion NTOSS in Strukturen oder Records eingebaut werden.

Programmbeispiel:

```

' -----
' Name: LREAL.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  REAL R1,R2                             ' REAL-Variablen deklarieren
  LONG Hi,Lo                             ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  R1 = 12345.6789                         ' R1 vorbesetzen
  Lo = LREAL (R1)                         ' Low 32 Bit von R1 --> Lo
  Hi = HREAL (R1)                         ' High 32 Bit von R1 --> Hi
  R2 = LLTOR (Lo,Hi)                      ' Erzeuge R2 aus Lo und Hi
  USING "EF<0><1> <8>V0.0.0.0.0.0.1,8.0.0.0.0.0.0"
  PRINT USING #1, "<1>R1="; R1            ' zeige R1 im Original
  PRINT #1, "Lo="; Lo                     ' zeige Low 32 Bit von R1
  PRINT #1, "Hi="; Hi                     ' zeige High 32 Bit von R1
  PRINT USING #1, "R2="; R2               ' zeige R2 (Lo + Hi --> REAL)
END                                        ' Ende Task MAIN

```

Siehe auch: HREAL, LLTOR

LTR

LTR

LTR (Wert)

LTR: Typumwandlung: LONG to REAL

Funktion: Wandelt einen LONG-**Wert** in einen REAL-Wert um (**Long-To-Real**).

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ LONG.

Funktionswert ist vom Typ REAL.

Programmbeispiel:

```
' -----  
' Name: LTR.TIG  
' -----  
TASK MAIN                                ' Beginn Task MAIN  
  LONG L                                ' LONG-Variable deklarieren  
  REAL R                                ' REAL-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8  
  L = -234567                            ' L vorbesetzen  
  R = LTR (L)                            ' Umwandeln in REAL  
  PRINT #1, "<l>LONG ="; L                ' Ausgabe auf LC-Display  
  PRINT #1, "REAL ="; R                  ' Ausgabe auf LC-Display  
END                                       ' Ende Task MAIN
```

Siehe auch: RTL

MASK

MASK (BitNr)

MASK: Gibt Maske mit einem gesetzten Bit zurück

Funktion: Liefert eine Bitmaske zurück, bei der alle Bits gleich 0 sind und nur das Bit an der Position **BitNr** gleich 1 ist. Das niederwertigste Bit ist Bitnummer 0.

BitNr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt das auf 1 gesetzte Bit an. Das niederwertigste Bit ist Bitnummer 0.

Funktionswert ist vom Typ BYTE, WORD oder LONG und enthält das gesetzte Bit. Alle anderen Bits sind 0.

Mit der MASK.-Funktion kann auf einfache Weise eine Bitmaske erzeugt werden, die in Ausdrücken in Zusammenhang mit AND, OR oder den Instruktionen IN und OUT Verwendung findet.

MASK (1)

liefert als Funktionswert in einer BYTE-Variablen:

0	0	0	0	0	0	1	0

Programmbeispiel:

```

' -----
' Name: MASK.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
    LONG L, Value                        ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
Value = 1234567890                      ' Value vorbesetzen
FOR L = 0 TO 31                          ' Werte von 0 bis 31
    PRINT #1, "<1>Bit"; L; " is"         ' Ausgabe auf LC-Display
    IF (Value AND MASK (L)) = 0 THEN     ' Falls Bit L nicht gesetzt
        PRINT #1, "not ";               ' Ausgabe auf LC-Display
    ENDIF
    PRINT #1, "set."                    ' Ausgabe auf LC-Display
    WAIT DURATION 2000                  ' 2 Sek warten
NEXT                                     ' naechstes Bit pruefen
END                                     ' Ende Task MAIN

```

Siehe auch: IMASK

MAX_LEN

MAX_LEN

MAX_LEN (String)

MAX_LEN: Ermittelt die maximale String-Länge

Funktion: Liefert die maximale Länge eines Strings.

String ist eine Variable vom Typ STRING.

Länge ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und gibt die neue Länge des Strings an. Wertebereich 0...maximale Stringlänge.

MAX_LEN gibt Auskunft über die maximale Länge, die der Argument-String annehmen kann. Die Stringlänge ist begrenzt durch:

- die maximale Stringlänge in Tiger-BASIC® (32765 Zeichen).
- die gesetzte Standardlänge von Strings: Standard 64 bytes, setzbar mit USER_STRING_SIZE.
- die explizit gesetzte maximale Stringlänge: A\$(10) bedeutet, A\$ kann maximal 10 Bytes lang werden.

Programmbeispiel:

```
'-----  
' Name: MAX_LEN.TIG  
' gibt Auskunft ueber die maximale Laenge eines Strings  
'-----  
USER_STRING_SIZE 82                ' Standard: 64  
  
TASK MAIN                          ' Beginn Task MAIN  
  STRING A$                        ' Standard gesetzt: 82  
  STRING B$(1)                     ' fuer 1 Byte  
  STRING C$(32765)                 ' maximal moeglich  
  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  
  PRINT #1, "A$:";MAX_LEN ( A$ )   ' Auf LC-Display ausgeben  
  PRINT #1, "B$:";MAX_LEN ( B$ )  
  PRINT #1, "C$:";MAX_LEN ( C$ )  
END                                ' Ende Task MAIN
```

Siehe auch: SET_LEN\$, USER_STRING_SIZE

MID\$

MID\$ (*String*, *Start*, *Länge*)

MID\$: Mittlerer Teilstring

Funktion: Liefert einen Teil aus der Zeichenfolge **String** zurück.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition in der Zeichenfolge an, an der die Zeichen entnommen werden. Die erste Position in String ist Position 0.
Länge	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Zeichen an, die aus dem String entnommen werden.
Funktionswert	ist vom Typ STRING und enthält den Teil aus String, der mit der angegebenen Länge an der angegebenen Position entnommen worden sind.

Sowohl Start als auch Länge sind stets positiv. Es werden Zeichen bis maximal zum String-Ende entnommen.

Programmbeispiel:

```

' -----
' Name: MID$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "The BASIC-Tiger"                 ' String vorbesetzen
  B$ = MID$ (A$,4,5)                     ' Teilstring erzeugen
  PRINT #1, "A$: "; A$                   ' A$ ausgeben
  PRINT #1, "B$: "; B$                   ' B$ ausgeben
END                                       ' Ende Task MAIN

```

Siehe auch: LEFT\$, RIGHT\$, LEN, STOSS

MIRROR\$

MIRROR\$

MIRROR\$ (*String*)

MIRROR\$: Kehrt Reihenfolge der Zeichen in einem String um

Funktion: Liefert den gespiegelten **String** zurück, d.h. vertauscht die Reihenfolge der Zeichen im **String**.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.

Funktionswert ist vom Typ STRING und enthält die Bytes in umgekehrter Reihenfolge.

Programmbeispiel:

```
'-----  
' Name: MIRROR$.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, MIRROR$ ("RAIL LIVE")        ' Ausgabe auf LC-Display  
END                                       ' Ende Task MAIN
```

Siehe auch: BIT_MIRR, BYTE_MIRR

MOD

MOD (Wert1, Wert2)

MOD: Modulo, Rest einer Integer-Division

Funktion: Liefert den Rest der Integer-Division von **Wert1** / **Wert2** zurück.

Wert1 ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Wert2 ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
' -----
' Name: MOD.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "25 modulo 10 =" ; MOD(25,10) ' Ausgabe auf LC-Display
END                                       ' Ende Task MAIN
```

MODULO_INC

MODULO_INC

MODULO_INC (*Wert*, *Min*, *Max*, *Step*)

MODULO_INC: Modulo Increment, umlaufendes Increment/Decrement
(nur in Projektmodell 'PM_FULL')

Funktion: Inkrementiert oder dekrementiert den **Wert** innerhalb der angegebenen Grenzen **Min** und **Max** um den Wert **Step**.

3

Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.
Min	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den unteren, gerade noch zulässigen Wert an.
Max	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den oberen, gerade noch zulässigen Wert an.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite des Inkrements/Dekrements an. Negative Werte für Dekrementierung sind nur in LONG möglich.
Funktionswert	ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

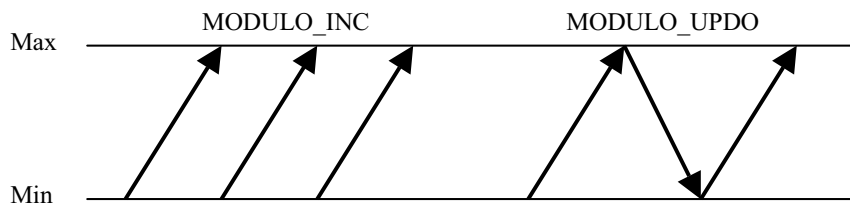
MODULO_INC verändert den Wert in der angegebenen Schrittweite und verhindert, dass der Wert außerhalb des angegebenen Zahlenbereichs liegt. Bei Überlauf über Max wird der Wert auf Min gesetzt und dann der Step-Inkrement ausgeführt. Bei Unterschreiten von Min wird der Wert auf Max gesetzt.

Beispiele:

A =	Operation	B =
19	B = MODULO_INC (A, 12, 22, 1)	20
20	B = MODULO_INC (A, 12, 22, 1)	21
21	B = MODULO_INC (A, 12, 22, 1)	22
22	B = MODULO_INC (A, 12, 22, 1)	13
A =	Operation	B =
15	B = MODULO_INC (A, 12, 22, -1)	14
14	B = MODULO_INC (A, 12, 22, -1)	13
13	B = MODULO_INC (A, 12, 22, -1)	12
12	B = MODULO_INC (A, 12, 22, -1)	21

3

Unterschied in der Zählweise zwischen MODULO_INC und MODULO_UPDO bei positivem Wert für Step:



MODULO_INC

Programmbeispiel:

```
' -----
' Name: MODULO_INC.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I, M
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  M = 18                                ' fange an mit M=18
  PRINT #1, "begin 18, inc 1:"
  FOR I = 0 TO 5                        ' 1 Zeile auf LCD4x20
    M = MODULO_INC ( M, 12, 22, 1 )    ' Inkrementiere
    PRINT #1, M;                       ' zeige M
  NEXT
  M = 16                                ' fange an mit M=16
  PRINT #1, "<13><10>begin 16, inc 2:"
  FOR I = 0 TO 5                        ' genug f. LCD4x20
    M = MODULO_INC ( M, 12, 22, 2 )    ' Inkrementiere
    PRINT #1, M;                       ' zeige M
  NEXT
END                                    ' Ende Task MAIN
```

Siehe auch: MODULO_UPDO, MOD, Unregelmäßige Zahlenfolge auf Seite 296

MODULO_UPDO

MODULO_UPDO (Startwert, Min, Max, Step, ID-Nr)

MODULO_UPDO: Umlaufendes Modulo-INC/DEC
(nur in Projektmodell 'PM_FULL')

Funktion: Richtet einen Parametersatz unter der angegebenen **ID-Nr** zur späteren Ausführung von MODULO_UPDO mit einer Variablen ein und führt einen Inkrement aus. Später inkrementiert oder dekrementiert MODULO_UPDO die Variable innerhalb der angegebenen Grenzen **Min** und **Max** um den Wert **Step**.

3

Startwert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Startwert fest. Da der Inkrement bereits einmal ausgeführt wird, entsteht direkt ein entsprechend veränderter Startwert.
Min	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den unteren, gerade noch zulässigen Wert an.
Max	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den oberen, gerade noch zulässigen Wert an.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite des Inkrements/Dekrements an. Negative Werte für Dekrementierung sind nur in LONG möglich.
ID-Nr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist eine Identifikationsnummer für den Parametersatz.
Funktionswert	ist vom Typ LONG und wird auf Startwert gesetzt. Eine Typwandlung bei der Zuweisung findet automatisch statt.

MODULO_UPDO setzt den Funktionswert auf den Startwert und richtet unter ID-Nr einen Parametersatz ein, in dem die Minimal- und Maximalwerte sowie die Schrittweite gespeichert werden. Es wird direkt ein Inkrement ausgeführt.

MODULO_UPDO wird später so aufgerufen:

Funktionswert = MODULO_UPDO (Wert, ID-Nr)

Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Wert, auf den die Operation angewendet wird.

MODULO_UPDO

ID-Nr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Identifikationsnummer des Parametersatzes, der verwendet wird.

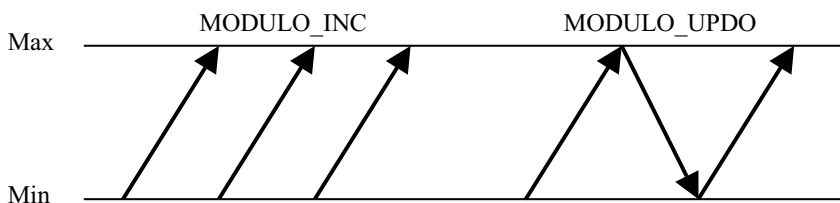
Funktionswert ist vom Typ LONG und wird um die Schrittweite Step des parametersatzes verändert. Eine Typwandlung bei der Zuweisung findet automatisch statt.

MODULO_UPDO verändert den Wert in der angegebenen Schrittweite und verhindert, dass der Wert außerhalb des angegebenen Zahlenbereichs liegt. Bei Überlauf über Max wird der Wert auf Max gesetzt und der nächste Step-Inkrement wird mit dem Stepwert mit umgekehrten Vorzeichen ausgeführt. Bei Unterschreiten von Min wird der Wert auf Min gesetzt und der nächste Step-Inkrement wird mit dem Stepwert mit umgekehrten Vorzeichen ausgeführt.

Beispiele:

Operation	A =
A = MODULO_UPDO (18, 12, 22, 1, 0)	19
A = MODULO_UPDO (A, 0)	20
A = MODULO_UPDO (A, 0)	21
A = MODULO_UPDO (A, 0)	22
A = MODULO_UPDO (A, 0)	21
A = MODULO_UPDO (A, 0)	20
A = MODULO_UPDO (114, 100, 200, -6, 1)	108
A = MODULO_UPDO (A, 1)	102
A = MODULO_UPDO (A, 1)	100
A = MODULO_UPDO (A, 1)	106
A = MODULO_UPDO (A, 1)	112
A = MODULO_UPDO (A, 1)	118

Unterschied in der Zählweise zwischen MODULO_INC und MODULO_UPDO bei positivem Wert für Step:



Programmbeispiel:

```

-----
' Name: MODULO_UPDO.TIG
-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I, M, N
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  M = MODULO_UPDO ( 18, 12, 22, 1, 0 ) ' setze Parametersatz 0 fuer
  N = MODULO_UPDO ( 16, 12, 22, 2, 1 ) ' setze Parametersatz 1 fuer
  PRINT #1, "begin 18, inc 1:"          ' folgende MODULO UPDOs
  FOR I = 0 TO 5                        ' 1 Zeile auf LCD4x20
    M = MODULO_UPDO ( M, 0 )            ' Parametersatz 0
    PRINT #1, M;                        ' zeige M
  NEXT
  PRINT #1, "<13><10>begin 16, inc 2:"
  FOR I = 0 TO 5                        ' genug f. LCD4x20
    N = MODULO_UPDO ( N, 1 )            ' Parametersatz 1
    PRINT #1, N;                        ' zeige M
  NEXT
END                                    ' Ende Task MAIN

```

3

Siehe auch: MODULO_INC, MOD, Unregelmäßige Zahlenfolge auf Seite 296

Unregelmäßige Zahlenfolge

Konstruktionen wie $A=A+1$ sowie die Instruktionen MODULO_INC, MODULO_UPDO ermöglichen das regelmäßige Durchzählen von Zahlen. Manchmal jedoch will man in einer Folge eine Zahl auslassen oder eine ganz unregelmäßige Folge durchzählen. Dann bietet sich folgende Konstruktion an, die über eine Stringkonversion im Zahlenbereich von 0 bis 255 funktioniert:

x = asc (convert\$ (chr\$(y), "07 06 04 03 02"%))

3

Y wird in einen String gewandelt, um in der Funktion CONVERT\$ verwendet werden zu können. Da am Ende eine Zahl benötigt wird, ist der eigentlich wandelnde Ausdruck in ASC eingebettet. CONVERT\$ liefert dann für Y=0 bis Y=4 die Werte 7, 6, 4, 3, 2 (unter Auslassung der 5).

Programmbeispiel:

```
'-----
' Name: UNREG.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE X, Y
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR Y = 0 TO 4                          ' um Folge zu erzeugen
    X = ASC ( CONVERT$ ( CHR$(Y), "07 06 04 03 02"% ) )' konvertiere
    PRINT #1, X;                          ' zeige X
  NEXT
END                                        ' Ende Task MAIN
```

NFROMS

NFROMS (*String*, *Position*, *Anzahl*)

NFROMS: Schnelle Ausführung: Numerischer Wert (Integer) aus String.

Funktion: 'Num-From-String' liest **Anzahl** Bytes aus **String** an der angegebenen **Position** und bildet daraus eine BYTE, WORD oder LONG-Zahl.

- **Anwendung:** Datenspeicherung (Flash) oder Datenübertragung (seriell) in Form von Datensätzen, die verschiedene Datentypen in kompakter Form zusammenhalten.
- **Vorteil:** Platz-- und Rechenzeit-sparende Funktion, um in Basic mit strukturierten Datensätzen zu arbeiten.
- **Besonderheiten:** Die Stringlänge wird automatisch an die Bedürfnisse angepasst und ein 'random'-Zugriff auf die Elemente ist möglich. Im String dürfen undefinierte Bereiche existieren.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich entweder von -4 bis -1 oder von 1 bis 4 und gibt die Anzahl der zu entnehmenden Bytes an.
Funktionswert	ist vom Typ LONG und enthält die aus dem String gelesene Zahl.

NFROMS entnimmt einem String eine angegebene Anzahl von Bytes und bildet daraus eine Zahl. Wird die Anzahl positiv angegeben, liest NFROMS 'low-Byte-first', bei negativer Anzahl 'high-Byte-first'. Ist die Anzahl zu groß, werden die übrigen Bytes ignoriert. Bytes, die außerhalb der initialisierten Stringlänge liegen, werden als 0 gelesen.

Die Funktionen NTOSS\$, NFROMS, RTOSS\$, RFROMS, STOSS\$ erhöhen die Verarbeitungsgeschwindigkeit und sparen Speicherplatz, wenn Zahlenwerte zwischengespeichert oder übertragen werden sollen. Der Ablage-String entspricht in etwa einer Struktur in C.

NFROMS

Programmbeispiel:

```
'-----
' Name: NFROMS.TIG
' transferiert aus einem String Bytes in eine numerische Variable
'-----
TASK MAIN                                ' Beginn Task MAIN
STRING A$                                ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "UH<8><8> k 0.0.0.0.8"                ' Format HEX-Ausgabe
A$ = "ABC<10h><11h>A<12h>DE"              ' A$: 414243101141124445
PRINT USING #1, NFROMS ( A$, 3, 4 )      ' ab Pos.3, 4 Byte, lo first
PRINT USING #1, NFROMS ( A$, 3, -4 )     ' ab Pos.3, 4 Byte, hi first

A$ = ""                                  ' jenseits d. (initialisierten)
                                           ' Stringlaenge: Wert = 0
PRINT USING #1, NFROMS ( A$, 0, 4 )      ' ab Pos.0, 4 Byte, low first
                                           ' jenseits d. max. Stringl.:0
PRINT USING #1, NFROMS ( A$, 100, 4 )    ' ab Pos.100, 4 Byte, low first

SET_LEN$ ( A$, 2 )                       ' Stringlaenge ist nur 2
PRINT USING #1, NFROMS ( A$, 0, 4 )      ' ab Pos.0, 4 Byte, low first

SET_LEN$ ( A$, 4 )                       ' Bytes sind unveraendert
PRINT USING #1, NFROMS ( A$, 0, 4 )      ' ab Pos.0, 4 Byte, low first
END                                        ' Ende Task MAIN
```

Siehe auch inverse-Funktion: NTOSS\$ und RFROMS, RTOSS

NTOS\$

NTOS\$ (*String*, *Position*, *Anzahl*, *Wert*)

NTOS\$: Schnelle Ausführung: Numerischer Wert (Integer) nach String.

Funktion: 'Num-To-String' legt **Anzahl** Bytes des **Wertes** in den **String** an die angegebene **Position**. Die Bytes werden ohne Wandlung geschrieben.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition im String an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der abzulegenden Bytes an.
Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist das Element, aus dem Bytes in den String eingefügt werden.
Funktionswert	ist vom Typ STRING und enthält den String mit der eingefügten Zahl.

NTOS\$ legt in einem String eine angegebene Anzahl von Bytes aus der Zahl ab. Wird die Anzahl positiv angegeben, legt NTOS\$ die Bytes 'low-Byte-first', bei negativer Anzahl 'high-Byte-first' ab. Ist die Anzahl zu groß, werden die übrigen Bytes als 0-Bytes geschrieben.

Beachte: NTOS\$ verlängert den String, wenn die angegebene Position außerhalb des Strings liegt. Die maximale Länge wird jedoch nie überschritten. Es kann dadurch auch eine "Lücke" im String entstehen, die undefinierte Bytes enthält.

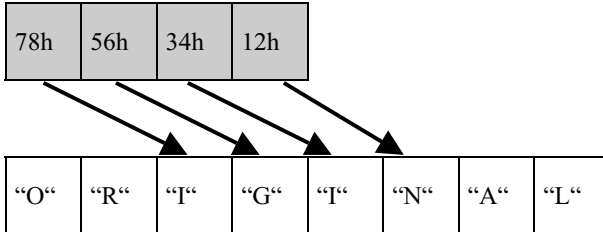
NTOS\$ verlängert den String nicht? Dann prüfen Sie nach, ob der String initialisiert ist.

NTOS\$

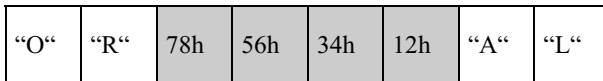
Die Funktionen NTOS\$, NFROMS, RTOS\$, RFROMS, STOS\$ erhöhen die Verarbeitungsgeschwindigkeit und sparen Speicherplatz, wenn Zahlenwerte zwischengespeichert oder übertragen werden sollen. Der Ablage-String entspricht in etwa einer Struktur in C, in der verschiedene Datentypen abgelegt werden können.

Folgendes Beispiel mit A\$ = "ORIGINAL" und N = 12345678h

A\$ = NTOS\$ (A\$, 2, 4, N)

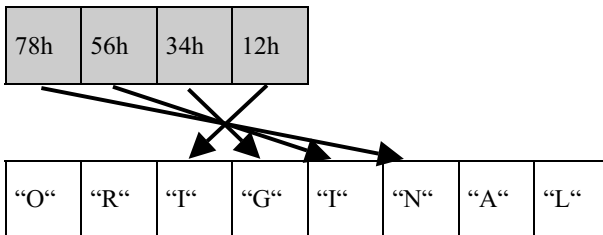


Ergebnis:

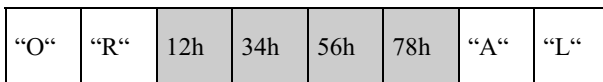


Und nun mit negativer Angabe der Anzahl:

A\$ = NTOS\$ (A\$, 2, -4, N)



Ergebnis:



Und mit A\$ = "" vor der Funktion:

A\$ = NTOS\$ (A\$, 2, 4, N)

78h	56h	34h	12h

Ergebnis 6 Bytes:

x	x	78h	56h	34h	12h

x= Bytes haben zufälligen Inhalt

NTOS\$

Programmbeispiel:

3

```
'-----
' Name: NTOS$.TIG
'-----

TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
  LONG N                                 ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
N = 12345678h                            ' N vorbesetzen
A$ = "Original"                          ' A$ vorher
CALL SHOW_STRING_AS_HEX ( A$ )
A$ = NTOS$ ( A$, 3, 4, N )                ' Zahl einbauen erst low Byte
CALL SHOW_STRING_AS_HEX ( A$ )
A$ = NTOS$ ( A$, 3, -4, N )               ' Zahl umgekehrt einbauen
CALL SHOW_STRING_AS_HEX ( A$ )

                                           ' B$ nicht initialisiert:
                                           ' Laenge waere Zufall!
B$ = ""                                  ' hier: leer
B$ = NTOS$ ( B$, 0, -4, N )               ' Zahl umgekehrt einbauen
PRINT #1, "<1>";LEN ( B$ )                ' Laenge ist 4
CALL SHOW_STRING_AS_HEX ( B$ )            ' Bytes sind geschrieben

                                           ' hier: leer
B$ = NTOS$ ( A$, 1, -4, N )               ' Zahl umgekehrt einbauen
PRINT #1, LEN ( B$ )                     ' Laenge ist 8(von A$)
CALL SHOW_STRING_AS_HEX ( B$ )            ' Bytes sind geschrieben
END                                        ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----

SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2"              ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1             ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                            ' neue Zeile
END
```

Siehe auch inverse-Funktion: NFROMS, STOSS

ON_ERROR_RESET

ON_ERROR_RESET

ON_ERROR_RESET: Setzt Fehler-Flag zurück nach der Bearbeitung

Instruktion: Setzt das Error-Flag nach einer Verzweigung in eine Fehlerbehandlung zurück.

Das Error-Flag muß zurückgesetzt werden, um die Funktion der Fehlerbehandlung zu erhalten.

Programmbeispiel:

3

```

' -----
' Name: ON_ERR_RESET.TIG
' -----
' Alle Bits der Port 23 bis 15 (zaehlt abwaerts) werden der Reihe
' nach auf HIGH und wieder auf LOW gesetzt. Bei Port = 15 (Fehler,
' Port exist. nicht) wird in das Unterprogramm Error gesprungen,
' der Fehler angezeigt und zum Hauptprogramm zurueckgesprungen.
' -----
BYTE I, J                                ' BYTE-Variablen deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
ON ERRTASK CALL Error                    ' Unterprogramm bei Fehler
FOR J=23 TO 15 STEP -1                  ' Zaehlschleife fuer J
  I = 128                                ' Variable I initialisieren
  LOOP 8                                ' 8 Schleifendurchlaeufer
    PRINT #1, "<l>Port: "; J;              ' Port-Nr. ausgeben
    PRINT #1, " - Bit: "; LD(I)          ' Bit-Nr. ausgeben
    OUT J, 255, I                        ' I auf Port J ausgeben
    I = I SHR 1                          ' Bitweise rechts verschieben
    WAIT_DURATION 200                    ' 200 ms warten
  ENDLOOP                                ' Ende der Schleife
NEXT J                                  ' naechster Wert
END                                      ' Ende Task MAIN

' -----
' Unterprogramm zur Fehlerbehandlung
' -----
SUB Error                                ' Beginn Unterprogramm Error
PRINT #1, "Error!"                      ' Ausgabe des Fehlers
PRINT #1, SYSVAR$(0,0)                  ' auf dem LC-Display
WAIT_DURATION 3000                      ' 3 Sek warten
ON_ERROR_RESET                          ' Error-Flag zuruecksetzen
END                                      ' Ende Unterprogramm Error

```

ON_ERRTASK_CALL

ON_ERRTASK_CALL

ON_ERRTASK_CALL *Name*

ON_ERRTASK_CALL: Ruft Unterprogramm auf bei Fehler in Task

Instruktion: Im Fall eines Laufzeitfehlers wird das Unterprogramm **Name** aufgerufen. Jede Task kann ihre eigene Fehlerbehandlung haben.

3

Das Unterprogramm wird nicht aufgerufen, wenn der Fehler in einer Instruktion mit expliziter Fehlerbehandlung aufgetreten ist (z.B.: POKE_FLASH, PUT_FIFO, ...).

Die Art des Fehlers kann im Unterprogramm durch Abfragen von Systemparametern festgestellt werden (siehe Funktionen SYSVARx ab Seite 458). Die Run-Time-Fehlerbehandlung dient im Wesentlichen dazu, während der Entwicklungsphase Laufzeitfehler zu erkennen und durch entsprechende Programmierung zu vermeiden. Bei abgeschlossener Entwicklung sollten keine Laufzeitfehler mehr auftreten.

Am Ende der Fehlerbehandlung muß das Fehlerflag mit ON_ERROR_RESET zurückgesetzt werden.



Weitere Informationen zur Fehlerbehandlung während der Laufzeit finden Sie auf Seite 50ff.

Programmbeispiel:

```

'-----
' Name: ON_ERRTASK_CALL.TIG
'-----
' Alle Bits der Port 23 bis 15 (zaehlt abwaerts) werden der Reihe
' nach auf HIGH und wieder auf LOW gesetzt. Bei Port = 15 (Fehler,
' Port exist. nicht) wird in das Unterprogramm Error gesprungen,
' der Fehler angezeigt und zum Hauptprogramm zurueckgesprungen.
'-----
BYTE I, J                                ' BYTE-Variablen deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
ON_ERRTASK_CALL Error                    ' Unterprogramm bei Error
FOR J=23 TO 15 STEP -1                  ' Zaehlschleife fuer J
  I = 128                               ' Variable I initialisieren
  LOOP 8                                ' 8 Schleifendurchlaeufer
    PRINT #1, "<1>Port: "; J;            ' Port-Nr. ausgeben
    PRINT #1, " - Bit: "; LD(I)         ' Bit-Nr. ausgeben
    OUT J, 255, I                        ' I auf Port J ausgeben
    I = I SHR 1                          ' Bitweise rechts verschieben
    WAIT DURATION 200                    ' 200 ms warten
  ENDLOOP                                ' Ende der Schleife
NEXT                                     ' naechster Wert
END                                       ' Ende Task MAIN

'-----
' Unterprogramm zur Fehlerbehandlung
'-----
SUB Error                                ' Beginn Unterprogramm Error
  PRINT #1, "Error!"                     ' Ausgabe des Fehlers
  PRINT #1, SYSVAR$(0,0)                 ' auf dem LC-Display
  WAIT DURATION 3000                     ' 3 Sek warten
  ON_ERROR_RESET                         ' Error-Flag zuruecksetzen
END                                       ' Ende Unterprogramm Error

```

ON_ERRTASK_GOTO

ON_ERRTASK_GOTO

ON_ERRTASK_GOTO Name

ON_ERRTASK_GOTO: Springt zu Sprungmarke bei Fehler in Task

Instruktion: Im Fall eines Fehlers wird an die Stelle **Name** verzweigt, wenn der Fehler nicht in einer Instruktion mit expliziter Fehlerbehandlung aufgetreten ist.

Die Art des Fehlers kann durch Abfragen von Systemparametern festgestellt werden (siehe Funktionen SYSVARx ab Seite 458).

Nach der Fehlerbehandlung muß das Fehlerflag mit ON_ERROR_RESET zurückgesetzt werden.

Weitere Informationen zur Fehlerbehandlung während der Laufzeit finden Sie auf Seite 50ff.

Programmbeispiel:

```
'-----
' Name: ON_ERR_TASK_GOTO.TIG
'-----
' Alle Bits der Port 23 bis 15 (zaehlt abwaerts) werden der Reihe
' nach auf HIGH und wieder auf LOW gesetzt. Bei Port = 15 (Fehler,
' Port exist. nicht) wird zum Label "Error" verzweigt und der
' Fehler angezeigt. Danach ist das Programm beendet.
'-----
BYTE I, J                                ' BYTE-Variablen deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
ON_ERRTASK_GOTO Error                    ' Sprung bei Fehler
FOR J=23 TO 15 STEP -1                    ' Zaehlschleife fuer J
  I = 128                                ' Variable I initialisieren
  LOOP 8                                  ' 8 Schleifendurchlaeufer
    PRINT #1, "<1>Port: "; J;              ' Port-Nr. ausgeben
    PRINT #1, " - Bit: "; LD(I)           ' Bit-Nr. ausgeben
    OUT J, 255, I                          ' I auf Port J ausgeben
    I = I SHR 1                             ' Bitweise rechts verschieben
    WAIT_DURATION 200                       ' 200 ms warten
  ENDLOOP                                  ' Ende der Schleife
NEXT J                                    ' naechster Wert
Error:                                    ' Ab hier Fehlerbehandlung
  PRINT #1, "Error!"                        ' Ausgabe des Fehlers
  PRINT #1, SYSVAR$(0,0)                   ' auf dem LC-Display
  WAIT_DURATION 3000                       ' 3 Sek warten
  ON_ERROR_RESET                           ' Error-Flag zuruecksetzen
END                                         ' Ende Task MAIN
```

OR\$

OR\$ (*String*, *Bitmuster*)

OR\$: Maskiert alle Bytes eines Strings per OR
(nur in Projektmodell 'PM_FULL')

Funktion: Führt mit allen Bytes in **String** eine OR-Operation aus.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu bearbeitenden Zeichen.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält das 8-Bit breite Bitmuster für die OR-Operation.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Zeichen in **String** werden byteweise mit einer OR-Operation bearbeitet. Dabei werden alle Bits, die im Bitmuster auf '1' stehen, im Zielbyte auf '1' gesetzt. Alle Bits, die im Bitmuster auf '0' stehen, werden aus dem Quell-Byte ins Zielbyte übernommen.

Beispiel:

Quellstring (2 Bytes):	01100111	01100110
Bitmuster:	01010101	01010101
Ergebnis-String:	01110111	01110111

OR\$

Programmbeispiel:

```
' -----
' Name: OR$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "<67H><66H>"                      ' das is A$
  PRINT #1, "0:";                        ' zeige A$
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = OR$ ( A$, 55H )                   ' A$ OR 55H
  PRINT #1, "0:";
  CALL DISP_BIN ( 2, "<55H><55H>" )        ' zeige AND-Maske
  PRINT #1, "1:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
END                                       ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )            ' zeige CNT Bytes binaer
  BYTE B, I, J                          ' lokale Variable
  FOR I = 1 TO CNT                      ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ) ) ' hole ein Byte
    FOR J = 1 TO 8                      ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN          ' teste ob bit 1 ist
        PRINT #1, "1";                 ' drucke Bit
      ELSE                              ' falls kein Rest
        PRINT #1, "0";                 ' drucke Bit
      ENDIF
      B = B SHL 1                       ' B eins nach rechts (neues Bit)
    NEXT J
    PRINT #1, " ";                      ' Leerzeichen zwischen Bytes
  NEXT I
END                                       ' Ende Unterprogramm
```

Siehe auch: ADD\$, AND\$, BIT_MIRR\$, XOR\$, Operator BIT_OR

OUT

OUT *Log_Portadr., Maske, Wert*

OUT: Schreibt Bits auf I/O-Port

Instruktion: Gibt **Wert** maskiert durch **Maske** an den internen oder externen Port mit der Adresse **Log_Portadr.** aus.

Log_Portadr. ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die logische Portadresse an.

Maske ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und maskiert die auszugebenden Bits. 0-Bits in der Maske verhindern die Wirkung der OUT-Instruktion auf den entsprechenden Bits.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Maske unterbindet die Wirkung der Instruktion an einzelnen Bits. Ein 0-Bit in der Maske bewirkt, daß das entsprechende Port-Bit sich nicht ändert.

Maskenbit	Bit am I/O-Port
0	keine Änderung.
1	Bit wird ausgegeben, wie in Wert festgelegt.



Sowohl das erweiterte Portsystem als auch Device-Treiber können Portbits gegen Zugriff schützen, so daß diese dann nicht mehr durch IN- und OUT-Instruktionen angesprochen werden können. In einem solchen Fall wird im Debug-Modus am PC eine fehlermeldung angezeigt, im Run-Modus läuft das Programm weiter.

Es werden maximal soviele Bits wirksam, wie an dem entsprechenden Port vorhanden sind.

Siehe auch 'Die Softwareseite der erweiterten I/O-Ports' im Hardware-Handbuch.

OUT

Programmbeispiel:

```
'-----  
' Name: OUT.TIG  
' Ausgabe auf interne und erweiterte Ports  
' Beachte: Port 6 scheint zu blinken, da er fuer die erweiterten  
' Ports als Bus verwendet wird  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
  DIR_PORT 8,00000000b                  ' interner Port 8 als Ausgang  
                                          ' externe Ports sind fix  
  LOOP 9999999                          ' sehr viele Loops  
    OUT 8,255,255                        ' Int. I/O Port 8 setzen  
    OUT 10h,255,0                       ' Ext. I/O Port 10h loeschen  
    WAIT_DURATION 200                   ' 200ms warten  
                                          ' benutze Maske  
    OUT 8,0AAh,0                        ' Int. I/O Port 8 auf 0 setzen  
    OUT 10h,255,255                    ' Ext. I/O Port 10h setzen  
    WAIT_DURATION 200                   ' 200ms warten  
  ENDLOOP                               '  
END                                     ' Ende Task MAIN
```

Siehe auch: LL_IPORT_OUT, XOUT, XBUS_OUTR\$

PARITY

PARITY (Wert, Pos, Len,)

PARITY: Berechnet Parity eines numerischen Wertes
(nur in Projektmodell 'PM_FULL')

Funktion: Liefert Paritäts-Information von **Wert** ab der Position **Pos** über die Anzahl Bits **Len**.

Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Wert, über den die Parity-Information gebildet wird.
Pos	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position des Bits an, ab dem die Parity-Prüfung beginnt.
Len	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der zu berücksichtigenden Bits fest.
Funktionswert	ist vom Typ LONG und enthält 0: wenn die Anzahl der ,1'-Bits ungerade ist 1: wenn die Anzahl der ,1'-Bits gerade ist.

3

PARITY

PARITY zählt die ,1'-Bits im angegebenen Bereich einer numerischen Variablen und liefert als Funktionswert 0, wenn die Anzahl der ,1'-Bits ungerade ist, und 1, wenn die Anzahl der ,1'-Bits gerade ist.

Programmbeispiel:

3

```
'-----
' Name: PARITY.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  PRINT #1, "(00000000,0,8)=>"; PARITY ( 00000000, 0, 8 )
  PRINT #1, "(00001000,0,8)=>"; PARITY ( 00001000, 0, 8 )

  PRINT #1, "(00001000,0,8)=>"; PARITY ( 00001000, 0, 8 )
  PRINT #1, "(00001000,0,1)=>"; PARITY ( 00001000, 0, 1 )
  PRINT #1, "(00001000,0,2)=>"; PARITY ( 00001000, 0, 2 )
  PRINT #1, "(00001000,0,3)=>"; PARITY ( 00001000, 0, 3 )
  PRINT #1, "(00001000,0,4)=>"; PARITY ( 00001000, 0, 4 )

  PRINT #1, "(11110000,0,8)=>"; PARITY ( 11110000, 0, 8 )
  PRINT #1, "(11110000,2,3)=>"; PARITY ( 11110000, 2, 3 )

  PRINT #1, "(11110000,2,1)=>"; PARITY ( 11110000, 2, 3 )
  PRINT #1, "(11110000,2,2)=>"; PARITY ( 11110000, 2, 3 )
  PRINT #1, "(11110000,2,3)=>"; PARITY ( 11110000, 2, 3 )
  PRINT #1, "(11110000,2,4)=>"; PARITY ( 11110000, 2, 3 )

END                                     ' Ende Task MAIN
```

Siehe auch: BIT, SET_BIT

PACK_BYTE\$

PACK_BYTE\$ (String, Start, Anzahl, Flag, Typ)

PACK_BYTE\$: Komprimiert Strings
(nur in Projektmodell 'PM_FULL')

Funktion: Komprimiert die angegebene **Anzahl** Bytes aus **String** ab der Position **Start**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu komprimierenden Zeichen.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition der Komprimierung an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der zu komprimierenden Bytes fest.
Flag	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 0...255 und legt das Kompressionsflag fest.
Typ	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Kompressionsart (zur Zeit ist Typ immer = 0).
Funktionswert	ist vom Typ STRING und enthält den komprimierten String.

PACK_BYTE\$ führt eine ein-eindeutige Abbildung aus, die nach dem entpacken mit UNPACK_BYTE\$ wieder zu dem identischen Ergebnis führt.

PACK_BYTE\$ mit Kompressionstyp 0: ersetzt aufeinanderfolgende gleiche Codes durch folgende Byte-Sequenz mit dem Wertebereich für Anzahl von 4 bis 255:

<Flag><Anzahl><Code>

Es gibt folgende Sondersequenzen:

Sequenz	Bedeutung
<Flag><0>	Endesequenz
<Flag><1>	1x Flag
<Flag><2>	2x Flag
<Flag><3>	3x Flag

PACK_BYTE\$

Programmbeispiel:

```
'-----
' Name: PACK_BYTE$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$ = "AAAAAAAAAAAAAAAAAAAAAAAAAAAA"   ' das is A$ (25xA)
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO 4                         ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  B$ = PACK_BYTE$ ( A$, 0, 5, 11H, 0 )   ' komprimiere 5 Bytes ab Pos. 0
  PRINT #1, "<13><10>B$=";                ' zeige Ergebnis in B$
  FOR I = 0 TO LEN(B$)-1                 ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(B$,I,1));
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                      ' Ende Task MAIN
```

Siehe auch: UNPACK_BYTE\$, BYTE_ANIB\$, Incremental-Conversion
BYTE_INIB\$, INIB_BYTE\$

PBCD_ASC\$

PBCD_ASC\$ (*String*)

PBCD_ASC\$: Konvertiert: packed BCD -> ASCII-HEX
(nur in Projektmodell 'PM_FULL')

Funktion: Konvertiert alle Nibbles in **String** in das ASCII-Format im Ergebnis-String.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden Zeichen im beiden Nibbles.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Nibbles in **String** werden in den ASCII-Code umgesetzt. Die Werte 10...15 werden in die HEX-Zeichen 'A'...'F' umgesetzt. Am Ende enthält der Zielstring doppeltsoviele Bytes wie der Quellstring.

Beispiel:

Quellstring (1 Byte):	01001111	
Ergebnis-String (2 Bytes):	4	F
Binäre Darstellung:	34H (00110100)	46H (01000110)

PBCD_ASC\$

Programmbeispiel:

```
'-----
' Name: PBCD_ASC$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$ = "<4FH>"                          ' das is A$
  PRINT #1, "A$=";"    HEX:";          ' zeige A$
  PRINT_USING #1, ASC(A$)
  B$ = PBCD_ASC$ ( A$ )                 ' wandele nach ASCII
  PRINT #1, "after PBCD_ASC$"
  PRINT #1, "B$=";B$;"=HEX:";          ' zeige Ergebnis in B$
  PRINT_USING #1, ASC(LEFT$(B$,1));" ";ASC(RIGHT$(B$,1))
END                                      ' Ende Task MAIN
```

Siehe auch: ASC_BCD\$, ASC_PBCD\$, BCD_ASC\$, BCD_PBCD\$, PBCD_BCD\$

PBCD_BCD\$

PBCD_BCD\$ (String)

PBCD_BCD\$: Konvertiert: „packed BCD“ -> BCD
(nur in Projektmodell 'PM_FULL')

Funktion: Konvertiert alle Nibbles in **String** in das 'binary-coded-digit'-Format im Ergebnis-String. Jedes Byte im Ergebnis-String enthält nur ein BCD-codierte Nibble.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu konvertierenden Nibbles.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Nibbles in **String** werden in die unteren Nibbles des Ergebnis-Strings kopiert. Die Bytes im Zielstring enthalten nur in ihren unteren Nibbles BCD-codierte Zeichen. Am Ende enthält der Zielstring doppeltsoviele Bytes wie der Quellstring.

Beispiel:

Quellstring (1 Byte):	01001111	
Ergebnis-String (2 Bytes):	00000100	00001111

Programmbeispiel:

```

' -----
' Name: PBCD_BCD.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"          ' fuer Ausgabe als HEX
  A$ = "<4FH>"                          ' das is A$
  PRINT #1, "A$=";"  HEX:";            ' zeige A$
  PRINT_USING #1, ASC(A$)
  B$ = PBCD_BCD$ ( A$ )                  ' wandele nach BCD
  PRINT #1, "after PBCD_BCD$"
  PRINT #1, "B$=";"  HEX:";            ' zeige Ergebnis in B$
  PRINT_USING #1, ASC(LEFT$(B$,1));" ";ASC(RIGHT$(B$,1))
END                                       ' Ende Task MAIN

```

Siehe auch: ASC_BCD\$, ASC_PBCD\$, PBCD_ASC\$, PBCD_BCD\$

PEEK_FLASH

PEEK_FLASH *Adresse, Variable, Anzahl[, Fehlerbehandlung]*

PEEK_FLASH: Liest Daten aus FLASH in Variable ein (standard)

PEEK_FLASH: Liest Daten aus FLASH in Variable ein (high-speed)
(nur in Projektmodell 'PM_FULL')

3

Instruktion: Liest **Anzahl** Bytes aus dem Flash-Speicher von **Adresse** in die **Variable**.

Adresse ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsadresse an, ab der die Bytes gelesen werden.

Variable ist vom Typ BYTE, WORD, LONG, REAL oder STRING.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der zu lesenden Bytes an. Es werden maximal so viel Bytes gelesen, wie die Variable aufnehmen kann.

Diese Instruktion kann ihre eigene **Fehlerbehandlung** in Form eines Unterprogramms oder einer Verzweigung haben. Schreibweise:

ON_ERROR_CALL *Unterprogramm*

ON_ERROR_GOTO *Sprungmarke*

Wenn es FLASH-Sektoren gibt, die nicht vom Programm-Code belegt werden, stehen diese einem Tiger-BASIC® Programm als Daten-Speicher frei zur Verfügung. Der benutzbare Flash-Speicher beginnt für den Anwender immer bei Adresse 0. Es stehen immer komplette Sektoren zur Verfügung, die byteweise beschrieben und gelesen werden können. Das Löschen erfolgt stets Sektorweise.

Weitere Informationen über den Flashspeicher finden Sie auf im Hardware-Handbuch.

Programmbeispiel:

```

'-----
' Name: PEEK_FLASH.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE I,J                               ' BYTE-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 100                        ' For...Next Schleife
    POKE_FLASH I,2*I,0                   ' 2*I ins Flash schreiben
  NEXT                                    ' For...Next-Schleife Ende
  FOR I = 0 TO 100                        ' For...Next Schleife
    PEEK_FLASH I,J,1                     ' 1 Byte aus Flash auslesen
    PRINT #1, "<1>J ="; J                 ' Ausgabe auf LC-Display
    WAIT DURATION 500                    ' 500 ms warten
  NEXT                                    ' For...Next Schleife Ende
END                                        ' Ende Task MAIN

```

3

POKE_FLASH

POKE_FLASH

POKE_FLASH Adresse, Wert, Index[, Fehlerbehandlung]

POKE_FLASH: Schreibt Daten in Daten-FLASH Bereich (standard)

Instruktion: Schreibt ein durch **Index** angegebenes Byte von **Wert** an **Adresse** in den Flash-Speicher. Die Zelle muß nicht unbedingt gelöscht sein. Es werden 1-Bits (=gelöschte Bits) zu 0-Bits geschrieben, nie können jedoch 0-Bits zu 1-Bits werden.

3

Adresse ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsadresse an, ab der die Bytes geschrieben werden.

Wert ist vom Typ BYTE, WORD, LONG, REAL oder STRING.

Index ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welcher 8-Bit-Anteil von Wert geschrieben wird. Bei numerischem Wert bedeutet Index 0 das niederwertigste Byte. Beim Datentyp STRING ist Index 0 das erste Byte des Strings.

Datentyp	mögliche Werte für Index
BYTE	0
WORD	0,1
LONG	0,1,2,3
REAL	0,1,2,3,4,5,6,7
STRING	0...LEN-1

Diese Instruktion kann ihre eigene **Fehlerbehandlung** in Form eines Unterprogramms oder einer Verzweigung haben. Schreibweise:

ON_ERROR_CALL Unterprogramm

ON_ERROR_GOTO Sprungmarke

Wenn es FLASH-Sektoren gibt, die nicht vom Programm-Code belegt werden, stehen diese einem Tiger-BASIC® Programm als Daten-Speicher frei zur Verfügung. Der benutzbare Flash-Speicher beginnt für den Anwender immer bei Adresse 0. Es stehen immer komplette Sektoren zur Verfügung, die byteweise beschrieben und gelesen werden können. Das Löschen erfolgt stets Sektorweise.

Weitere Informationen über den Flashspeicher finden Sie im Hardware-Handbuch.

Tip: Die Funktion **POKEM_FLASH** ersetzt die ältere Instruktion POKE_FLASH und ist in der Ausführung schneller.

Programmbeispiel:

```

' -----
' Name: POKE_FLASH.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG I,VALUE                          ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  USING "UH<8><1> k 0.0.0.0.8"          ' Format fuer PRINT USING
  VALUE = 12345678h                    ' Value vorbesetzen
  FOR I = 0 TO 3                        ' For...Next Schleife
    POKE_FLASH I,VALUE,I               ' Vier 8-Bit-Anteile schreiben
  NEXT                                ' For...Next-Schleife Ende
  PEEK_FLASH 0,VALUE,4                 ' Ein LONG aus Flash auslesen
  PRINT USING #1, "<1>Value = "; VALUE  ' Ausgabe auf LC-Display
END                                    ' Ende Task MAIN

```

3

Siehe auch: POKEM_FLASH

POKEM_FLASH

POKEM_FLASH (Adresse, W, Index, Anz, Mode)

POKEM_FLASH: Schreibt Daten in Daten-FLASH Bereich (high-speed)
(nur in Projektmodell 'PM_FULL')

Funktion: Schreibt **Anzahl** Bytes aus **W** ab der angegebenen **Position** an **Adresse** in den Flash-Speicher. Die Zelle muß nicht unbedingt gelöscht sein. Es werden 1-Bits (=gelöschte Bits) zu 0-Bits geschrieben, nie können jedoch 0-Bits zu 1-Bits werden.

Adresse ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsadresse an, ab der die Bytes geschrieben werden.

W ist vom Typ BYTE, WORD, LONG, REAL oder STRING.

Datentyp	mögliche Werte für Index
BYTE	0
WORD	0,1
LONG	0,1,2,3
REAL	0,1,2,3,4,5,6,7
STRING	0...LEN-1

Anz ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Bytes an, die geschrieben werden.

Mode ist vom Typ BYTE, WORD, LONG und bestimmt den Schreibmodus:
0 = Flashzellen müssen gelöscht sein (0FFh).
1 = Es werden Bits von „1“ zu „0“ geschrieben, wenn nötig.
Bits von „0“ zu „1“ zu schreiben führt zu einem Fehler.

Datenbyte	FLASH-Byte	FLASH anschliessend
1111 0000	1111 1001	1111 0000 OK
0011 0000	1111 1001	0011 0000 OK
0001 0000	1111 1001	0001 0000 OK
1111 0111	1111 1001	1111 0### Fehler

2: Dies führt niemals zu einem Schreibfehler. Es werden bitweise „0“-Bits geschrieben, wie dies im Datenbyte angegeben wird. „1“-Bits der Datenbytes haben gar keine Wirkung.

Funktionswert

ist vom Typ LONG und zeigt das Ergebnis der Schreibfunktion folgendermaßen an:

0 = ok. Es wurde wie beabsichtigt in das Flash geschrieben.

-1 = Warnung: Quellvariable hatte nicht genügend Bytes.

-2 = Warnung: Ende des Flash-Speichers erreicht.

-3 = Warnung: keine Quellbytes.

-4 = Warnung: Adresse außerhalb Flashbereich.

-16 = Warnung: Parameterfehler.

-17 = Warnung: POKEM_FLASH wird in diesem Modul nicht unterstützt.

1...nnnn = in das Byte vor der angegebenen Adresse hätte mindestens ein Bit von „0“ zu „1“ geschrieben werden müssen, was im Flash nicht möglich ist.

3

POKEM_FLASH wird ab Modulversion 1.06n unterstützt.

Die Modulversion ist per Software Laufzeit abfragbar mit der Funktion SYSVARn, über das Kommando Tiger-Status aus dem Menü Anzeige sowie durch einen Aufkleber auf dem Modul gekennzeichnet.

Wenn es FLASH-Sektoren gibt, die nicht vom Programm-Code belegt werden, stehen diese einem Tiger-BASIC® Programm als Daten-Speicher frei zur Verfügung. Der benutzbare Flash-Speicher beginnt für den Anwender immer bei Adresse 0. Es stehen immer komplette Sektoren zur Verfügung, die byteweise beschrieben und gelesen werden können. Das Löschen erfolgt stets Sektorweise.

Weitere Informationen über den Flashspeicher finden Sie im Hardware-Handbuch.

Die Funktion POKEM_FLASH ersetzt die ältere Instruktion POKE_FLASH und ist in der Ausführung schneller.

Das Beispielprogramm zeigt die Wirkungsweise der verschiedenen Schreibmodi. Wenn das Programm nach einem Durchlauf neu gestartet wird, dann ist das Flash nicht gelöscht, und der Schreibversuch im Modus 0 führt zu einer Fehlermeldung.

POKEM_FLASH

Programmbeispiel:

3

```
'-----
' Name: POKEM_FLASH.TIG
' schreibt mehrere Bytes ins Flash
' beobachte Fehler bei Neustart des Programms (Flash nicht geloescht)
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG R                                ' POKEM_FLASH-Ergebnis
  STRING A$                             ' enthaelt Bytes zum Schreiben
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  CALL SHOW_FLASH_AS_HEX (0, 3)         ' zeige Flash vorm Schreiben
  PRINT #1, " before write"

  A$ = "FF F0 E0%"                     ' Bytes zu Schreiben
  '      adr,src,offs,len,mode
  R = POKEM_FLASH (0, A$, 0, 3, 0)      ' schreiben 3 Bytes im Modus 0
  CALL SHOW_FLASH_AS_HEX (0, 3)         ' zeige Flash danach
  PRINT #1, " ->result: ";R             ' Schreiberfolg

  A$ = "FF F0 C0%"                     ' letztes Byte: 1 Bit mehr =0
  '      adr,src,offs,len,mode
  R = POKEM_FLASH (0, A$, 0, 3, 1)      ' schreiben 3 Bytes im Modus 1
  CALL SHOW_FLASH_AS_HEX (0, 3)         ' zeige Flash danach
  PRINT #1, " ->result: ";R             ' Schreiberfolg

  A$ = "FF FF C0%"                     ' Bits muessten wieder 1 werden
  '      adr,src,offs,len,mode
  R = POKEM_FLASH (0, A$, 0, 3, 2)      ' schreiben 3 Bytes im Modus 2
  CALL SHOW_FLASH_AS_HEX (0, 3)         ' zeige Flash danach
  PRINT #1, " ->result: ";R             ' Schreiberfolg
END                                     ' Ende Task MAIN

'-----
' zeigt Bytes aus Flash als HEX-Bytes
'-----
SUB SHOW_FLASH_AS_HEX ( LONG ADDR, LENGTH )
  BYTE B
  LONG A
  A = ADDR
  USING "UH<2><2> 0.0.0.0.2"           ' Format HEX-Ausgabe
  FOR I = 0 TO LENGTH - 1              ' soviel Bytes zeigen
    PEEK_FLASH A+I, B, 1                ' ein Byte aus Flash auslesen
    PRINT_USING #1, B;                  ' als HEX anzeigen
  NEXT
END
```

Siehe auch: POKE_FLASH

PRIME

PRIME (*Zahl, Tabelle, Länge*)

PRIME: Testet, ob Integerwert Primzahl ist.
(nur in Projektmodell 'PM_FULL')

Funktion: prüft, ob die **Zahl** eine Primzahl ist..

Zahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die zu überprüfende Zahl.
Tabelle	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist eine Adresse im Flash, wo eine binär aufgebaute Tabelle beginnt, die den Überprüfungsvorgang stark beschleunigt.
Länge	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der Tabelle im Flash an. Ist die Länge der Tabelle mit 0 angegeben, wird keine Tabelle verwendet.
Funktionswert	ist vom Typ LONG und ist 0: wenn die Zahl keine Primzahl ist 1: wenn die Zahl eine Primzahl ist.

3

PRIME überprüft Zahlen im positiven Wertebereich von LONG, ob die Zahl eine Primzahl ist. Um dies festzustellen, muß die zu prüfende Zahl durch alle ungeraden Zahlen von 3 bis zu der größten noch möglicherweise teilbaren Zahl geteilt werden. Die größte Zahl, die als Teiler in Frage kommt, liegt bei der Wurzel der zu überprüfenden Zahl. Entsteht bei der Teilung ein Rest, war eine Teilung nicht möglich.

Nun läßt sich der Prüfvorgang weiter beschleunigen, wenn man nur die **Primzahlen** in dem in Frage kommenden Zahlenbereich überprüft. Dazu kann an PRIME eine binär codierte Tabelle übergeben werden. Jedem Bit der Tabelle entspricht eine ungerade Zahl, beginnend bei 1. Ist die Zahl eine Primzahl, dann ist das Bit eine '1', sonst '0'. Es müssen bei der Überprüfung nur Zahlen geteilt werden, deren Bit-Entsprechung in der Tabelle auf '1' sitzt. Ein 4 kByte große Tabelle ist im Verzeichnis Flash mitgeliefert, entsprechend den ungeraden Zahlen von 1 bis 65535.

Die Verwendung der Tabelle bringt einen Geschwindigkeitsvorteil für große Primzahlen ab ca. 100.000.

Primzahlen spielen eine Rolle bei Verschlüsselungsverfahren (z.B. RSA-Verschlüsselung). Ebenso können sie eingesetzt werden, um eine erste Authentizitätskontrolle durchzuführen. Werden zum Beispiel Codenummern oder TAN-Nummern vergeben, die nur durch eine Primzahl (z.B. nur durch 17) teilbar sind, dann sind falsche Nummern bereits im Vorfeld leicht zu identifizieren. Einen Klassi-

PRIME

fizierung von Nummern in Gruppen ist möglich, wenn nur Nummern vergeben werden, die nur durch bestimmte Primzahlen teilbar sind:

- 965413: teilbar durch 17, aber nicht durch 23
- 965402: teilbar durch 23, aber nicht durch 17

3

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139
140	141	142	143	144	145	146	147	148	149
150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179
180	181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198	199

Prime numbers from 0 ... 199

Programmbeispiel:

```

'-----
' Name: PRIME.TIG
' Findet Primzahlen von 2...99 Millionen
'-----
DATALABEL PRIM_BMAP

#DEFINE ISPRIME 1

TASK MAIN                                ' Beginn Task MAIN
PRIM_BMAP::
DATA FILE "PRIMTBL1.BIN"                ' binaere Hilfstabelle
                                           ' ohne diese Tabelle funktioniert
                                           ' die Funktion PRIME viel langsamer

    LONG N, P

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8

P = 0
PRINT #1,"<1>Prime 1...99 Mio";
FOR N = 1 TO 99999999 STEP 2             ' nur ungerade Zahlen
    IF PRIME ( N, PRIM_BMAP, 4K ) = ISPRIME THEN
        P = P + 1
        PRINT #1, "<1BH>A<0><1><0F0H> Prime:";N
        PRINT #1, "<1BH>A<0><2><0F0H> Primes:";P
    ENDIF
NEXT
PRINT #1, ""
PRINT #1, "<1BH>A<0><3><0F0H>***** Ende *****";
END

```

3

PRINT

PRINT

PRINT #Dev_Nr. [, #Sek_Adr][, #Func_Nr], *Ausgabeliste*

PRINT: Formatierte Daten-Ausgabe an Dev.-Treiber

SPC: Erzeugt feste Anzahl Leerzeichen in der PRINT-Ausgabe

TAB: Erzeugt Leerzeichen in der PRINT-Ausgabe bis zu Tabulatorposition

3

Instruktion: PRINT benötigt einen Device-Treiber zur Ausgabe (LCD, serielle Treiber, paralleler Ausgang). Schreibt die in der Ausgabeliste aufgeführten Konstanten, Variablen oder Ausdrücke in das Ausgabegerät **#Dev_Nr**. Steuerzeichen sind KOMMA und SEMIKOLON.

- #Dev_Nr** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-Gerätes an.
- #Sek_Adr** Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.
- #Func_Nr** Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.
- Ausgabeliste** ist eine Liste, die Variable, Konstanten oder Ausdrücke vom Typ BYTE, WORD, LONG oder STRING enthält, auch gemischt. Zum Schluß wird der Cursor auf eine neue Zeile gesetzt, wenn dies nicht durch Angabe eines SEMIKOLON unterdrückt wird.

Die Trennzeichen zwischen den in der Liste aufgeführten Elemente haben Auswirkungen auf die Formatierung der Ausgabe. Außerdem sind in der Liste Funktionen als Ausgabe-Elemente erlaubt, die die Formatierungsmöglichkeiten erweitern:

Trennzeichen oder Funktion	Auswirkung
; (SEMIKOLON)	Elemente werden aneinandergereiht ohne Zwischenraum. Am Ende der Liste: 'Neue Zeile' wird unterdrückt.
, (KOMMA)	nächstes Element wird am nächsten Standardtabulator ausgegeben.
TAB(n)	nächstes Element wird an der Position n ausgegeben.
SPC(n)	rückt um n Schritte vor und gibt dann nächstes Element aus. Es werden keine Leerzeichen gedruckt.

Beachte: Die Ausgabelänge einer PRINT-Instruktion ist auf 240 Zeichen begrenzt.

Weitere Formatierungsmöglichkeiten bietet die Instruktion PRINT_USING in Zusammenhang mit USING (Seite 498), FRAME (Seite 206), TABS (Seite 209).

Multitasking: Wenn der Ausgabeprozess vorübergehend „wartet“, laufen alle anderen Tasks ungestört weiter. Die CPU-Zeit der wartenden Task wird zugunsten anderer Tasks freigegeben.

Device-Treiber

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc., können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B. Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“

PRINT

enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

Programmbeispiel:

```
' -----  
' Name: PRINT.TIG  
' -----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, "Hello World"                ' Auf LC-Display ausgeben  
END                                       ' Ende Task MAIN
```

PRINT_USING

PRINT_USING #Dev_Nr. [, #Sek_Adr][, Func_Nr], Ausgabeliste

PRINT USING: Formatierte Daten-Ausgabe an Dev.-Treiber mit zusätzlicher Ausgabeformatierung.

Instruktion: PRINT_USING benötigt einen Device-Treiber zur Ausgabe (LCD, serielle Treiber, paralleler Ausgang). Schreibt die in **Variablenliste** aufgeführten Datenelemente in das Ausgabegerät **Dev_Nr**. Die zuvor mit den Instruktionen USING, FRAME und TABS gesetzten Ausgabeformate kommen zur Auswirkung.

- #Dev_Nr** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-Gerätes an.
- #Sek_Adr** Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.
- #Func_Nr** Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.
- Ausgabeliste** ist eine Liste, die Variable, Konstanten oder Ausdrücke vom Typ BYTE, WORD, LONG oder STRING enthält, auch gemischt. Zum Schluß wird der Cursor auf eine neue Zeile gesetzt, wenn dies nicht durch Angabe eines SEMIKOLON unterdrückt wird.

Zuvor ausgeführte FRAME, TABS und USING-Instruktionen werden bei der Formatierung der Ausgabe berücksichtigt.

Die Trennzeichen zwischen den in der Liste aufgeführten Elemente haben Auswirkungen auf die Formatierung der Ausgabe. Außerdem sind in der Liste Funktionen als Ausgabe-Elemente erlaubt, die die Formatierungsmöglichkeiten erweitern:

PRINT_USING

3

Trennzeichen oder Funktion	Auswirkung
; (SEMIKOLON)	Elemente werden aneinandergereiht ohne Zwischenraum. Am Ende der Liste: 'Neue Zeile' wird unterdrückt.
, (KOMMA)	nächstes Element wird am nächsten Standardtabulator ausgegeben. (in Schritten von 8)
TAB(n)	nächstes Element wird an der Position n ausgegeben. TAB geht auch zurück.
SPC(n)	rückt um n Schritte vor. Es werden keine Leerzeichen gedruckt. SPC geht nur vorwärts.

Weitere Formatierungsmöglichkeiten bieten die Instruktionen

- USING, Seite 498
- FRAME, Seite 206
- TABS, Seite 209

Diese Instruktionen müssen vor der Benutzung von PRINT_USING ausgeführt worden sein.

Beachte: Die Ausgabelänge einer PRINT_LINE-Instruktion ist auf 240 Zeichen begrenzt.

Beim Multitasking werden die Aufgaben auf verschiedene Prozesse verteilt. Wenn der Ausgabeprozess vorübergehend „wartet“, laufen alle anderen Prozesse ungestört weiter. Desweiteren wird die CPU-Zeit des wartenden Prozesses freigesetzt und kann so von anderen Prozessen genutzt werden.

Device-Treiber

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc, können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die

man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

Programmbeispiel:

```

' -----
' Name: PRINT_USING.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
    REAL A,B,C                            ' REAL-Variablen deklarieren
    ' LCD-Treiber installieren (BASIC-Tiger)
    INSTALL DEVICE #1, "LCD1.TDD"
    ' LCD-Treiber installieren (TINY-Tiger)
    '   INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
    USING "NF<2><1> <5>V0.0.0.0.&
0.0.0.2.5.0.0.0.0.0.0"                  ' Ausgabe-Format festlegen
    A = 2.45678                          ' Variablen besetzen
    B = -34.5656                          '
    C = A + B                            ' Summe zuweisen
    PRINT_USING #1, " A ="; A             ' Var. formatiert ausgeben
    PRINT_USING #1, " B ="; B             '
    PRINT_USING #1, " C ="; C             '
END                                        ' Ende Task MAIN

```

PUT

PUT

PUT #Dev_Nr. [, #Sek_Adr][, #Func_Nr], *Ausgabeliste*

PUT: Unformatierte Daten-Ausgabe an Device-Treiber

Instruktion: PUT benötigt immer einen Device-Treiber. Schreibt die in **Ausgabeliste** aufgeführten Konstanten, Variablen oder Ausdrücke in den Ausgabepuffer des Gerätes **Dev_Nr.**

3

#Dev_Nr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Nummer des I/O-Gerätes an.
#Sek_Adr	Sekundär-Adresse, legt die Kanal-Nummer innerhalb dieses Device-Treibers fest mit dem kommuniziert werden soll. Ohne diese optionale Angabe wird die Sekundär-Adresse = 0 genommen. Sek-Adr ist Variable, Konstante oder Ausdruck vom Typ BYTE, WORD oder LONG.
#Func_Nr	Function-Code. Dieser optionale Parameter legt spezielle Funktions-Anforderungen für den jeweils verwendeten Device-Treiber fest. Näheres siehe unter der Beschreibung des jeweiligen Device-Treibers.
Ausgabenliste	ist eine Liste, die Variable, Konstanten oder Ausdrücke vom Typ BYTE, WORD, LONG oder STRING enthält, auch gemischt. Trennzeichen in der Ausgabeliste ist das Komma.

Es findet weder eine Formatierung noch eine ASCII-Umwandlung der numerischen Variablen statt. Die niederwertigen Bytes der numerischen Variablen werden zuerst übertragen.

Beachte: Die Ausgabelänge einer PUT-Instruktion ist auf 240 Zeichen begrenzt.

Die Aufgabe des Device-Treibers ist es, gerätespezifische Funktionen zu verwalten und ggf. selbständig gerätespezifische Operationen auszuführen. Dies vereinfacht die Programmierung von Ein- und Ausgaben erheblich. Alle Arten von Geräten und seien sie noch so unterschiedlich, werden mit den gleichen Ein-/Ausgabe-Instruktionen: PUT, GET, PRINT, INPUT, PRINT USING und INPUT LINE von einem BASIC-Programm aus angesprochen. Jedem Ein-/Ausgabe-Gerät wird im Programm eine eindeutige Device-Nr zugewiesen unter der es angesprochen wird.

Geräte, die selbst wiederum viele Kanäle besitzen, z.B. A/D-Wandler mit mehreren Kanälen, Mehrfach-Schnittstellen, Maschinen mit mehreren steuerbaren Achsen etc, können zudem von Sekundär-Adressen Gebrauch machen mit der der entsprechende Kanal eines Devices ausgewählt wird. Je nach Device-Treiber sind auch Kanäle möglich, die nur logische Funktion besitzen und keine physikalische Entsprechung haben.

Und schließlich kann man Daten-Ströme und Steuerinformationen mit dem Device-Treiber austauschen. Als Daten-Ströme werden die Ein- und Ausgaben aufgefaßt, die man im normalen Betrieb zum Gerät senden oder von ihm erhalten möchte. Steuerinformationen dienen zur Einstellung bestimmter Betriebs-Parameter (z.B. Baudrate, Spannungs-Bereich, Samplerate, Betriebsart etc.), bzw. zur Abfrage bestimmter Geräte-Zustände (Status-Informationen wie: Puffer-Füllung, Ready/Busy, Betriebsstunden, Fehler-Zustände, Device-Treiber Version, ... etc.). Für den Datenverkehr wird kein Function-Code angegeben (bzw. Function-Code = 0), für Steuerinformationen gibt es eine Reihe von Function-Codes, die jeweils vom Device-Treiber abhängen. Typische Funktionen, die immer wieder auftreten (wie z.B: Puffer-Füllung abfragen), sind durch einheitliche 'Function-Codes' geregelt. Die Include-Datei „UFUNC.INC“ enthält die jeweils verfügbaren Definitionen von Function-Codes. Näheres siehe Device-Treiber-Handbuch unter 'Device-Treiber-Funktionen'.

PUT schreibt eine **Anzahl** Bytes auf ein byteorientiertes Gerät. **Dev_Nr** spezifiziert ein bestimmtes Gerät, das zu Beginn des Programms mit `INSTALL_DEVICE` unter dieser Device-Nummer definiert wurde. Als Ergebnis dieser Ausgabe-Instruktion wird die **Variable** in den entsprechenden Ausgabepuffer geschrieben oder direkt ausgegeben. PUT gibt immer unformatiert aus.

Programmbeispiel:

```

' -----
' Name: PUT.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  L=41424344h                            ' LONG L vorbesetzen
  PRINT #1, "PRINT: "; L                  ' L --> LONG-Wert
  PUT #1, "PUT: ", L                      ' L --> Chars, Low-Byte first
END                                         ' Ende Task MAIN

```

PUT_FIFO

PUT_FIFO

PUT_FIFO Name, Element[, Fehlerbehandlung]

PUT_FIFO: Schreibe Eintrag in Fifo (wenn Platz)

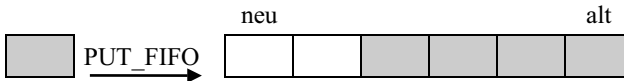
Instruktion: Schreibt ein **Element** in den FIFO-Puffer **Name**. (FIFO=First-In-First-Out) Wenn der FIFO-Puffer voll ist, wird das Element nicht geschrieben und es tritt ein Fehler auf, der mit **Fehlerbehandlung** bearbeitet werden kann:

Element ist vom Typ BYTE, WORD oder LONG.

ON_ERROR_CALL *Unterprogramm*

ON_ERROR_GOTO *Sprungmarke*

PUT_FIFO schreibt nur dann in den Puffer, wenn noch Platz für das Element ist. Ist der Puffer jedoch voll, so tritt ein für diese Instruktion expliziter Fehler auf, der mit der anzugebenden Fehlerbehandlung behandelt wird. Puffer werden dann mit PUT_FIFO beschrieben, wenn alle anfallenden Elemente benötigt werden und kein Element verloren gehen soll. Über den FIFO-Puffer findet auf diese Weise ein Geschwindigkeitsausgleich statt.



Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```

'-----
' Name: PUT_FIFO.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER(512) OF WORD                ' FIFO-Buffer definieren
  WORD I,J                                ' WORD-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 511                        ' 512 Werte in den
    PUT_FIFO BUFFER, 2*I                  ' FIFO-Buffer schreiben
  NEXT I                                  '
  FOR I = 0 TO 18                          ' Die ersten 19 Werte
    GET_FIFO BUFFER, J                    ' aus dem FIFO-Buffer lesen
    PRINT #1,"<1>Buffer";I;" ="; J        ' Wert auf LC-Display ausgeben
    WAIT_DURATION 1000                    ' 1 Sek warten
  NEXT I                                  '
END                                        ' Ende Task MAIN

```

3

Siehe auch: PUTU_FIFO, UNGET_FIFO, GET_FIFO.

PUTU_FIFO

PUTU_FIFO

PUTU_FIFO Name, Element

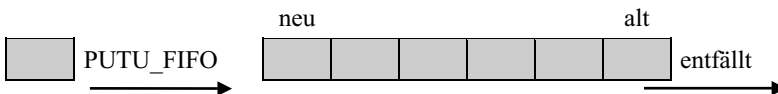
PUTU_FIFO: Schreibe Eintrag in Fifo (ältester Eintrag ev. verloren)

Instruktion: **PUT-U**nconditional, schreibt ein **Element** in den FIFO-Puffer **Name**. Auch wenn der Puffer voll ist, wird geschrieben und das älteste Element geht verloren. (FIFO=First-In-First-Out)

Element ist vom Typ BYTE, WORD oder LONG.

PUTU_FIFO = **PUT U**nconditionally into **FIFO**.

Im Unterschied zu PUT_FIFO schreibt PUTU_FIFO auch dann in den Puffer, wenn dieser bereits voll ist. Das älteste Element geht dabei verloren. Werden zum Beispiel Werte des A/D-Wandlers eingelesen, um nur von Zeit zu Zeit den momentanen A/D-Wert geglättet auszulesen, so gehen sinnvollerweise die weiter zurückliegenden Werte verloren. Die Werte werden mit PUTU_FIFO eingeschrieben und es stehen jederzeit aktuelle Werte im Puffer.



Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

Programmbeispiel:

```

' -----
' Name: PUTU_FIFO.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER(512) OF WORD                ' FIFO-Buffer deklarieren
  WORD I,J                                ' WORD-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 520                          ' 521 Werte in den
    PUTU_FIFO BUFFER, 2*I                  ' FIFO-Buffer schreiben
  NEXT I                                    '
  FOR I = 0 TO 18                          ' Die ersten 19 Werte
    GET_FIFO BUFFER, J                    ' aus dem FIFO-Buffer lesen
    PRINT #1, "<1>Buffer";I;" ="; J        ' Wert auf LC-Display ausgeben
    WAIT DURATION 1000                   ' 1 Sek warten
  NEXT I                                    '
END                                        ' Ende Task MAIN

```

3

Siehe auch: PUT_FIFO, UNGET_FIFO, GET_FIFO.

QUICK_WORD_COS

QUICK_WORD_COS (*String*, *BYBreit*, *PxHoch*, *Flag*)

QUICK_WORD_COS: Integer COSINUS, schnelle Berechnung
(nur in Projektmodell 'PM_FULL')

Funktion: Liefert den Cosinus von **Grad** gewichtet mit **Gewicht** zurück.

3

Grad	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Winkel in hundertstel Grad an.
Gewicht	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Multiplikationsfaktor an, um den Sinuswert im ganzzahligen Bereich darzustellen.
Funktionswert	ist vom Typ LONG und liegt im Bereich von -1 bis 1. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{Bogenmaß} * 180/\text{PI}$

In der Welt der Technik ist ein QUICK_WORD Sinus-Wert in 99% aller Fälle vollkommen ausreichend. Die hohe Geschwindigkeit ist ein grosser Gewinn, wenn viele Operationen erforderlich sind, und/oder kurze Antwortzeiten verlangt werden. Quick-WORD-Sinus ist vorteilhaft, weil die Berechnung sehr schnell ist. Beispiele für Anwendungen sind:

- Grafik: Kreise, Ellipsen, Rotationen zeichnen.
- Regelungen mit Nachführungssysteme, elektrische Maschinen
- Signal-Synthese, Nachbildung vorgegebener Spektren
- Sprach-Synthese

Programmbeispiel:

```

'-----
' Name: QUIK_WORD_COS.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG DEG                              ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR DEG = 0 TO 36000 STEP 200          ' Winkel 0...360 STEP 2 Grad
    PRINT #1, "<1>Quick WORD<13><10> COS ("; DEG/100; " Grad )"
    PRINT #1, "="; QUICK_WORD_COS ( DEG, 10000 )
    WAIT DURATION 1000                  ' 1 Sek warten
  NEXT                                  ' naechster Winkelwert
END                                      ' Ende Task MAIN

```

3

Siehe auch: COS

QUICK_WORD_SIN

QUICK_WORD_SIN

QUICK_WORD_SIN (Grad, Gewicht)

QUICK_WORD_SIN: Integer SINUS, schnelle Berechnung
(nur in Projektmodell 'PM_FULL')

Funktion: Liefert den Sinus von **Grad** gewichtet mit **Gewicht** zurück.

3

Grad	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Winkel in hundertstel Grad an.
Gewicht	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Multiplikationsfaktor an, um den Sinuswert im ganzzahligen Bereich darzustellen.
Funktionswert	ist vom Typ LONG und liegt im Bereich von -1 bis 1. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Bogenmaß in Grad: $\text{Bogenmaß} * 180/\text{PI}$

In der Welt der Technik ist ein QUICK_WORD Sinus-Wert in 99% aller Fälle vollkommen ausreichend. Die hohe Geschwindigkeit ist ein grosser Gewinn, wenn viele Operationen erforderlich sind, und/oder kurze Antwortzeiten verlangt werden. Quick-WORD-Sinus ist vorteilhaft, weil die Berechnung sehr schnell ist. Beispiele für Anwendungen sind:

- Grafik: Kreise, Ellipsen, Rotationen zeichnen.
- Regelungen mit Nachführungssysteme, elektrische Maschinen
- Signal-Synthese, Nachbildung vorgegebener Spektren
- Sprach-Synthese

Programmbeispiel:

```

'-----
' Name: QUIK_WORD_SIN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG DEG                              ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR DEG = 0 TO 36000 STEP 200          ' Winkel 0...360 STEP 2 Grad
    PRINT #1, "<1>Quick WORD<13><10> SIN ("; DEG/100; " Grad )"
    PRINT #1, "="; QUICK_WORD_SIN ( DEG, 10000 )
    WAIT DURATION 1000                  ' 1 Sek warten
  NEXT                                  ' naechster Winkelwert
END                                    ' Ende Task MAIN

```

3

Siehe auch: SIN

RANDOMIZE

RANDOMIZE

RANDOMIZE

RANDOMIZE: Setzt neuen Anfangswert für die Erzeugung der Pseudo-Zufallswerte

Instruktion: Setzt den Anfangswert der Funktion RND unter Verwendung interner sich ständig verändernder Systemparameter.

Durch diese Instruktion ist die durch RND erzeugte Zufallsfolge bei jedem Aufruf eine andere.

Programmbeispiel:

```
'-----
' Name: RANDOMIZE.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG Value                             ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
RANDOMIZE                                ' Startwert ermitteln
LOOP 20                                 ' 20 Durchläufe
  Value = 0                             ' Wert vorbesetzen
  WHILE Value < 1 OR Value > 6           ' Solange Wert nicht 1 bis 6
    Value = RND(0) BITAND 7h            ' Zufallswert <= 7 generieren
  ENDWHILE
  PRINT #1, "<1>RND(0) ="; Value         ' Zufallswert ausgeben
  WAIT DURATION 2000                    ' 2 Sek warten
ENDLOOP
END                                     ' Ende Task MAIN
```

READ_FIFO

READ_FIFO (*Fifobu*, *Index*)

READ_FIFO: Beliebiges Element aus Fifo lesen

Funktion: Liest ein beliebiges Element aus dem FIFO-Puffer **Fifobu**, ohne das Element zu entnehmen (im Gegensatz zu GET_FIFO). (FIFO=First-In-First-Out)

Fifobu ist eine Variable vom Typ FIFO.

Index ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Index des zu lesenden Elementes an. Der Index 0 entspricht dem zuletzt in den Puffer geschriebenen Element, der Index 1 dem vorletzten, etc.

Funktionswert ist vom gleichen Typ wie der Datentyp der FIFO-Elemente. Eine Typwandlung bei der Zuweisung von LONG nach REAL findet automatisch statt.

Programmbeispiel:

```

' -----
' Name: READ_FIFO.TIG
' -----
TASK MAIN
  FIFO BUFFER (512) OF WORD
  WORD I,J
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 511
    PUT_FIFO BUFFER, I
  NEXT
  PRINT #1, "Length:"; LEN_FIFO (BUFFER)
  FOR I = 0 TO 9
    J = READ_FIFO (BUFFER, I)
    PRINT #1, "<2><10>Value:"; J
    WAIT DURATION 1000
  NEXT
  PRINT #1, "Length:"; LEN_FIFO (BUFFER)
END
  ' Beginn Task MAIN
  ' FIFO-Buffer deklarieren
  ' WORD-Variablen deklarieren
  ' 512 Werte in den
  ' FIFO-Buffer schreiben
  ' Groesse FIFO-Buffer ausgeben
  ' Die ersten 10 Werte
  ' aus dem FIFO-Buffer lesen
  ' Wert auf LC-Display ausgeben
  ' 1 Sek warten
  ' Groesse FIFO-Buffer ausgeben
  ' Ende Task MAIN

```

Siehe auch: GET_FIFO, PUT_FIFO, PUTU_FIFO

RELEASE_TASK

RELEASE_TASK

RELEASE_TASK Name

RELEASE_TASK: Rest der augenblicklichen Zeitscheibe frei geben für andere Tasks

Instruktion: Verläßt die task und gibt die CPU-Zeit frei.

Durch diese Instruktion ist eine andere Form dynamischer Priorisierung bzw. der dynamischen Verteilung der CPU-Leistung auf die Prozesse möglich.

Beispiel: Bei einer bestimmten Task fallen nur zeitweise Aufgaben an. Sind diese jedoch vorhanden, dann sollen sie schnell bearbeitet werden. Daher läuft die Task mit hoher Priorität.

Liegt keine Aufgabe an, wird ein RELEASE_TASK-Instruktion ausgeführt, um die CPU-Zeit sofort wieder für andere Tasks freizugeben.

Programmbeispiel:

```
'-----
' Name: RELEASE_TASK.TIG
'-----

TASK MAIN                                ' Beginn Task MAIN
  LONG Count                             ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  RUN TASK T2                            ' Starte Task T2
  FOR Count = 0 TO 9999999                ' Anfang (leere) FOR-Schleife
  NEXT                                    ' Ende (leere) FOR-Schleife
END                                        ' Ende Task MAIN

'-----
' Task T2: Wartet auf Tastendruck
'-----

TASK T2                                ' Beginn Task T2
  BYTE N                                ' BYTE-Variable deklarieren
  PRINT #1, "<l>Wait for key"           ' Ausgabe auf LC-Display
  FOR N = 0 TO 0 STEP 0                  ' Endlosschleife bis N=1(GET!)
    RELEASE_TASK                         ' Rest der Task-Zeit freigeben
    GET #1, #0, #1, 1, N                 ' N=Zeichen im Tastatur-Buffer
  NEXT                                    ' Ende Endlosschleife
  PRINT #1, "Key pressed"                ' Ausgabe auf LC-Display
END                                        ' Ende Task T2
```

REMDOUBLE\$

REMDOUBLE\$ (String1, String2)

REMDOUBLE\$: Entferne doppelte Zeichen aus String

Funktion: Entfernt doppelt aufeinanderfolgende Zeichen, die in **String2** enthalten sind, aus **String1**. Ein Zeichen bleibt erhalten.

String1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist der zu bearbeitende String..
String2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Zeichen, die in String1 bearbeitet werden.
Funktionswert	ist vom Typ STRING und enthält String1 mit denjenigen Bytes, die in String2 vorkommen, nicht mehr doppelt oder mehrfach aufeinanderfolgend. Diese Zeichen kommen nur noch einzeln vor.

3

Mit der Funktion REMDOUBLE\$ können auf einfache Weise Zeichen, die bei der weiteren Verarbeitung stören oder nicht mehr benötigt werden, aus einem Datenstrom entfernt werden.

Beispiele:

- String2 = “,“: aus ‘X2.3,,,Y3.4,,Z5.66’ wird ‘X2.3,Y3.4,Z5.66’.
- String2 = “x*#“: aus ‘###777*****345xxx66’ wird ‘#777*345x66’.
- String2 = “0“: aus ‘Hello, 1000\$’ wird ‘Hello, 10\$’.
- String2 = “+“: aus ‘+++++++’ wird ‘+’.

Programmbeispiel:

```

' -----
' Name: REMDOUBLE$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
PRINT #1, REMDOUBLE$("The quick & ' entfernt alle doppelten
                brown fox", " ") ' Leerzeichen aus dem Text
END                                          ' Ende Task MAIN

```

Siehe auch: REMOVES\$, TRIM\$, LEFT\$, RIGHT\$

REMOVE\$

REMOVE\$

REMOVE\$ (*String1*, *String2*)

REMOVE\$: Entferne Zeichen aus String

Funktion: Entfernt Zeichen, die in **String2** enthalten sind, aus **String1**.

String1 ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist der zu bearbeitende String.

String2 ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Zeichen, die in String1 bearbeitet werden.

Funktionswert ist vom Typ STRING und enthält String1 ohne Bytes, die in String2 vorkommen.

Mit der Funktion REMOVE\$ können auf einfache Weise Zeichen, die bei der weiteren Verarbeitung stören oder nicht mehr benötigt werden, aus einem Datenstrom entfernt werden.

Beispiele:

- String2 = “.,“: aus ‘X2.3,,,Y3.4,,Z5.66’ wird ‘X23Y34Z566’.
- String2 = “x*#“: aus ‘###777*****345xxx66’ wird ‘77734566’.
- String2 = “0“: aus ‘Hello, 1000\$’ wird ‘Hello, 1\$’.
- String2 = “+“: aus ‘+++++++’ wird ‘’.

Programmbeispiel:

```
'-----
' Name: REMOVE$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, REMOVE$("Th-e qui--ck -&    ' entfernt alle "-" aus
bro-w-n --fo-x", "-")                  ' dem Ausgangstext
END                                     ' Ende Task MAIN
```

Siehe auch: REMDOUBLES\$, TRIM\$, LEFT\$, RIGHT\$

RES_BIT

RES_BIT Variable, Bitnr

RES_BIT: Setzt ein Bit in numerischer Variable auf ,0'

Instruktion: Setzt das Bit mit der Nummer **Bitnr** in der **Variablen** auf 0.

Variable ist vom Datentypen BYTE, WORD oder LONG.

Bitnr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welches Bit invertiert werden soll. Ist Bitnr größer als die Anzahl Bits der Variablen, dann verändert sich die Variable nicht.

Programmbeispiel:

```
'-----
' Name: RES_BIT.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE B                                ' BYTE-Variable deklarieren
  LONG L                                ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2> k 0.0.0.0.2UH<8><8> k 0.0.0.0.8"
  B = 0FFh                                ' B vorbesetzen
  L = 7FFFFFFFh                            ' L vorbesetzen
  RES_BIT B,4                             ' Bit 4 loeschen
  RES_BIT L,14                             ' Bit 14 loeschen
  PRINT USING #1, "B = "; B, "L = "; L     ' B und L ausgeben
END                                         ' Ende Task MAIN
```

RESTART_PROG

RESTART_PROG

RESTART_PROG ()

RESTART_PROG: Startet BASIC-Programm neu, Warmstart
(nur in Projektmodell 'PM_FULL')

Instruktion: Startet das Programm per Software neu. Alle Tasks werden angehalten und wie nach einem Reset eine Neu-Initialisierung durchgeführt. Die Task Main wird neu gestartet und dort werden, wenn vorhanden, die Device-Treiber neu initialisiert.

Beachte: Wenn andere Chips im Gesamtsystem ihren Reset von einem dafür vorgesehenem IC erhalten, so werden diese Chips nicht mit einem Hardware-Reset be-
dacht. Das gilt zum Beispiel auch für bestimmte Flash-Typen (ab 2Mbit) und die
CAN-Hardware im TCAN-Modul. Als externe Geräte kommen am häufigsten LCDs
vor.

Programmbeispiel:

```
'-----  
' Name: RESTART_PROG.TIG  
' startet das Programm per Software neu (wie Reset)  
' ab Modulversion 1.06n  
'-----  
TASK Main  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  
  PRINT #1, "<1>";  
  FOR X = 0 TO 9  
    PRINT #1, CHR$(48+X);  
    WAIT_DURATION 250  
  NEXT  
  PRINT #1, "<13><10>program will be"  
  PRINT #1, "restarted in 1 sec"  
  WAIT_DURATION 1000  
  RESTART_PROG          ' startet das Programm neu  
  PRINT #1, "<1>FAILED<13><10>program NOT restarted"  
END
```


RETURN

RETURN

RETURN: Unterprogramm verlassen

Instruktion: Kehrt zurück aus Unterprogramm.

RETURN darf an jeder Stelle des Unterprogramms stehen und mehrfach vorkommen.

Anmerkung: Die END-Anweisung der Unterprogramm-Definition erzeugt ebenfalls ein RETURN, wenn dieses vergessen wurde.

Programmbeispiel:

```
' -----
' Name: RETURN.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, "<l>Connect pin L87"          ' Ausgabe auf LC-Display
  PRINT #1, "with GND"                  ' Ausgabe auf LC-Display
  CALL BLINK                            ' Aufruf der Subroutine BLINK
  PRINT #1, "=== END ==="               ' Ausgabe auf LC-Display
END                                     ' Ende Task MAIN

' -----
' Subroutine laesst Port 8, Pin 3 blinken bis Port 8, Pin 7 low ist
' -----
SUB BLINK                                ' Beginn Subroutine BLINK
  BYTE B                                ' BYTE-Variable deklarieren
  DIR_PIN 8,3,0                          ' Port 8, Pin 3 ist Ausgang
  DIR_PIN 8,7,1                          ' Port 8, Pin 7 ist Eingang
  LL_IPOINT_OUT 8,8                      ' Port 8, Pin 3 = High
  LOOP 9999999                          ' sehr viele Loops
    LL_IPOINT_XOR 8,8                    ' Toggle Port 8, Pin 3
    LL_IPOINT_IN 8,B,128                 ' B <-- Port 8, Pin 7
    WAIT DURATION 200                    ' 200 ms warten
    IF B = 0 THEN                        ' Falls Port 8, Pin 7 = Low,
      RETURN                            ' Subroutine BLINK verlassen
    ENDIF
  ENDDLOOP                              ' naechster Loop
END                                     ' Ende Subroutine BLINK
```

RFROMS

RFROMS

RFROMS (*String, Position, Anzahl*)

RFROMS: Schnelle Ausführung: Real Wert aus String.

Funktion: **REAL-from-String** liest **Anzahl** Bytes aus **String** an der angegebenen **Position** und setzt daraus eine REAL-Zahl zusammen.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich entweder von -8 bis -1 oder von 1 bis 8 und gibt die Anzahl der zu entnehmenden Bytes an. Sinnvolle Anzahl ist 8 oder -8.
Funktionswert	ist vom Typ REAL und enthält die aus dem String gelesene Zahl.

RFROMS entnimmt einem String eine angegebene Anzahl von Bytes und bildet daraus eine REAL-Zahl. Wird die Anzahl positiv angegeben, liest RFROMS 'low-Byte-first', bei negativer Anzahl 'high-Byte-first'. Ist die Anzahl zu groß, werden die übrigen Bytes ignoriert. Aufgrund des speziellen Formates der REAL-Zahlen ist eine andere Anzahl als 8 oder -8 nicht sinnvoll.

Die Funktionen NTO\$\$, NFROMS, RTO\$\$, RFROMS, STOS\$ erhöhen die Verarbeitungsgeschwindigkeit und sparen Speicherplatz, wenn Zahlenwerte zwischengespeichert oder übertragen werden sollen. Der Ablage-String entspricht in etwa einer Struktur in C.

Programmbeispiel:

```

-----
' Name: RFROMS.TIG
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  REAL R,S                               ' REAL-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  R = 2.22E17                             ' R vorbesetzen
  A$ = "ABC.....DE"                       ' String enthaelt 13 Bytes
  A$ = RTOS$ (A$, 3, 8, R)                 ' REAL-Zahl R in A$ ablegen
  S = RFROMS (A$, 3, 8)                    ' ab Pos.3, 8 Bytes, low first
  PRINT #1, "R ="; R; "<13><10>S ="; S      ' REAL-Zahlen R und S ausgeben
END                                         ' Ende Task MAIN

```

3

Siehe auch inverse-Funktion: RTOS\$

RIGHT\$

RIGHT\$

RIGHT\$ (*String*, *Länge*)

RIGHT\$: Rechter Teilstring

Funktion: Liefert den rechten Teil der angegebenen **Länge** aus der Zeichenfolge **String** zurück.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Länge	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Zeichen an, die in dem Zielstring enthalten sind.
Funktionswert	ist vom Typ STRING und enthält den rechten Teil vom Quell-String mit der angegebenen Länge. Die Länge ist jedoch nie größer als die ursprüngliche Länge des Strings.

Programmbeispiel:

```
'-----
' Name: RIGHT$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "Hello, I am the BASIC Tiger"      ' String vorbesetzen
  PRINT #1, RIGHT$ (A$,5)                 ' rechten Teilstring ausgeben
END                                         ' Ende Task MAIN
```

Siehe auch: LEFT\$, MID\$, LEN

RND

RND (Wert)

RND: Liefert Pseudo-Zufallswert

Funktion: Liefert eine Pseudo-Zufallszahl im Bereich zwischen 0 und 65535 zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und 65535 und gibt den Startwert der Zufallserzeugung vor.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```

' -----
' Name: RND.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG Value                             ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  RANDOMIZE                              ' Startwert ermitteln
  LOOP 20                                ' 20 Durchläufe
    Value = 0                             ' Wert vorbesetzen
    WHILE Value < 1 OR Value > 6           ' Solange Wert nicht 1 bis 6
      Value = RND(0) BITAND 7h             ' Zufallswert <= 7 generieren
    ENDWHILE
    PRINT #1, "<1>RND(0) ="; Value          ' Zufallswert ausgeben
    WAIT DURATION 2000                     ' 2 Sek warten
  ENDLOOP
END                                        ' Ende Task MAIN

```

Siehe auch: RANDOMIZE

RTL

RTL

RTL (Wert)

RTL: Typumwandlung: REAL to LONG

Funktion: **REAL-To-LONG** wandelt einen REAL-**Wert** in einen LONG-Wert um.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL.

Funktionswert ist vom Typ LONG.

Programmbeispiel:

```
' -----  
' Name: RTL.TIG  
' -----  
TASK MAIN                                ' Beginn Task MAIN  
  LONG L                                ' LONG-Variable deklarieren  
  REAL R                                ' REAL-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8  
  R = -234.567                            ' R besetzen  
  L = RTL (R)                            ' Umwandeln in LONG  
  PRINT #1, "<1>REAL ="; R                ' Ausgabe auf LC-Display  
  PRINT #1, "LONG ="; L                  ' Ausgabe auf LC-Display  
END                                        ' Ende Task MAIN
```

Siehe auch: LTR

RTOS\$

RTOS\$ (*String*, *Position*, *Anzahl*, *Wert*)

RTOS\$: Schnelle Ausführung: Real Wert nach String.

Funktion: **REAL-To-String** schreibt **Anzahl** Bytes des **Wertes** in den **String** an die angegebene **Position**. Die Bytes werden ohne Wandlung geschrieben.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.
Position	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition im String an. Die Position beginnt mit 0 und endet mit der Stringlänge minus 8.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der abzulegenden Bytes an. Bei REAL-Zahlen ist als einziger Wert 8 sinnvoll.
Wert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL und ist das Element, aus dem Bytes in den String eingefügt werden.
Funktionswert	ist vom Typ STRING und enthält den String mit der eingefügten Zahl.

3

RTOS\$ legt in einem String eine angegebene Anzahl von Bytes aus der Zahl ab. Wird die Anzahl positiv angegeben, legt RTOS\$ die Bytes 'low-Byte-first', bei negativer Anzahl 'high-Byte-first' ab. Ist die Anzahl zu groß, werden die übrigen Bytes als 0-Bytes geschrieben.

Der String wird verlängert, wenn die angegebene Position außerhalb des Strings liegt. Die maximale Länge wird jedoch nie überschritten. Es kann dadurch auch eine "Lücke" im String entstehen, die undefinierte Bytes enthält.

Die Funktionen NTOSS\$, NFROMS, RTOS\$, RFROMS, STOSS erhöhen die Verarbeitungsgeschwindigkeit und sparen Speicherplatz, wenn Zahlenwerte zwischengespeichert oder übertragen werden sollen.

Der Ablage-String entspricht in etwa einer Struktur in C. Es können verschiedene Datentypen in einer Variablen abgelegt werden.

RTOS\$

Programmbeispiel:

```
'-----
' Name: RTOS$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  REAL R,S                               ' REAL-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  R = 2.22E17                             ' R vorbesetzen
  A$ = "ABC.....DE"                       ' String enthaelt 13 Bytes
  A$ = RTOS$ (A$, 3, 8, R)                 ' REAL-Zahl R in A$ ablegen
  S = RFROMS (A$, 3, 8)                   ' ab Pos.3, 8 Bytes, low first
  PRINT #1, "R ="; R; "<13><10>S ="; S      ' REAL-Zahlen R und S ausgeben

  A$ = "ABC.....DE"                       ' String enthaelt 13 Bytes
  PRINT #1, "<1>";LEN(A$)                   ' Laenge A$ vorher: 13
  A$ = RTOS$ (A$, 15, 8, R)               ' REAL-Zahl R in A$ ablegen
  S = RFROMS (A$, 15, 8)                 ' ab Pos.3, 8 Bytes, low first
  PRINT #1, "R ="; R; "<13><10>S ="; S      ' REAL-Zahlen R und S ausgeben
  PRINT #1, LEN(A$)                      ' Laenge A$ nachher: 23
END                                       ' Ende Task MAIN
```

Siehe auch inverse-Funktion: RFROMS

RUN_TASK

RUN_TASK *Name* [, *Priorität*]

RUN_TASK: Startet Task

Instruktion: Startet Task **Name** mit der optional angegebenen **Priorität**.

Priorität ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welche die gestartete Task hat. Die niedrigste Priorität ist 0, die höchste 255. Priorität 0 stoppt die Task!

Die zur Verfügung gestellte CPU-Leistung hängt von der gewählten Priorität dieses Prozesses und der momentanen Priorität aller anderen aktiven Prozesse ab.

Näheres zur Priorität erfahren Sie unter der Instruktion SET_TASK_Prio.

Programmbeispiel:

```

'-----
' Name: RUN_TASK1.TIG
'-----
' Task MAIN und Task GIVE_BEEPS benutzen fuer ihre Ausgaben dasselbe
' Unterprogramm. Eine Task gibt Text aus, die andere Signaltoene.
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
RUN_TASK GIVE_BEEPS                      ' TASK GIVE_BEEPS starten
CALL LCD ("hello world ",200)           ' Aufruf der Ausgabe-Routine
END                                     ' Ende Task MAIN

'-----
' TASK GIVE_BEEPS gibt einen Signal-Ton aus
'-----
TASK GIVE_BEEPS                          ' Beginn Task GIVE_BEEPS
CALL LCD ("<7>",1000)                   ' Aufruf der Ausgabe-Routine
END                                     ' Ende Task GIVE_BEEPS

'-----
' Unterprogramm LCD gibt A$ mit einer Pause von L ms aus
'-----
SUB LCD (STRING A$; LONG L)              ' Beginn Unterprogramm LCD
FOR N=0 TO 1 STEP 0                     ' Endlosschleife
PRINT #1, A$;                          ' Ausgabe auf LC-Display
WAIT_DURATION L                        ' L ms warten
NEXT N                                  ' Ende Endlosschleife
END                                     ' Ende Unterprogramm LCD

```

RUN_TASK

Programmbeispiel:

3

```
'-----
' Name: RUN_TASK2.TIG
'-----
' Zwei Tasks (T1 und T2) mit identischen Aufbau und identischer
' Funktion werden mit unterschiedlicher Prioritaet gestartet. Dies
' zeigt sich durch unterschiedliche Ausgabegeschwindigkeit (mehr
' "." als "-").
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
RUN_TASK T1,100                          ' TASK T1 starten (Prio=100)
RUN_TASK T2,50                           ' TASK T2 starten (Prio=50)
LOOP 9999999                             ' sehr viele Loops
ENDLOOP                                  '
END                                       ' Ende Task MAIN

'-----
' TASK T1 gibt Punkte "." aus
'-----
TASK T1                                ' Beginn Task T1
BYTE B,K                                ' BYTE-Variable deklarieren
FOR B = 0 TO 1 STEP 0                    ' Endlosschleife
PRINT #1, ".";                           ' Ausgabe auf LC-Display
FOR K = 0 TO 2000                        ' verbrauche Zeit
NEXT                                     '
NEXT                                     ' Ende der Endlosschleife
END                                       ' Ende Task T1

'-----
' TASK T2 gibt Striche "-" aus
'-----
TASK T2                                ' Beginn Task T2
BYTE B,K                                ' BYTE-Variable deklarieren
FOR B = 0 TO 1 STEP 0                    ' Endlosschleife
PRINT #1, "-";                           ' Ausgabe auf LC-Display
FOR K = 0 TO 2000                        ' verbrauche Zeit
NEXT                                     '
NEXT                                     ' Ende der Endlosschleife
END                                       ' Ende Task T2
```

SCAN_TO_CHAR\$

SCAN_TO_CHAR\$ (Code, Toggle)

SCAN_TO_CHAR\$: Scan Taste, die Code generiert in Keyboards
(nur in Projektmodell 'PM_FULL')

Funktion: .

Code	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und übergibt den Scan-Code (Wertebereich 0...255).
Toggle	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und übergibt die derzeit gültigen Toggle-Bits.
Funktionswert	ist vom Typ STRING und erhält das Zeichen angehängt , welches durch Code als Index aus der Codetabelle hervorgeht.

3

Tip Hinweis: diese Funktion ist in ausgezeichneter Weise geeignet, über einen analogen Eingang eine Tastatur zu realisieren. Siehe dazu auch INDEX_1D.

Vor der Anwendung der Funktion SCAN_TO_CHAR\$ muß einmal die Funktion SCAN_TO_CHAR_SETUP ausgeführt worden sein. In der Regel ist ebenfalls vor SCAN_TO_CHAR\$

- ein Scan-Code entstanden (INDEX_1D, INDEX_2D, etc.)
- hat eine Entprellung stattgefunden (DEBOUNCE)
- sind die Toggles verwaltet worden (Shift, etc.) mit SCAN_TO_TOGGLE.

SCAN_TO_CHAR\$ setzt einen Scan-Code in einen Zeichencode um, wie die Funktion CONVERT\$, jedoch mit zusätzlicher Funktionalität, wie sie bei der Bearbeitung von Tastaturen meist benötigt wird. Dazu zählen folgende Funktionen, die mit Hilfe des Parameters TOGGLE gesteuert werden:

- Verschieben des Code-Bereiches, z.B. Großbuchstaben weil Shift gedrückt ist, etc.
- Automatisches Wiederholen einer Taste
- Sondertasten wie Shift, Control, ALT, die die Toggle-Bits setzen bzw. zurücksetzen. Dabei kann ein Toggle-Bit auch verwendet werden, um die Funktion mit einer LED anzuzeigen.
- Sonderfunktionen bei kleinen Tastaturen, die mit einer Hand bedient werden. Z.B. Nach dem Drücken einer Sondertaste wie Shift bleibt die Shift-Funktion aktiv bis die nächste Taste eingegeben worden ist, dann fällt die Funktion zurück (STICK).
- Tasten de-aktivieren

SCAN_TO_CHAR\$

Bedeutung der Toggle-Bits:

Bitposition								Bedeutung
7	6	5	4	3	2	1	0	
•								0: einfacher Tastendruck, 1:Auto-Wiederholung
	•							0: kein Offset für diesen Code, 1: Offset für diesen Code
		•						Toggle 2: 0: inaktiv, 1: aktiv
			•					Toggle 1: 0: inaktiv, 1: aktiv
				•				Toggle 0: 0: inaktiv, 1: aktiv
					0	0	0	normale Taste: Code generieren
					0	0	1	inaktive Taste, keinen Code generieren
					0	1	0	setze Toggle(s), keinen Code generieren
					0	1	1	setze Toggle(s) zurück, keinen Code generieren
					1	0	0	STICK_1 Toggle(s), keinen Code generieren
					1	0	1	invertiere Toggle(s), keinen Code generieren
					1	1	0	reserviert, keinen Code generieren
					1	1	1	reserviert, keinen Code generieren

Beispiele:

Bitposition								Bedeutung
7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	1	Toggle-0 + Flip
0	0	0	1	0	0	1	0	Toggle-1 + Set
0	0	0	1	1	0	1	1	Toggle-0 + Toggle-1 + Reset
0	0	0	0	1	1	0	0	Toggle-0 + Stick

Siehe auch: SCAN_TO_CHAR_SETUP, SCAN_TO_TOGGLE, DEBOUNCE, CONVERT\$

SCAN_TO_CHAR_SETUP

SCAN_TO_CHAR_SETUP (*Attrib*\$, *Conv*\$, *Wh1*, *Wh2*, *Offset*, *Offset2*, *Offset3*)

SCAN_TO_CHAR_SETUP: Tastatur-Scan einrichten
(nur in Projektmodell 'PM_FULL')

Funktion: .

Attrib\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Attribute der Zeichencodes.				
Conv\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist die Tabelle mit den Zeichencodes.				
Wh1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Funktionsaufrufe an, die notwendig sind, um die Automatische Wiederholungsfunktion der Taste in Gang zu setzen (0=1=sofort, 2=nach 2 Aufrufen, etc.).				
Wh2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Funktionsaufrufe an, die notwendig sind, bei bereits aktivierter Automatischer Wiederholungsfunktion eine neue Tasteneingabe zu generieren. Daraus ergibt sich die Wiederholfrequenz.				
Offsetn	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und geben den Offset in der Code-Tabelle in 3 Stufen an.				
Funktionswert	ist vom Typ LONG und enthält das Ergebnis der Funktion folgendermaßen an: <table> <tr> <td>0:</td><td>ok. Funktion wurde erfolgreich ausgeführt.</td></tr> <tr> <td>-1:</td><td>es ist ein Parameterfehler aufgetreten</td></tr> </table>	0:	ok. Funktion wurde erfolgreich ausgeführt.	-1:	es ist ein Parameterfehler aufgetreten
0:	ok. Funktion wurde erfolgreich ausgeführt.				
-1:	es ist ein Parameterfehler aufgetreten				

Die Funktion SCAN_TO_CHAR_SETUP bereitet die Funktion SCAN_TO_CHAR\$ vor. Es werden Attribute für Tasten gesetzt und Zeichencodes, die aus den Scancodes hervorgehen sollen. Der Parameter ,TOGGLE' der SCAN_TO_CHAR\$-Funktion wählt einen von vier Offsets (0, Offset1, Offset2, Offset3) in die Code-Tabelle an. Somit können 4 verschiedene Code-Bereiche vorgegeben werden. Ein Code-Bereich kann zum Beispiel einer gedrückten Shift-Taste entsprechen, oder bei Eingabe durch schnelles hintereinanderdrücken einer Taste ein bestimmter Zustand wie ,2-mal schnell gedrückt bedeutet 2. Code-Bereich.

SCAN_TO_CHAR_SETUP

Bedeutung der Attribut-Bits:

Bitposition								Bedeutung
7	6	5	4	3	2	1	0	
•								0: einfacher Tastendruck, 1:Auto-Wiederholung
	•							0: kein Offset für diesen Code, 1: Offset für diesen Code
		•						Toggle 2: 0: inaktiv, 1: aktiv
			•					Toggle 1: 0: inaktiv, 1: aktiv
				•				Toggle 0: 0: inaktiv, 1: aktiv
					0	0	0	normale Taste: Code generieren
					0	0	1	inaktive Taste, keinen Code generieren
					0	1	0	setze Toggle(s), keinen Code generieren
					0	1	1	setze Toggle(s) zurück, keinen Code generieren
					1	0	0	STICK_1 Toggle(s), keinen Code generieren
					1	0	1	invertiere Toggle(s), keinen Code generieren
					1	1	0	reserviert, keinen Code generieren
					1	1	1	reserviert, keinen Code generieren

Eine einzige Taste (in dieser Funktion ist das 1 Scan-Code), kann in ganz unterschiedlicher Weise einen oder mehrere Toggles beeinflussen:

- 000 = SET -> Toggle-Bit wird auf "1" gesetzt. 1,2 oder 3 Toggles gleichzeitig
- 001 = RESET -> dito, Toggle-Bit "0" setzen
- 010 = STICK-1 -> Setzt Toggle-Bit jetzt = 1, und beim LOSLASSEN, des nächsten Tastendruckes, wird Toggle wieder = 0 gesetzt. Das benutzt man z.B. in der 1-Handbedienung für zweite oder dritte Funktion der gleichen Taste.
- 011 = FLIP -> Invertiert den Wert des Toggles

Programmbeispiel:

Siehe : SCAN_TO_CHAR\$

SCAN_TO_TOGGLE

SCAN_TO_TOGGLE (*Code, Toggle*)

SCAN_TO_TOGGLE: Scan Toggles in Keyboards
(nur in Projektmodell 'PM_FULL')

Funktion: .

Code ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und übergibt den Scan-Code.

Toggle ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und übergibt die derzeit gültigen Toggle-Bits.

Funktionswert ist vom Typ LONG und enthält die neuen Toggle-Bits.

Vor der Anwendung der Funktion SCAN_TO_TOGGLE oder SCAN_TO_CHARS muß einmal die Funktion SCAN_TO_CHAR_SETUP ausgeführt worden sein.

Die Funktion SCAN_TO_TOGGLE liefert die neu gesetzten Toggle-Bits zurück. Welche Bits gesetzt oder zurückgesetzt werden, wird anhand des Scan-Codes und der Attribut-Tabelle entschieden, die zuvor mit SCAN_TO_CHAR_SETUP gesetzt worden ist.

Programmbeispiel:

Siehe : SCAN_TO_CHAR\$

SCRAMBLE\$

SCRAMBLE\$

SCRAMBLE\$ (Src\$, XP\$, Start, Step)

SCRAMBLE\$: Macht Stringbytes unkenntlich lt. Scramble-String
(nur in Projektmodell 'PM_FULL')

Funktion: Liefert als Funktionswert einen Sting mit Bytes aus **Src\$**, die durch eine XOR-Funktion verändert worden sind.

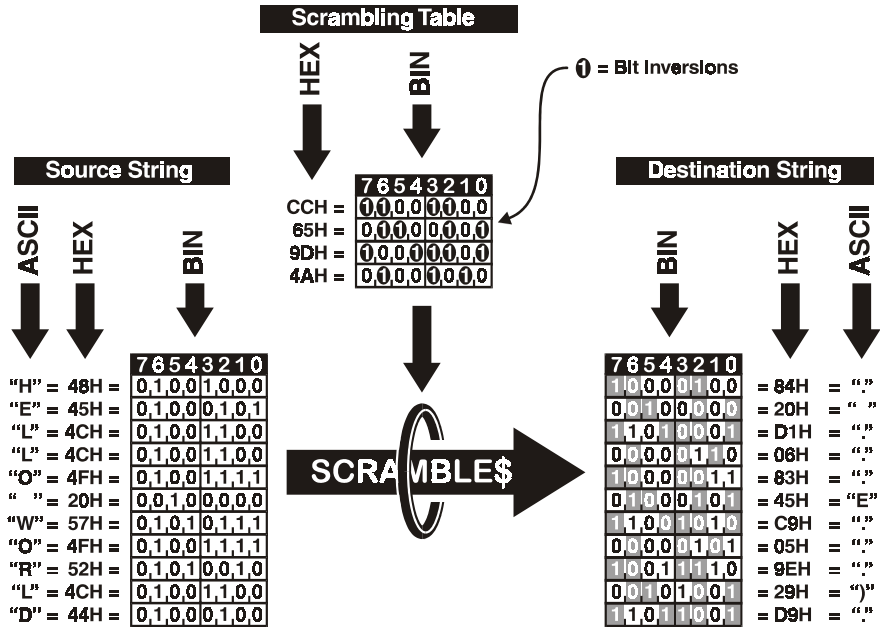
3

Src\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu verändernden Bytes.
XP\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Bytes, die für die XOR-Operation auf Src\$ verwendet werden.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt einen Anfangs-Offset innerhalb des Strings XP\$ mit den XOR-Bytes an.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite des Increments innerhalb des Strings XP\$ mit den XOR-Bytes an.
Funktionswert	ist vom Typ STRING und enthält die veränderten Bytes.

Scramble\$ führt eine XOR-Operation mit allen Bytes in Src\$ durch. In XP\$ sind die Bytes hinterlegt, mit denen die Operation ausgeführt wird. Die Bytes aus XP\$ werden der Reihe nach reih-um verwendet. Um nicht immer mit den gleichen Bytes aus XP\$ zu beginnen, kann ein Offset angegeben werden. Eine unterschiedliche Schrittweite der Increments innerhalb XP\$ verhindert, daß immer die gleiche Abfolge von Bytes für die XOR-Operation herangezogen wird. Die gleiche XOR-Operation noch einmal ausgeführt bringt wieder die ursprünglichen Bytes hervor.

Anwendungen für SCRAMBLE liegen im Bereich Verschlüsselung, eventuell in Kombination mit BIT_MIX\$ oder BYTE_MIX\$ für eine noch stärkere Verschlüsselung.

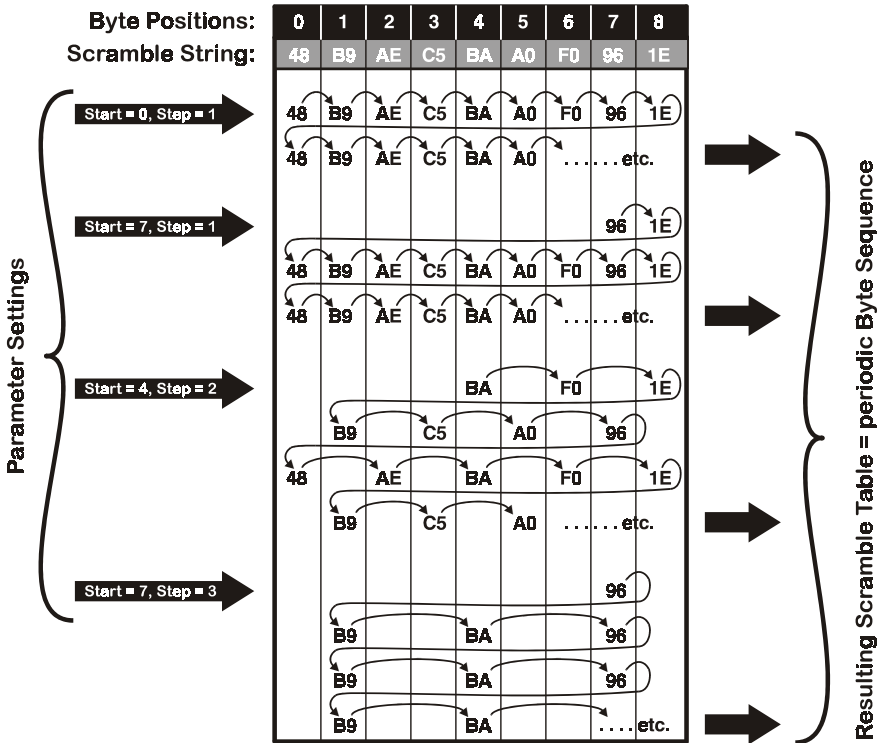
Die Abbildung zeigt, wie einzelne Bits durch die Operation im Zielstring invertiert erscheinen:



Converting example with a 4 byte SCRAMBLE-Table

SCRAMBLE\$

Um mit einem XOR-String angewendet auf denselben Quellstring nicht immer das selbe Ergebnis zu produzieren, ermöglichen die Parameter Start und Step, die XOR-Bytes in einer anderen Folge zu verwenden.



Generating different Scrambling Tables from 1 Scramble String

Programmbeispiel:

```

'-----
' Name: SCRAMBLE$.TIG
' Fuehrt SCRAMBLE mit 2 verschiedenen Offsets und Increments aus.
' CALL SHOW_STRING_AS_HEX ist auskommentiert.
' Verwenden Sie es, wenn Steuerzeichen das LCD stoeren.
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

A$ = "Hello ABC"
CALL SHOW_STRING_AS_HEX ( A$ ) ' zeige String an
PRINT #1, A$
B$ = SCRAMBLE$ ( A$, "00 FF C0 E0 07"% , 0, 1 )
CALL SHOW_STRING_AS_HEX ( B$ ) ' zeige String an
PRINT #1, B$

B$ = SCRAMBLE$ ( A$, "00 FF C0 E0 07"% , 1, -1 )
CALL SHOW_STRING_AS_HEX ( B$ ) ' zeige String an
PRINT #1, B$

S$ = "00 FF C0 E0 07"%
PRINT #1, SCRAMBLE$ ( SCRAMBLE$ ( A$, S$, 4, 2), S$, 4, 2 )

END                                     ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2" ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1 ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) ); ' als HEX
  NEXT
  PRINT #1, "" ' neue Zeile
END

```

3

Siehe auch: DISTRIBUTE\$, BIT_MIX\$, BYTE_MIX\$

SELECT\$

SELECT\$

SELECT\$ (*Index*, *Liste*)

SELECT\$: Selektiert String-Baustein an Hand eines Integer Wertes

Funktion: Liefert das Stringelement aus **Liste** zurück, welches durch **Index** indiziert wird. Das erste Element wird mit 0 initiiert. Die Listenelemente sind durch Trennzeichen getrennt.

3

Index	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.
Liste	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält als erstes Zeichen das Trennzeichen zwischen den Stringelementen. Es folgen die Stringelemente solange, bis das Trennzeichen zweimal hintereinander aufgeführt ist..
Funktionswert	ist vom Typ STRING und enthält das durch Index ausgewählte Stringelement aus dem übergebenen String.

Die Funktion SELECT\$ liefert ein Stringelement aus einer Liste zurück, welches mit dem Wert Index angezeigt wird. Das erste Element wird mit 0 indiziert.

Die Liste ist als String aufgebaut, der als erstes Zeichen das gewünschte Trennzeichen für die Listenelemente enthält, welches sich am Ende des Strings in zweifacher Form wiederholt. Die Elemente sind nicht in der Länge begrenzt, jedoch können sie nicht das Trennzeichen enthalten. Die Länge der Listenelemente kann variieren. Eine natürliche Grenze ergibt sich aus der maximalen Stringlänge.

SELECT\$ erlaubt die Umsetzung von Indices, z.B. Bytes, in Strings: SELECT ermittelt zur Laufzeit den ausgewählten Teilstring. Es entsteht eine Suchzeit die von der Größe des verwendeten Indexes abhängt. Anwendungsbeispiele:

- einfache Auswahl von Texten für Menü-Selektionen
- Expansion von Kurz-Mitteilungen.

Für String-Umsetzungen mit besonders hoher Geschwindigkeit: siehe STOSS\$.

Beispiele für Strings, die als Liste aufgebaut sind:

```
B$ = ",John,Mary,Harry,Mathilde,Nobody,,"
X = INDEX ( B$, "Mathilde" )           ` liefert 3 fuer X

C$ = "!0,00DM!1,25DM!2,50DM!3,75DM!5,00DM!6,25DM!!"

BIN$ = ":0000:0001:0010:0011:0100:0101:0110:0111:&
1000:1001:1010:1011:1100:1101:1110:1111:"
```

Anwendungen: Menüselektion, Konverter, Daten-Kompression/-Expansion

Programmbeispiel:

```
'-----  
' Name: SELECT$.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
  STRING List$                          ' STRING-Variable deklarieren  
  LONG Ind                              ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8  
  List$ = ":This:is:one:Test::"          ' Liste$ vorbesetzen  
  Ind = 2                                ' Index vorbesetzen  
  PRINT #1, "<l>";List$                    ' Ausgabe der Liste  
  PRINT #1, "Index:";Ind                 ' Ausgabe des Index  
  PRINT #1, SELECT$(Ind,List$)           ' Ausgabe des Stringelements  
END                                       ' Ende Task MAIN
```

3

Siehe auch: INDEX, STOSS

SERIAL_NO\$

SERIAL_NO\$

SERIAL_NO\$ (*Pos*, *Anzahl*)

SERIAL_NO\$: Liest Serien-Nummer, Installations-Code
(nur in Projektmodell 'PM_FULL')

Funktion: Liest **Anzahl** Bytes ab der angegebenen **Position** aus einem besonderen Bereich des Flash-Speichers.

3

Pos ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsadresse an, ab der die Bytes gelesen werden.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Anzahl der Bytes, die gelesen werden.

Funktionswert ist vom Typ STRING und enthält die gelesenen Bytes.

Der spezielle Flash-Speicherbereich liegt außerhalb des User-Flash und umfaßt 128 Bytes. Für den Anwender ergeben sich Adressen von 0 bis Adresse 127. Der Flash-Bereich ist nur mit SET_SERIAL_NO beschreibbar und wird nur zusammen mit Programm im Modul gelöscht.

Weitere Informationen über den Flashspeicher finden Sie im Hardware-Handbuch.

Das Beispielprogramm beschreibt den Flashbereich und liest den dort stehenden Wert (hier ein Text) wieder aus. Wenn das Programm nach einem Durchlauf neu gestartet wird, dann ist das Flash nicht gelöscht, und der Schreibversuch im Modus 0 führt zu einer Fehlermeldung. Es kann natürlich jederzeit mit SERIAL_NO\$ gelesen werden.

Programmbeispiel:

```

'-----
' Name: SERIAL_NO$.TIG
' Liest String aus speziellem Flashbereich
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG R                                ' SET_SERIAL_NO-Ergebnis
  STRING A$                             ' enthaelt Bytes zum Schreiben
  STRING S$                             ' zurueckgelesene Bytes
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  A$ = "Copyright"                      ' Bytes zu Schreiben
'                                     src,pos,mode
  R = SET_SERIAL_NO ( A$, 0, 0)          ' schreibe String im Modus 0
  PRINT #1, "mode 0 ->result:";R        ' Schreiberfolg
'                                     pos, Len
  S$ = SERIAL_NO$ ( 0, len ( A$) )      ' zuruecklesen
  PRINT #1, S$
END                                      ' Ende Task MAIN

```

3

Siehe auch: SET_SERIAL_NO, POKEM_FLASH

SET_BIT

SET_BIT

SET_BIT Variable, Bitnr

SET_BIT: Setze ein Bit in numerischer Variable auf ,1'

Instruktion: Setzt das Bit mit der Nummer **Bitnr** in der **Variablen** auf 1.

Variable ist vom Datentypen BYTE, WORD oder LONG.

Bitnr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welches Bit invertiert werden soll. Ist Bitnr größer als die Anzahl Bits der Variablen, dann verändert sich die Variable nicht.

Programmbeispiel:

```
'-----
' Name: SET_BIT.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  BYTE B                                ' BYTE-Variable deklarieren
  LONG L                                ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2> k 0.0.0.0.2UH<8><8> k 0.0.0.0.8"
  B = 0                                ' B vorbesetzen
  L = 0                                ' L vorbesetzen
  SET_BIT B,4                           ' Bit 4 setzen
  SET_BIT L,14                           ' Bit 14 setzen
  PRINT USING #1, "B = "; B, "L = "; L   ' B und L ausgeben
END                                       ' Ende Task MAIN
```

Siehe auch: RES_BIT

SET_LEN\$

SET_LEN\$ (String, Länge)

SET_LEN\$: Setzt die Länge eines Strings gezielt

Funktion: Setzt die **Länge** eines **Strings** unabhängig von dessen Inhalt.

String ist eine Variable vom Typ STRING.

Länge ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und gibt die neue Länge des Strings an. Wertebereich 0...maximale Stringlänge.

3

Diese Funktion liefert keinen Funktionswert zurück.

SET_LEN\$ setzt die Länge eines Strings. Der Inhalt des Strings ist zufällig, wenn der String dadurch verlängert wird. Für manche Funktionen muß ein String mit einer bestimmten Länge vorliegen, auch wenn die Funktion den Inhalt selbst beeinflusst. Das Setzen der Länge ist schneller als den String mit der Funktion FILL\$ zu füllen.

Programmbeispiel:

```
' -----
' Name: SET_LEN$.TIG
' setzt die Laenge eines Strings, ohne den Inhalt zu kontrollieren
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8

  A$ = "Hello World"                    ' A$ Text zuweisen
  PRINT #1, A$                          ' Auf LC-Display ausgeben
  SET_LEN$ ( A$, 5 )                   ' A$ ist nur 5 lang
  PRINT #1, A$                          ' Auf LC-Display ausgeben
  SET_LEN$ ( A$, 11 )                  ' A$ ist wieder 11 lang
  PRINT #1, A$                          ' der Inhalt blieb erhalten
END                                     ' Ende Task MAIN
```

SET_SERIAL_NO

SET_SERIAL_NO

SET_SERIAL_NO (*Data*, *Pos*, *Mode*)

SET_SERIAL_NO: Schreibt Serien-Nummer, Installations-Code, etc.
(nur in Projektmodell 'PM_FULL')

Funktion: Schreibt **Data** ab der angegebenen **Position** in einen besonderen Bereich des Flash-Speichers. Der Bereich kann nur durch ein Löschen des Programms im Modul wieder gelöscht werden. Es werden 1-Bits (=gelöschte Bits) zu 0-Bits geschrieben, nie können jedoch 0-Bits zu 1-Bits werden.

Data ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG, REAL oder STRING und enthält die zu schreibenden Bytes.

Pos ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsadresse an, ab der die Bytes geschrieben werden.

Mode ist vom Typ BYTE, WORD, LONG und bestimmt den Schreibmodus:
0 = Flashzellen müssen gelöscht sein (0FFh).
1 = Es werden Bits von „1“ zu „0“ geschrieben, wenn nötig.
Bits von „0“ zu „1“ zu schreiben führt zu einem Fehler.

Datenbyte	FLASH-Byte	FLASH anschliessend
1111 0000	1111 1001	1111 0000 OK
0011 0000	1111 1001	0011 0000 OK
0001 0000	1111 1001	0001 0000 OK
1111 0111	1111 1001	1111 0### Fehler

2: Dies führt niemals zu einem Schreibfehler. Es werden bitweise „0“-Bits geschrieben, wie dies im Datenbyte angegeben wird. „1“-Bits der Datenbytes haben gar keine Wirkung.

Funktionswert ist vom Typ LONG und zeigt das Ergebnis der Schreibfunktion folgendermaßen an:

- 1: ok. Es wurde wie beabsichtigt in das Flash geschrieben.
- 2: Offset oder Längenfehler, es wird nichts geschrieben.
- 3: Parameterfehler, es wird nichts geschrieben.
- 0...nnnn: in das Byte an der angegebenen Adresse hätte mindestens ein Bit von „0“ zu „1“ geschrieben werden müssen, was im Flash nicht möglich ist.

Der spezielle Flash-Speicherbereich liegt außerhalb des User-Flash und umfaßt 128 Bytes. Für den Anwender ergeben sich Adressen von 0 bis Adresse 127. Der Flash-Bereich ist nur mit SET_SERIAL_NO beschreibbar und wird nur zusammen mit Programm im Modul gelöscht.

Weitere Informationen über den Flashspeicher finden Sie im Hardware-Handbuch.

Das Beispielprogramm zeigt die Wirkungsweise der verschiedenen Schreibmodi. Wenn das Programm nach einem Durchlauf neu gestartet wird, dann ist das Flash nicht gelöscht, und der Schreibversuch im Modus 0 führt zu einer Fehlermeldung.

SET_SERIAL_NO

Programmbeispiel:

```
-----
' Name: SET_SERIAL_NO.TIG
' schreibt String in speziellen Flashbereich
' beobachte Fehler bei erneutem Schreibversuch im Modus 0
-----
TASK MAIN
    LONG R
    STRING A$
    STRING S$
    ' LCD-Treiber installieren (BASIC-Tiger)
    INSTALL DEVICE #1, "LCD1.TDD"
    ' LCD-Treiber installieren (TINY-Tiger)
    ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

    A$ = "FF F0 E0"%
    ' Bytes zu Schreiben
    '      adr,src,pos,mode
    R = SET_SERIAL_NO ( A$, 0, 0)
    ' schreiben 3 Bytes im Modus 0
    PRINT #1, "mode 0 ->result:";R
    ' Schreiberfolg
    S$ = SERIAL_NO$ ( 0, 8 )
    ' zuruecklesen
    CALL SHOW_STRING_AS_HEX ( S$ )
    ' als HEX anzeigen
    WAIT_DURATION 3000
    PRINT #1, "<1>";
    ' LCD loeschen

    A$ = "FF F0 C0"%
    ' letztes Byte: 1 Bit mehr =0
    '      adr,src,offs,len,mode
    R = SET_SERIAL_NO ( A$, 0, 1)
    ' schreiben 3 Bytes im Modus 1
    PRINT #1, "mode 1 ->result:";R
    ' Schreiberfolg
    S$ = SERIAL_NO$ ( 0, 3 )
    ' zuruecklesen
    CALL SHOW_STRING_AS_HEX ( S$ )
    ' als HEX anzeigen
    WAIT_DURATION 3000
    PRINT #1, "<1>";
    ' LCD loeschen

    A$ = "FF FF C0"%
    ' Bits muessten wieder 1 werden
    '      adr,src,offs,len,mode
    R = SET_SERIAL_NO ( A$, 0, 2)
    ' schreiben 3 Bytes im Modus 2
    PRINT #1, "mode 2 ->result:";R
    ' Schreiberfolg
    S$ = SERIAL_NO$ ( 0, 3 )
    ' zuruecklesen
    CALL SHOW_STRING_AS_HEX ( S$ )
    ' als HEX anzeigen
    ' Ende Task MAIN
END

-----
' zeigt den String als HEX-Bytes
-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
    BYTE I
    USING "UH<2><2> 0.0.0.0.2"
    ' Format HEX-Ausgabe
    FOR I = 0 TO LEN ( S$ ) - 1
    ' alle Bytes des Strings
        PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
    NEXT
    PRINT #1, ""
    ' neue Zeile
END
```

Siehe auch: SERIAL_NO\$, POKEM_FLASH

SET_SYSVARN

SET_SYSVARN (*FunktionsNr*, *Parameter*)

SET_SYSVARN: Setzt Systemparametert

Funktion: Setzt den Wert einer LONG-Systemvariablen. Systemvariable des numerischen Typs (außer REAL) werden mit dieser Funktion gesetzt.

FunktionsNr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG und ist die Nummer der Systemvariablen.
Parameter	kann verschiedene Bedeutungen haben oder ist manchmal irgendeine Zahl (Dummy).
Funktionswert	ist vom Typ LONG und meldet den Erfolgsstatus der Operation zurück. Eine Typwandlung bei der Zuweisung findet automatisch statt. 0: o.k. 1: Systemvariable existiert nicht. 2: Parameter falsch. Dieser Wert ist nicht möglich.

Den Funktionsnummern sind in der Include-Datei DEFINE_A.INC Namen zugeordnet, die Sie in untenstehender Tabelle finden.

Tip Benutzen Sie die Namen, dann brauchen Sie bei Änderungen im Nummernsystem Ihre Programme nicht anzupassen. 'THIS_TASK' wird für die aktuelle Task eingesetzt.

Binden Sie die Datei 'DEFINE_A.INC' ein, um die Funktionsnamen zu verwenden, da sich die Funktionsnummern im Rahmen der Weiterentwicklung von Tiger-BASIC® ändern können. Funktionen von SET_SYSVARN:

Funktionsname	Nr	2. Parameter	Beschreibung
SHIFT_SLOWER	<00>	Wert	Setzt einen Parameter der Funktionen SHIFT_IN und SHIFT_OUT, um die Clock-Pulse dieser Funktionen zu verlangsamen. Wertebereich: 0...15

SET_SYSVARN

Programmbeispiel:

```
'-----
' Name: SET_SYSVARN.TIG
' setzt Systemvariable fuer SHIFT_IN und SHIFT_OUT,
' um Clock zu verlangsamen
'-----
#include DEFINE_A.INC

TASK MAIN                                ' Beginn Task MAIN
    LONG R                               ' Rueckmeldung von SET_SYSVARN
    LONG D                               ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
DIR_PORT 8,0                            ' Port 8 als Ausgang
LL_IOPORT_OUT 8, 00000000b               ' Ruhepegel
D = 0AAAAAAAAAH                          ' D vorbesetzen
SHIFT_OUT 8, 0, 1, D, 32                 ' 32 Bits werden ausgegeben
R = SET_SYSVARN ( SHIFT_SLOWER, 15 )     ' setze Systemparameter
PRINT #1, "result:";R
SHIFT_OUT 8, 0, 1, D, 32                 ' langsamer ausgegeben
END                                       ' Ende Task MAIN
```

SET_TASK_PRIO

SET_TASK_PRIO Name, Priorität

SET_TASK_PRIO: Setzt Priorität für Task-Switching

Instruktion: Setzt die Priorität der Task **Name** auf den in **Priorität** angegebenen Wert.

SET_TASK_PRIO startet die Task nicht.

Priorität ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG. Die niedrigste Priorität ist 0, die höchste 255. Priorität 0 stoppt die Task.

Die Prioritätszahl gibt an, wieviel Zeitscheiben die Task erhält. Die momentan gültige Priorität läßt sich mit Hilfe einer Systemabfrage 'TASK_PRIO' ermitteln (siehe SYSVARN ab Seite 460). Die prozentuale CPU-Leistung der aktuellen Task ergibt sich aus:

Priorität der aktuellen Task * 100 / Summe aller Taskprioritäten

Setzen Sie die Taskprioritäten mit kleinen Werten im gewünschten Verhältnis, so wechseln die Tasks sich schneller ab, als dies bei gleichen Zahlenverhältnissen, aber größeren Zahlen der Fall wäre. Beispiel für 3 Tasks:

	schneller Taskwechsel	langsamer Taskwechsel
Task1	3	30
Task2	5	50
Task3	7	70
Gesamtprioritätszahl	15	150

In obigem Beispiel hat zum Beispiel die Task 1 eine CPU-Leistung von 20% ($3 \cdot 100 / 15 = 20\%$).

Die Prioritäten der Tasks lassen sich bereits beim Starten der Task in der Instruktion RUN_TASK einstellen. Mit SET_TASK_PRIO können die Prioritätsverhältnisse der Anwendung während der Laufzeit dynamisch verändert werden.

SET_TASK_Prio

Programmbeispiel:

3

```
'-----
' Name: SET_TASK_Prio.TIG
'-----
' Die Variablen I und J werden in unterschiedlichen Tasks mit
' unterschiedlicher Prioritaet hochgezaehlt. Dabei laeuft T1
' schneller als T2. Beim Erreichen eines bestimmten Wertes wird
' die Prioritaet umgekehrt, d.h. T2 laeuft nun schneller als T1.
'-----
LONG Flag,I,J                                ' Globale LONG-Variablen

TASK MAIN                                    ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
Flag = 0                                     ' Flag auf 0 setzen
RUN_TASK T1,100                             ' Task T1 starten
RUN_TASK T2,10                             ' Task T2 starten
LOOP 9999999                                ' sehr viele Loops
    IF (I > 100000) AND (Flag = 0) THEN      ' Falls I>100000 und Flag=0
        Flag = 1                            ' Flag auf 1 setzen
        SET_TASK_Prio T1,10                 ' Prioritaet von T1 aendern
        SET_TASK_Prio T2,10                 ' Prioritaet von T2 aendern
    ENDIF
    PRINT #1, "<1>I ="; I                    ' I auf LC-Display anzeigen
    PRINT #1, "J ="; J                      ' J auf LC-Display anzeigen
    WAIT DURATION 100                       ' 100 ms warten
ENDLOOP
END                                           ' Ende Task MAIN

'-----
' Task T1 inkrementiert die Variable I
'-----
TASK T1                                     ' Beginn Task T1
    FOR I=0 TO 200000                       ' Zaehlschleife fuer I
    NEXT                                    ' naechster Wert fuer I
END                                         ' Ende Task T1

'-----
' Task T2 inkrementiert die Variable J
'-----
TASK T2                                     ' Beginn Task T2
    FOR J=0 TO 200000                       ' Zaehlschleife fuer J
    NEXT                                    ' naechster Wert fuer J
END                                         ' Ende Task T2
```


SET_TICKS

SET_TICKS (Wert)

SET_TICKS: Setzt 1 ms System-Taktzähler auf neuen Wert

Funktion: Setzt den internen Zähler der System-Taktes auf neuen **Wert**.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der neue Zählerwert der Systemticks.

3

Diese Funktion liefert keinen Funktionswert zurück.

Der Systemzähler zählt in Millisekunden und beginnt nach Reset mit 0. Der Tick-Zähler läuft nach 2.147.483.647 Ticks, also nach 24 Tagen über. Der Zähler bleibt im positiven Wertebereich der LONG-Zahlen. Der aktuelle Zählerstand wird mit der Funktion TICKS ausgelesen. Die Differenz zwischen zwei Tick-Zählerständen ermittelt die Funktion DIFF_TICKS und berücksichtigt dabei, daß der Zähler gerade übergelaufen sein kann.

Programmbeispiel:

```

' -----
' Name: SET_TICKS.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
LONG T                                  ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
T = TICKS()                             ' Speichert Ticks in T
PRINT #1, "Ticks:";T
SET_TICKS ( T - 10 )                     ' setze Ticks zurück
PRINT #1, "Ticks:";TICKS()
END                                       ' Ende Task MAIN

```

Siehe auch: TICKS, DIFF_TICKS

SGN

SGN

SGN (Wert)

SGN: Vorzeichen eines numerischen Wertes

Funktion: Liefert das Vorzeichen von Wert zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

-1	wenn der Wert negativ ist.
0	wenn der Wert Null ist.
+1	Wenn der Wert positiv ist.

Programmbeispiel:

```
'-----  
' Name: SGN.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, "SGN ="; SGN (-34.444)        ' Ausgabe auf LC-Display  
  PRINT #1, "SGN ="; SGN (0)             ' Ausgabe auf LC-Display  
  PRINT #1, "SGN ="; SGN (1101b)         ' Ausgabe auf LC-Display  
END                                       ' Ende Task MAIN
```

Siehe auch: SIGNEXT, INT, ABS

SHIFT_IN

SHIFT_IN *Log_iPortadr, Datenpin, Taktpin, Variable, Anzahl, P1*

SHIFT_IN: Getaktete, serielle Eingabe für externe Chips

Instruktion: Schiebt die angegebene **Anzahl** Bits in die **Variable**. An einem internen Port mit der Portadresse **Log_iPortadr** wird ein Pin als **Datenpin** und ein Pin als **Taktpin** verwendet. Das BASIC-Tiger[®]-Modul gibt den Takt vor, mit dem die Datenbits eingelesen werden.

Log_iPortadr	ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische interne Portadresse an.
Datenpin	ist eine Konstante vom Typ BYTE, WORD oder LONG und bestimmt, an welchem Pin die Datenbits eingelesen werden.
Taktpin	ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die Bitnummer des Pins an, welcher als Taktpin verwendet wird.
Variable	ist vom Datentyp BYTE, WORD, LONG oder STRING.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt bei einer numerischen Variablen an, wieviele Bits eingelesen werden. Ist die angegebene Anzahl größer als die Anzahl Bits der Variablen, dann werden überzählige Bits ignoriert. Bei positiver Anzahl wird das niederwertigste Bit zuerst gelesen, bei negativer Anzahl das höherwertigste. Bei Variablen vom Typ STRING legt die Größe des Strings fest, wieviele Bytes eingelesen werden, und Anzahl gibt an, wieviele Bits für jedes Byte verwendet werden. Eine REAL-Zahl (8 Bit * 8 Byte) kann geshiftet werden, wenn sie vorher mit den Funktionen RTL oder LREAL und HREAL in eine LONG-Zahl verwandelt worden ist.
P1	ist vom Datentyp BYTE, WORD, LONG und bestimmt, ob vor dem ersten Clock bereits ein Bit gelesen wird, oder ob erst nach dem ersten Clock das erste Bit eingelesen wird: 0: erstes Bit lesen, dann ersten Clock geben. 1: erst Clock geben, dann das erste Bit lesen.

SHIFT_IN liest einen seriellen Datenstrom getaktet ein. Der Takt wird vom BASIC-Tiger[®]-Modul vorgegeben. Ein Taktimpuls wird erzeugt, indem der Taktpin zweimal invertiert wird. Der Ruhepegel kann daher vorgegeben werden. Eine SHIFT_IN-Instruktion liest maximal soviele Bits ein, wie die Variable aufnehmen kann. Bei STRINGs gibt die Anzahl vor, wieviele Bits für jedes eingelesene Byte verwendet werden. Die Größe des Strings legt fest, wieviele Bytes eingelesen werden.

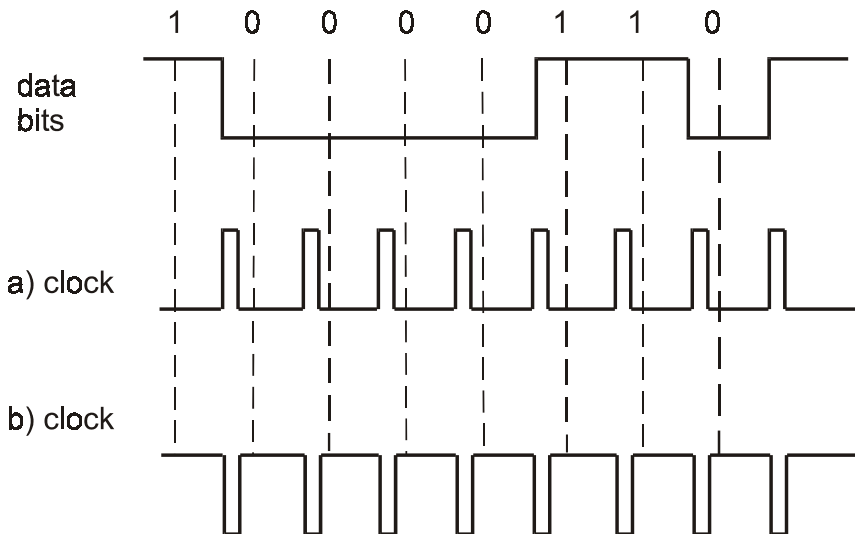
SHIFT_IN

Anwendungen der SHIFT_IN-Instruktion sind z.B. Schieberegister oder auch Verbindung mehrerer Module, wenn die seriellen Leitungen bereits für andere Zwecke belegt sind.

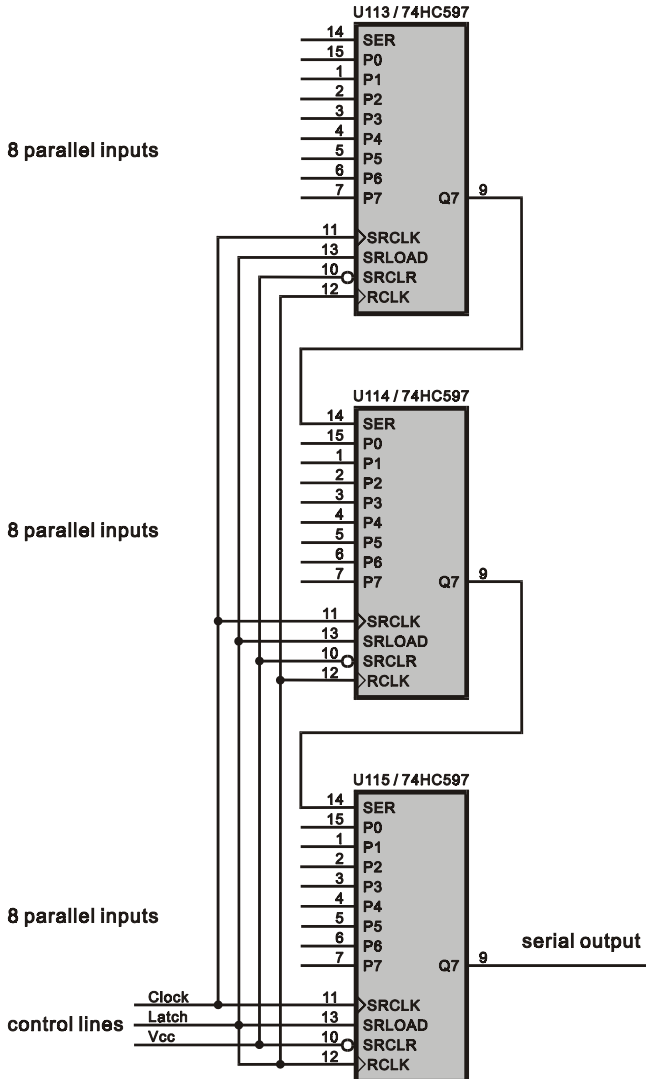
Die Abbildung zeigt die Übertragung des Zeichens 'a' einmal mit einem Ruhepegel 'low' (a) und einmal mit einem Ruhepegel 'high' (b) auf der Taktleitung.

"a" = 61h = 01100001

3



Die Abbildung zeigt ein serielles Schieberegister (3x 74HC597) mit 24 Bit parallelem Eingang. Durch einen Latch-Impuls werden die parallelen Eingangsdaten übernommen und werden dann mit der SHIFT_IN-Instruktion eingelesen: SHIFT_IN 8, 0, 1, LVAR, 24 (lies an Port 8 mit Bit 0 als Daten- und Bit 1 als Clockpin in die Variable LVAR 24 Bit aus dem Schieberegister ein).



SHIFT_IN

Programmbeispiel:

```
'-----
' Name: SHIFT_IN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG D                                ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<8><8>*kV2.2.2.2.2"
  DIR_PIN 8, 0, 1                        ' Port 8 Pin 0 als Eingang
  DIR_PIN 8, 1, 0                        ' Port 8 Pin 1 als Ausgang
  DIR_PIN 8, 2, 0                        ' Port 8 Pin 2 als Ausgang
                                          ' Ruhepegel
  LL_IPORT_OUT 8, 00000100b              ' clock low, latch high
  LL_IPORT_PULSE 8, 00000100b, 1         ' latch-Impuls f. Schiebereg.
  SHIFT_IN 8, 0, 1, D, 24                ' 24 Bits werden eingelesen
  PRINT_USING #1, "<1>"; D                ' und angezeigt
END                                        ' Ende Task MAIN
```

Siehe auch: SHIFT_OUT, Device-Treiber SER2

SHIFT_OUT

SHIFT_OUT *Log_iPortadr, Datenpin, Taktpin, Variable, Anzahl*

SHIFT_OUT: Getaktete, serielle Ausgabe zu externen Chips

Instruktion: Überträgt die angegebene **Anzahl** Bits der **Variablen** auf den **Datenpin**. Für jedes Datenbit wird ein Taktimpuls durch zweimaliges Invertieren des **Taktpins** erzeugt. Datenpin und Taktpin liegen auf einem internen Port mit der Adresse **Log_iPortadr**.

Log_iPortadr	ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die logische interne Portadresse an.
Datenpin	ist eine Konstante vom Typ BYTE, WORD oder LONG und bestimmt, an welchem Pin die Datenbits eingelesen werden.
Taktpin	ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt an, welcher Pin als Taktpin verwendet wird.
Variable	ist eine Konstante, Variable oder ein Ausdruck vom Datentyp BYTE, WORD, LONG oder STRING und wird ausgegeben.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt bei numerischen Variablen an, wieviele Bits geschrieben werden. Bei positiver Anzahl wird das niederwertigste Bit zuerst gesendet, bei negativer Anzahl das höherwertigste. Bei BYTE, WORD oder LONG ist die Anzahl auf 32 Bits begrenzt. Überzählige Bits bei BYTE oder WORD werden als 0-Bits geschrieben. Bei Variablen vom Typ STRING gilt die angegebene Anzahl für jedes Byte und ist auf 8 begrenzt. Es wird immer der ganze String geschrieben. Eine REAL-Zahl (8 Bit * 8 Byte) kann geschiftet werden, wenn sie vorher mit den Funktionen RTL oder LREAL und HREAL in eine LONG-Zahl verwandelt worden ist.

SHIFT_OUT schreibt einen seriellen Datenstrom getaktet auf einen Output-Pin eines internen Ports. Ein zweiter Pin des Ports wird als Taktpin verwendet. Der Takt wird vom BASIC-Tiger[®]-Modul vorgegeben. Ein Taktimpuls wird erzeugt, indem der Taktpin zweimal invertiert wird. Der Ruhepegel kann daher vorgegeben werden. Eine SHIFT_OUT-Instruktion schreibt maximal 32 Bits bei den ganzzahligen numerischen Variablen, immer 64 Bits bei REAL-Zahlen und bei Strings immer den ganzen String. Bei Strings gibt die Anzahl an, wieviele Bits von jedem Byte geschrieben werden.

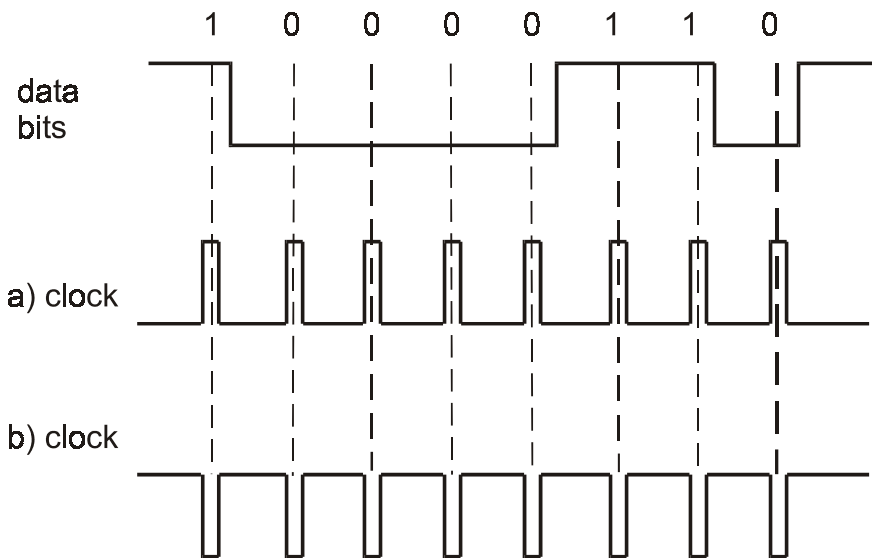
SHIFT_OUT

Anwendungen der SHIFT_OUT-Instruktion sind z.B. Schieberegister oder auch Verbindung mehrerer Module, wenn die seriellen Leitungen bereits für andere Zwecke belegt sind.

Die Abbildung zeigt die Übertragung des Zeichens 'a' einmal mit einem Ruhepegel 'low' (a) und einmal mit einem Ruhepegel 'high' (b) auf der Taktleitung.

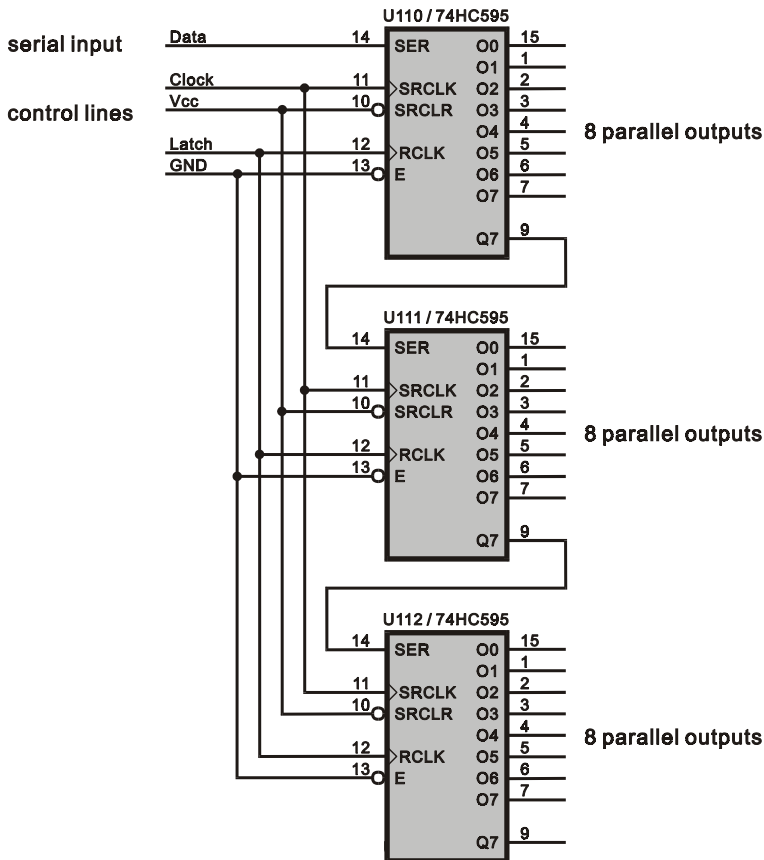
"a" = 61h = 01100001

3



Bei den BASIC-Tiger[®]-A-Modulen ist ein Clock ca. 0,5 Mikrosekunden lang und der Abstand beträgt standardmäßig 4,6 Mikrosekunden. Mit SET_SYSVARN läßt sich eine Systemvariable setzen, die den Clock verlangsamt, bis zu einer Clocklänge von 20 Mikrosekunden und einem Clock-Abstand von 43 Mikrosekunden.

Die Abbildung zeigt ein serielles Schieberegister (3x 74HC595) mit 24 Bit parallelem Ausgang. Die Daten werden mit der SHIFT_OUT-Instruktion in das Schieberegister geschrieben: SHIFT_OUT 8, 0, 1, LVAR, 24 (schreibe an Port 8 mit Bit 0 als Daten- und Bit 1 als Clockpin aus der Variable LVAR die niederwertigen 24 Bit in das Schieberegister). Durch einen Latch-Impuls werden die Ausgangsdaten übernommen und erscheinen an den parallelen Ausgängen.



SHIFT_OUT

Programmbeispiel:

```
'-----  
' Name: SHIFT_OUT.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
    LONG D                               ' LONG-Variable deklarieren  
    DIR_PORT 8,0                          ' Port 8 als Ausgang  
    LL_IPORT_OUT 8, 00000000b            ' Ruhepegel  
    D = 0AAAACF0FH                       ' D vorbesetzen  
    SHIFT_OUT 8, 0, 1, D, 24              ' 24 Bits werden ausgegeben  
    LL_IPORT_PULSE 8, 00000100b, 1        ' latch-Impuls f. Schiebereg.  
END                                        ' Ende Task MAIN
```

Siehe auch: SHIFT_IN

SHIFTL\$

SHIFTL\$ (*String*, *Shifts*)

SHIFTL\$: Schiebt die Bits eines Strings um 1...8 Positionen (nur in Projektmodell 'PM_FULL')

Funktion: Schiebt alle Bits in **String** um die Anzahl **Shifts**. Der-String liegt in der Sortierung 'niedrigstes Byte zuerst' vor (SHIFT String-Typ 'least-significant-byte-first').

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu schiebenden Bits. Die Bytes im String sind 'high-bytes-first' angeordnet.

Shifts ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von -8...+8 und bestimmt, um wieviel Bits geschoben wird.

Funktionswert ist vom Typ STRING und enthält die geschobenen Bits.

Die Bits in **String** werden um die angegebene Anzahl Shifts geschoben.

Positiver Wert für Shift: da bei positiver Angabe für Shift der binäre Wert der Bitkette größer wird und im String das höchste Byte an letzter Stelle liegt, wird insgesamt im String von den linken niederwertigeren Bytes zu den rechten höherwertigeren Bytes geschoben. Innerhalb der einzelnen Bytes werden die Bits jedoch wie immer nach links geschoben. Die dabei auf der linken Seite überlaufenden Bits erscheinen im rechtsliegenden nächst-höherwertigen Byte im untersten Bit. Im niederwertigen Byte wird von rechts ein Null-Bit eingeschoben (diese '0' ist hervorgehoben), das höchstwertige Bit verschwindet.

Beispiel für Shift = +1:

	LSB		MSB
Quellstring	11111111	10100001	11110000
in den Bytes	←	←	←
Überläufe	→		
Ergebnis-String	11111110	01000011	11110001

SHIFTL\$

Negativer Wert für Shift: da bei negativer Angabe für Shift der binäre Wert der Bitkette kleiner wird und im String das niedrigste Byte an erster Stelle liegt, wird insgesamt im String von den rechten höherwertigeren Bytes zu den linken niederwertigeren Bytes geschoben. Innerhalb der einzelnen Bytes werden die Bits jedoch nach rechts geschoben. Die dabei auf der rechten Seite überlaufenden Bits erscheinen im linksliegenden nächst-niederwertigen Byte im höchsten Bit. Das niederwertigste Bit verschwindet und im höchstwertigen Byte wird von links ein Null-Bit eingeschoben (diese '0' ist hervorgehoben).

Beispiel für Shift = -1:

	LSB		MSB
Quellstring	11111100	10100001	11110000
in den Bytes	→	→	→
Überläufe	←	←	←
Ergebnis-String	11111110	00100001	01111000

3

Programmbeispiel:

```

-----
' Name: SHIFTL$.TIG
-----

TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  A$ = "<A1H><FOH>"                      ' das is A$
  PRINT #1, "0:";                        ' zeige A$
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTL$ ( A$, 1 )                  ' schiebe A$ einmal
  PRINT #1, "1:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTL$ ( A$, 1 )                  ' schiebe A$ einmal
  PRINT #1, "2:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTL$ ( A$, 1 )                  ' schiebe A$ einmal
  PRINT #1, "3:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
END                                        ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )              ' zeige CNT Bytes binaer
  BYTE B, I, J                            ' lokale Variable
  FOR I = 1 TO CNT                        ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ) )   ' hole ein Byte
    FOR J = 1 TO 8                        ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN             ' teste ob bit 1 ist
        PRINT #1, "1";                    ' drucke Bit
      ELSE                                 ' falls kein Rest
        PRINT #1, "0";                    ' drucke Bit
      '
    NEXT J                                ' B eins nach rechts (neues Bit)
  NEXT I                                  ' Leerzeichen zwischen Bytes
END                                        ' Ende Unterprogramm

```

3

Siehe auch: SHIFTM\$

SHIFTM\$

SHIFTM\$

SHIFTM\$ (*String*, *Shifts*)

SHIFTM\$: Schiebt die Bits eines Strings um 1...8 Positionen
(nur in Projektmodell 'PM_FULL')

Funktion: Schiebt alle Bits in **String** um die Anzahl **Shifts**. Der-String liegt in der Sortierung 'höchstwertiges Byte zuerst' vor (SHIFT String-Typ 'most-significant-byte-first').

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu schiebenden Bits. Die Bytes im String sind 'low-bytes-first' angeordnet.

Shifts ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von -8...+8 und bestimmt, um wieviel Bits geschoben wird.

Funktionswert ist vom Typ STRING und enthält die geschobenen Bits.

Die Bits in **String** werden um die angegebene Anzahl Shifts geschoben.

Positiver Wert für Shift: da bei positiver Angabe für Shift der binäre Wert der Bitkette größer wird und im String das höchste Byte an erster Stelle liegt, wird insgesamt im String von den rechten niederwertigeren Bytes zu den linken höherwertigeren Bytes geschoben. Innerhalb der einzelnen Bytes werden die Bits ebenfalls nach links geschoben. Die dabei auf der linken Seite überlaufenden Bits erscheinen im linksliegenden nächst-höherwertigen Byte im untersten Bit. Im niederwertigen Byte wird von rechts ein Null-Bit eingeschoben (diese '0' ist hervorgehoben), das höchstwertige Bit verschwindet.

Beispiel für Shift = +1:

	MSB		LSB
Quellstring	10111110	10100001	11110001
in den Bytes	←	←	←
Überläufe	←		
Ergebnis-String	01111101	01000011	11100010

Negativer Wert für Shift: da bei negativer Angabe für Shift der binäre Wert der Bitkette kleiner wird und im String das niedrigste Byte an letzter Stelle liegt, wird insgesamt im String von den linken höherwertigeren Bytes zu den rechten niederwertigeren Bytes geschoben. Innerhalb der einzelnen Bytes werden die Bits ebenfalls nach rechts geschoben. Die dabei auf der rechten Seite überlaufenden Bits erscheinen im rechtsliegenden nächst-niederwertigen Byte im höchsten Bit. Das niederwertigste Bit verschwindet und im höchstwertigen Byte wird von links ein Null-Bit eingeschoben.

Beispiel für Shift = -1:

	MSB		LSB
Quellstring	11111100	10100001	11110000
in den Bytes	→	→	→
Überläufe	→	→	→
Ergebnis-String	01111110	01010000	11111000

SHIFTM\$

Programmbeispiel:

```
'-----
' Name: SHIFTM$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "<AlH><FlH>"                      ' das is A$
  PRINT #1, "0:";                        ' zeige A$
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTM$ ( A$, 1 )                 ' schiebe A$ einmal
  PRINT #1, "1:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTM$ ( A$, 1 )                 ' schiebe A$ einmal
  PRINT #1, "2:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
  A$ = SHIFTM$ ( A$, 1 )                 ' schiebe A$ einmal
  PRINT #1, "3:";                        ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                ' binaer
END                                        ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )             ' zeige CNT Bytes binaer
  BYTE B, I, J                           ' lokale Variable
  FOR I = 1 TO CNT                       ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ))   ' hole ein Byte
    FOR J = 1 TO 8                       ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN            ' teste ob bit 1 ist
        PRINT #1, "1";                   ' drucke Bit
      ELSE                                ' falls kein Rest
        PRINT #1, "0";                   ' drucke Bit
      ENDIF
      B = B SHL 1                         ' B eins nach rechts (neues Bit)
    NEXT J
    PRINT #1, " ";                        ' Leerzeichen zwischen Bytes
  NEXT I
END                                        ' Ende Unterprogramm
```

Siehe auch: SHIFTL\$

SIGNAL_ADD

SIGNAL_ADD (Str1, Start1, Anz, Step, Size, Str2, Start2, Rescale)

SIGNAL_ADD: Überlagert 2 Signale additiv
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion verwendet den Argumentstring **Str1** sowohl als Ein- als auch als Ausgang. Zu den Signalwerten in **Str1** werden die Signalwerte aus String **Str2** addiert. Die Startpositionen in **Str1** und in **Str2** sowie die **Anzahl** der Werte werden angegeben.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. Str1 ist gleichzeitig der Zielstring.
Start1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position im String Str1 an, ab der die Addition beginnt.
Anz	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die addiert werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Str2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten.
Start2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position im String Str2 an, ab der die Addition beginnt.
Rescale	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Skalierungsfaktor des Ergebnis-Signals in 10*Prozent an, d.h. 1000 bedeutet 100%.

SIGNAL_ADD

Funktionswert gibt es bei dieser Funktion nicht. Die Daten in **Str1** werden direkt verändert.

SIGNAL_ADD addiert ab der angegebenen Position Str1 über die angegebene Anzahl Signalwerte die Werte aus dem String Str2. Nach der Addition werden die neuen Signalwerte mit dem Skalierungsfaktor multipliziert. Wird die maximale Amplitude überschritten, dann werden die Werte auf den Maximalwert begrenzt.

In Tonanwendungen wirkt SIGNAL_ADD z.B. wie ein Mischpult mit 2 Eingängen und einem gemeinsamem Lautstärkeregler am Ausgang.

Programmbeispiel:

```
' -----
' Name: SIGNAL_ADD.TIG
' -----

TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2> 2 2 2 2"              ' fuer Ausgabe als HEX
  A$="3F 40 41 CF D0 D1"%               ' das is A$
  A$="3F 40 41 CF D0 D1"%               ' das is A$
  PRINT #1, "A$";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
  PRINT #1, " + + + + + "
  PRINT #1, "B$";                       ' zeige A$
  FOR I = 0 TO LEN(B$)-1
    PRINT_USING #1, " ";ASC(MID$(B$,I,1));
  NEXT
  SIGNAL_ADD (A$,0,6,1,8,B$,0,1000)     ' addiere A4 und B$
  PRINT #1, "A$";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
END                                      ' Ende Task MAIN
```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_AVERAGE

SIGNAL_AVERAGE (*String*, *Start*, *Anzahl*, *Step*, *Size*)

SIGNAL_AVERAGE: Ermittelt den Signal-Mittelwert, (DC-Komponente)
(nur in Projektmodell 'PM_FULL')

Funktion: Ermittelt über die angegebene **Anzahl** von Signalwerten den Mittelwert (Average) aus diesen Werten im **String**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Mittelwertberechnung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit zur Mittelwertberechnung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Funktionswert	ist vom Typ BYTE, WORD oder LONG.

SIGNAL_AVERAGE berechnet den Mittelwert einer angegebenen Anzahl von Signalwerten. Der Mittelwert stellt zum Beispiel bei Sound-Daten den Nullpunkt eines AC-Signals dar.

SIGNAL_AVERAGE

Programmbeispiel:

```
'-----
' Name: SIGNAL_AVERAGE.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="<20H><40H><20H><40H>"               ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1                 ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  A = SIGNAL_AVERAGE (A$,0,6,1,8)        ' berechne Mittelwert
  PRINT_USING #1, "<13><10>average="; A    ' zeige A
END                                        ' Ende Task MAIN
```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_BIGGER

SIGNAL_BIGGER (Str, Start, Anz, Step, Size, Level)

SIGNAL_BIGGER: Ermittelt Signal Samples die größer sind als Vergleichswert (nur in Projektmodell 'PM_FULL')

Funktion: Zählt ab der Position **Start** über die angegebene **Anzahl** Signalwerte in **String** diejenigen Signalwerte, die größer sind als **Level**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der das Zählen beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit zum Zählen herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Level	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Signalpegel vor.
Funktionswert	ist vom Typ BYTE, WORD oder LONG und liefert die Anzahl der Signalwerte, die über dem angegebenen Level liegen.

SIGNAL_BIGGER zählt ab der angegebenen Position über die angegebene Anzahl Signalwerte die Werte, die größer als **Level** sind.

SIGNAL_CUT_AMP

SIGNAL_CUT_AMP (*String*, *Start*, *Anz*, *Step*, *Size*, *Center*, *Amp*)

SIGNAL_CUT_AMP: Begrenzt Signal-Amplitude
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert den Argumentstring **String**. Über die angegebene **Anzahl** Signalwerte in **String** wird die maximale Amplitude relativ zum Wert **Center** begrenzt.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. Str1 ist gleichzeitig der Zielstring.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Begrenzung beginnt.
Anz	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die bearbeitet werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.
Amp	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den AMPimalwert der Signalwerte an.
Funktionswert	Die Daten in String werden direkt verändert.

SIGNAL_CUT_AMP begrenzt den maximal vorkommende Amplitude im Bereich der angegebenen Anzahl Signalwerte ab der angegebenen Startposition im String. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden.

Programmbeispiel:

```

'-----
' Name: SIGNAL_CUT_AMP.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>  2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="3F 40 41 CF D0 D1"%              ' das is A$
  PRINT #1, "A$";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
  PRINT #1, "cut amplitude at 40h";
  SIGNAL_CUT_AMP (A$,0,6,1,8,80h,40h)  ' beschneide Amplitude
  PRINT #1, "A$";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
END                                      ' Ende Task MAIN

```

3

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_CUT_MAX

SIGNAL_CUT_MAX

SIGNAL_CUT_MAX (*String*, *Start*, *Anz*, *Step*, *Size*, *Max*)

SIGNAL_CUT_MAX: Begrenzt Maximalwerte im Signal
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert ggf. den Argumentstring **String**. Die Signalwerte in **String** werden auf den Wert **Max** begrenzt. Die Startposition in **String** sowie die **Anzahl** der Werte werden angegeben.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. Str1 ist gleichzeitig der Zielstring.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Begrenzung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die bearbeitet werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
MAX	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Maximalwert der Signalwerte an.
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in String werden direkt verändert.

SIGNAL_CUT_MAX ersetzt in String ab der angegebenen Position Start über die angegebene Anzahl von Signalwerten diejenigen, die größer sind als der angegebene Maximalwert, durch den angegebenen Maximalwert.

Programmbeispiel:

```

-----
' Name: SIGNAL_CUT_MAX.TIG
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>  2 2 2 2"              ' fuer Ausgabe als HEX
  A$="<10H><40H><7FH><80H><81H><0FFH>"    ' das is A$
  PRINT #1, "A$";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
  PRINT #1, "cut at 80h"
  SIGNAL_CUT_MAX (A$,0,6,1,8,80h)        ' beschneide Maximum
  PRINT #1, "A$";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, " ";ASC(MID$(A$,I,1));
  NEXT
END                                      ' Ende Task MAIN

```

3

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_CUT_MIN

SIGNAL_CUT_MIN (*String*, *Start*, *Anz*, *Step*, *Size*, *Min*)

SIGNAL_CUT_MIN: Begrenzt Minimalwerte im Signal
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert ggf. den Argumentstring **String**. Die Signalwerte in **String** werden nach unten auf den Wert **Min** begrenzt. Die Startposition in **String** sowie die **Anzahl** der Werte werden angegeben.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. Str1 ist gleichzeitig der Zielstring.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Begrenzung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die bearbeitet werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Min	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Mindestwert der Signalwerte an.
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in String werden direkt verändert.

SIGNAL_CUT_MIN ersetzt in String ab der angegebenen Position Start über die angegebene Anzahl von Signalwerte diejenigen, die kleiner sind als der angegebene Mindestwert, durch den Mindestwert.

SIGNAL_GAP_END

SIGNAL_GAP_END (Str, Start, Anz, Step, Siz, Center, Ampl)

SIGNAL_GAP_END: Sucht eine Lücke im Signal
(nur in Projektmodell 'PM_FULL')

Funktion: Sucht ab der Position **Start** über die angegebene **Anzahl** Signalwerte in **String** das Ende einer Signallücke (engl. 'gap'=Lücke). Dabei ist die angegebene **Amplitude** relativ zum Nullwert **Center** noch zulässig, um 'kein Signal' zu bedeuten, d.h. innerhalb dieser Amplitude liegt die Lücke des Signals.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Suche nach dem Ende einer Signallücke beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Suche nach dem Ende einer Signallücke herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.
Funktionswert	ist vom Typ BYTE, WORD oder LONG. Ergebnis = -1 → Gap Ende nicht gefunden. Ergebnis = nnnn → Gap endet genau auf dieser Position. Dieser Wert gehört bereits nicht mehr zur Lücke.

SIGNAL_GAP_END

SIGNAL_GAP_END sucht ab der angegebenen Position über die angegebene Anzahl Signalwerte nach einem Signalwert, der größer ist als durch **Ampl** vorgegeben. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden. Es wird die erste Position in den Signalwerten zurückgeliefert, die bereits nicht mehr zur Lücke gehört. Die Funktion SIGNAL_GAP_END ist schneller als die Funktion SIGNAL_GAPI_END, erkennt jedoch Spikes im Signal als das Ende der Lücke.

SIGNAL_GAPI_END

SIGNAL_GAPI_END (*Str, Start, Anz, Step, Siz, Avr, Ctr, Ampl*)

SIGNAL_GAPI_END: Sucht eine Lücke im Signal, Kurzzeitintegration
(nur in Projektmodell 'PM_FULL')

Funktion: Wie **SIGNAL_GAP_END**, jedoch wird bei der Suche über mehrere Werte integriert, so daß trotz Spikes die Lücke richtig erkannt wird. Diese Funktion bestimmt ab der Position **Start** über die angegebene **Anzahl** Signalwerte in **String** das Ende einer Signallücke (engl. 'gap'=Lücke). Dabei ist die angegebene **Amplitude** relativ zum Nullwert **Ctr** noch zulässig, um 'kein Signal' zu bedeuten, d.h. innerhalb dieser Amplitude liegt das Rauschen des Signals.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und gibt die Position in String an, ab der die Suche nach dem Ende einer Signallücke beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Suche nach dem Ende einer Signallücke herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Avr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und gibt die Anzahl der Signalwerte an, über die bei der Suche integriert wird. Mögliche Werte: 8, 16, 32, 64, 128.
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE , WORD oder LONG und legt den Nullwert des Signals fest.

SIGNAL_GAPI_END

Funktionswert ist vom Typ BYTE, WORD oder LONG.
Ergebnis = -1 → Gap Ende nicht gefunden.
Ergebnis = nnnn → Gap endet genau auf dieser Position.
Dieser Wert gehört bereits nicht mehr zur Lücke.

SIGNAL_GAPI_END sucht ab der angegebenen Position über die angegebene Anzahl Signalwerte nach einem Bereich ohne Amplituden größer als durch den Wert **Ampl** vorgegeben. Bei der Suche wird über die Anzahl **Avr** Werte gemittelt, so daß einzelne Spikes im Signal die Suche nicht vorzeitig beenden. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden. Es wird die erste Position in den Signalwerten zurückgeliefert, die bereits nicht mehr zur Lücke gehört. Die Funktion SIGNAL_GAP_END ist langsamer als die Funktion SIGNAL_GAPI_END, toleriert jedoch Spikes im Signal.

3

SIGNAL_MAX

SIGNAL_MAX (*String*, *Start*, *Anzahl*, *Step*, *Size*)

SIGNAL_MAX: Ermittelt Maximalwert eines Signals
(nur in Projektmodell 'PM_FULL')

Funktion: Ermittelt über die angegebene **Anzahl** Signalwerte den maximalen Signalwert in **String**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Maximalwert-Ermittlung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit zur Maximalwert-Ermittlung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Funktionswert	ist vom Typ BYTE, WORD oder LONG.

SIGNAL_MAX ermittelt den Maximalwert aus der angegebenen Anzahl von Signalwerten. Dieser Wert wird zum Beispiel gebraucht, wenn der maximal mögliche Skalierungsfaktor für die Funktion SIGNAL_SCALE bestimmt werden soll, oder die Aussteuerungs-Verhältnisse ermittelt werden müssen.

SIGNAL_MAX

Programmbeispiel:

```
'-----
' Name: SIGNAL_MAX.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="<7FH><40H><C0H><30H><F0H>"         ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  A = SIGNAL_MAX (A$,0,5,1,-8)           ' suche Maximum
  PRINT_USING #1, "<13><10>Max=";A;       ' zeige A
END                                       ' Ende Task MAIN
```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_MAX_AMPL**SIGNAL_MAX_AMPL (*String, Start, Anzahl, Step, Size, Center*)**

SIGNAL_MAX_AMPL: Ermittelt größte Amplitude eines Signals
(nur in Projektmodell 'PM_FULL')

Funktion: Bestimmt über die angegebene **Anzahl** von Signalwerten in **String** die maximale Amplitude relativ zum Wert **Center**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Maximalwert-Ermittlung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit zur Maximalwert-Ermittlung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.
Funktionswert	ist vom Typ BYTE, WORD oder LONG.

SIGNAL_MAX_AMPL ermittelt die maximal vorkommende Amplitude im Bereich einer angegebenen Anzahl von Signalwerten. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden.

SIGNAL_MAX_AMPL

Programmbeispiel:

```
'-----
' Name: SIGNAL_MAX_AMPL.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="<7FH><40H><COH><10H><F0H>"         ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  A = SIGNAL_MAX_AMPL (A$,0,5,1,-8,7FH) ' suche maximale Amplitude
  PRINT_USING #1, "<13><10>Max. Amplitude=";A;
END                                      ' Ende Task MAIN
```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_MIN

SIGNAL_MIN (*String, Start, Anzahl, Step, Size*)

SIGNAL_MIN: Ermittelt Minimalwert eines Signals
(nur in Projektmodell 'PM_FULL')

Funktion: Bestimmt über die angegebene **Anzahl** an Signalwerten den Minimalwert in **String**.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Minimalwert-Ermittlung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit zur Minimalwert-Ermittlung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Funktionswert	ist vom Typ BYTE, WORD oder LONG.

SIGNAL_MIN ermittelt den Minimalwert über eine angegebene Anzahl von Signalwerten.

SIGNAL_MIN

Programmbeispiel:

```
'-----
' Name: SIGNAL_MIN.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  LONG A
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="<7FH><40H><C0H><30H><F0H>"         ' das is A$
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  A = SIGNAL_MIN (A$,0,5,1,-8)           ' suche Minimum
  PRINT_USING #1, "<13><10>Max=";A;      ' zeige A
END                                       ' Ende Task MAIN
```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_MUL

SIGNAL_MUL (*Str1, Start1, Anz, Step1, Size, Str2, Start2, Step2, Div*)

SIGNAL_MUL: Multipliziert 2 Signale
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert den Argumentstring **Str1**. Die Signalwerte in **Str1** werden mit den Signalwerten aus String **Str2** multipliziert. Die Startpositionen in **Str1** und in **Str2** sowie die **Anzahl** der Werte werden angegeben.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. Str1 ist gleichzeitig der Zielstring.
Start1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position im String Str1 an, ab der die Addition beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die multipliziert werden sollen.
Step1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Str1 an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Str2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten.
Start2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position im String Str2 an, ab der die Multiplikation beginnt.
Step2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Str2 an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden

SIGNAL_MUL

Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.

Div

ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und an, durch welchen Wert die Werte des Ergebnis-Signales dividiert werden.

Funktionswert

gibt es bei dieser Funktion nicht. Die Daten in **Str1** werden direkt verändert.

3

SIGNAL_MUL multipliziert die Signalwerte in String Str1 ab der angegebenen Position Start1 über die angegebene Anzahl Signalwerte mit den Werten aus dem String Str2. Nach der Multiplikation werden die neuen Signalwerte zur Skalierung durch den Wert Div dividiert. Überschreitungen des maximal zulässigen Zahlenbereiches werden nicht begrenzt.

SIGNAL_NEXT_GAP

SIGNAL_NEXT_GAP (Str, Start, Anz, Step, Size, Center, Ampl, Width)

SIGNAL_NEXT_GAP: Sucht nächste Lücke im Signal
(nur in Projektmodell 'PM_FULL')

Funktion: Sucht ab der Position **Start** über die angegebene **Anzahl** von Signalwerten in **String** den ersten Bereich ohne Signal (engl. 'gap'=Lücke). Dabei ist die angegebene **Amplitude** relativ zum Nullwert **Center** noch zulässig, um 'kein Signal' zu bedeuten, d.h. innerhalb dieser Amplitude liegt die Lücke im Signal. Für das Erkennen auf Signallücke, muß die in **Width** angegebene Anzahl von Signalwerten das Kriterium für eine Lücke erfüllen.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Suche nach der Signallücke beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Suche nach der Signallücke herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.
Width	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Mindestlänge der Signallücke fest.

SIGNAL_NEXT_GAP

Funktionswert ist vom Typ BYTE, WORD oder LONG.
Ergebnis = -1 → Gap Ende nicht gefunden.
Ergebnis = nnnn → Gap endet genau auf dieser Position. Dieser Wert gehört bereits nicht mehr zur Lücke.

SIGNAL_NEXT_GAP sucht ab der angegebenen Position über die angegebene Anzahl Signalwerte nach einem Bereich mit Amplituden die kleiner oder gleich dem durch **Ampl** angegebenen Wert sind. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden. Es wird die erste Position in den Signalwerten zurückgeliefert, die bereits zur Lücke gehört. Die Funktion SIGNAL_NEXT_GAP ist Schneller als die Funktion SIGNAL_NEXT_GAPI, jedoch gelten Spikes als Signal, was eventuell dazu führt, daß eine Lücke nicht gefunden wird.

3

SIGNAL_NEXT_GAPI

SIGNAL_NEXT_GAPI (Str, Start, Anz, Step, Siz, Avr, Ctr, Ampl, Width)

SIGNAL_NEXT_GAPI: Sucht nächste Lücke im Signal, Kurzzeitintegration
(nur in Projektmodell 'PM_FULL')

Funktion: Wie SIGNAL_NEXT_GAP, jedoch wird bei der Suche über mehrere Werte integriert, so daß trotz Spikes die Lücke noch erkannt wird. Diese Funktion bestimmt ab der Position **Start** über die angegebene **Anzahl** Signalwerte in **String** die erste Phase ohne Signal (engl. 'gap'=Lücke). Dabei ist die angegebene **Amplitude** relativ zum Nullwert **Ctr** noch zulässig, um als 'kein Signal' aufgefaßt zu werden, d.h. innerhalb dieser Amplitude liegt das Rauschen des Signals.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Suche nach einer Signallücke beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Suche nach einer Signallücke herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Avr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Signalwerte an, über die bei der Suche integriert wird. Mögliche Werte: 8, 16, 32, 64, 128.
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.

SIGNAL_NEXT_GAPI

Width ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Mindestlänge der Signallücke fest.

Funktionswert ist vom Typ BYTE, WORD oder LONG.
Ergebnis = -1 → Gap Ende nicht gefunden.
Ergebnis = nnnn → Gap endet genau auf dieser Position. Dieser Wert gehört bereits nicht mehr zur Lücke.

3

SIGNAL_NEXT_GAPI sucht ab der angegebenen Position über die angegebene Anzahl Signalwerte nach einem Bereich mit Amplituden die kleiner oder gleich dem durch **Ampl** angegebenen Wert sind. Der Wert für Center muß bekannt sein oder vorher mit der Funktion SIGNAL_AVERAGE festgestellt werden. Es wird die erste Position in den Signalwerten zurückgeliefert, die bereits zur Lücke gehört. Die Funktion SIGNAL_NEXT_GAP ist Schneller als die Funktion

Die Funktion SIGNAL_NEXT_GAPI arbeitet langsamer als die Funktion SIGNAL_NEXT_GAP, jedoch werden Spikes in Signallücken des Signals in gewissem Umfang toleriert.

SIGNAL_OFFSET

SIGNAL_OFFSET (*String*, *Start*, *Anz*, *Step*, *Size*, *Offset*)

SIGNAL_OFFSET: Verschiebt Signalpegel um einen Offset
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet den und verändert ggf. Argumentstring **String**. Ab der Position **Start** wird über die angegebene **Anzahl** von Signalwerten in **String** ein Offset auf die Signalwerte addiert.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Signaldaten. String ist gleichzeitig der Zielstring.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Offset-Operation beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die bearbeitet werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Offset	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Wert an, um den die Signalwerte im Pegel verschoben werden.
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in String werden direkt verändert.

SIGNAL_OFFSET verschiebt im Bereich der angegebenen Anzahl Signalwerte ab der angegebenen Startposition im String die Signalwerte um **Offset**. SIGNAL_OFFSET wird dazu benutzt, Signale in einen bestimmten Pegelbereich zu verschieben.

SIGNAL_SCALE

SIGNAL_SCALE (*String*, *Start*, *Anz*, *Step*, *Size*, *Center*, *Scale*, *MAmpl*)

SIGNAL_SCALE: Verstärkt oder Dämpft das Signal
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert ggf. den Argumentstring **String**. **String** wird unter Berücksichtigung des Signal-Nullwertes **Center** sowie der maximalen Amplitude **MAmpl** über die angegebene **Anzahl** Signalwerte mit einem Skalierungsfaktor **Scale** multipliziert und dabei direkt verändert. Dies entspricht bei Tonaufzeichnungen zum Beispiel einer Lautstärkeveränderung.

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Skalierung beginnt.
Anz	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Skalierung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Center	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt den Nullwert des Signals fest.
Scale	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Multiplikationsfaktor in 10*Prozent an, d.h. ein Scale -Wert von 1000 bedeutet 100%.

MAmplitude ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die maximale Amplitude nach der Skalierung fest. Dieser Wert ist immer positiv und gibt die maximale Auslenkung von **Center** an.

Funktionswert gibt es bei dieser Funktion nicht. Die Daten in **String** werden direkt verändert.

SIGNAL_SCALE multipliziert die angegebene Anzahl Signalwerte relativ zum Wert Center mit dem Skalierungsfaktor. Wird die maximale Amplitude überschritten, dann werden die betroffenen Signalwerte auf den Maximal- bzw. Minimalwert begrenzt. Diese Funktion entspricht zum Beispiel bei Sound-Daten einer Lautstärke-Einstellung.

Programmbeispiel:

```

'-----
' Name: SIGNAL_SCALE.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"           ' fuer Ausgabe als HEX
  A$="<7FH><40H><C0H><30H><F0H>"         ' das is A$
  PRINT #1, "A$=";                       ' zeige A$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  SIGNAL_SCALE (A$,0,5,1,-8,7FH,800,7FH) ' Scaliere auf 80%
  PRINT #1, "<13><10>scaled to 80% around 7FH"
  PRINT #1, "A$=";                       ' zeige Ergebnis in B$
  FOR I = 0 TO LEN(A$)-1
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
END                                        ' Ende Task MAIN

```

Siehe auch: weitere SIGNAL-Funktionen

SIGNAL_SMALLER

SIGNAL_SMALLER (Str, Start, Anz, Step, Siz, Level)

SIGNAL_SMALLER: Ermittelt Signal Samples die kleiner sind als Vergleichswert (nur in Projektmodell 'PM_FULL')

Funktion: Zählt ab der Position **Start** über die angegebene **Anzahl** Signalwerte in **String** diejenigen Signalwerte, die kleiner als **Level** sind.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Signaldaten.
Start	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Position in String an, ab der die Zählung beginnt.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die zur Zählung herangezogen werden sollen.
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Level	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Signalpegel vor.
Funktionswert	ist vom Typ BYTE, WORD oder LONG und liefert die Anzahl der Signalwerte, die unter dem angegebenen Level liegen.

SIGNAL_SMALLER zählt ab der angegebenen Position über die angegebene Anzahl Signalwerte die Werte, die kleiner als **Level** sind.

SIGNAL_SMOOTH

SIGNAL_SMOOTH (*String, Endwert, Anzahl, Ends, Step, Size*)

SIGNAL_SMOOTH: Flacht die Enden eines Signals ab
(nur in Projektmodell 'PM_FULL')

Funktion ohne Funktionswert. Diese Funktion bearbeitet und verändert ggf. den Argumentstring **String**. **String** wird an den mit **Ends** bestimmten Enden über die angegebene **Anzahl** von Signalwerten bearbeitet und dabei direkt verändert. Die Signalwerte an den zu bearbeitenden Stellen werden zu den Enden hin dem Wert **Endwert** angenähert.

3

String	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu verändernden Signalwerten.
Endwert	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Wert an, an den sich die bearbeiteten Daten annähern.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Signalwerte fest, die mit der Smooth-Funktion an den Enden des Strings bearbeitet werden sollen.
Ends	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt fest, welche Enden des Strings bearbeitet werden sollen: 1: Anfang 2: Ende 3: Anfang + Ende
Step	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Schrittweite in Signalwerten an. Zum Beispiel ist bei Stereo-Sounddaten mit abwechselnden Daten für rechten und linken Kanal die Schrittweite 2. Schrittweite 0 wird automatisch auf 1 gesetzt.
Size	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Größe und Art der Datenwerte: 8: BYTE ohne Vorzeichen (0...255) -8: BYTE mit Vorzeichen (-128...+127) 16: WORD ohne Vorzeichen (0...65535) -16: WORD mit Vorzeichen (-32768...+32767)
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in String werden direkt verändert.

SIGNAL_SMOOTH

SIGNAL_SMOOTH bearbeitet die mit **Ends** ausgewählten Enden eines Signal-strings mit der sogenannten Smooth-Funktion. Die Signalwerte werden an den Enden so in ihrer Amplitude relativ zu **Endwert** amplitudenmoduliert, daß dort eine Amplitudenrampe von 0% ... 100% entsteht. Bei Audiosignalen wird die Smooth-Funktion verwendet um Signale „weich“ ein- und auszublenzen.

SIGNEXT

SIGNEXT (Wert, Anzahl)

SIGNEXT: Vorzeichenrichtige Wert-Erweiterung

Funktion: Liefert die ab der angegebenen **Anzahl** Bits vorzeichenrichtig erweiterte Zahl zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Bits an, die der Funktionswert erhalten soll. Das höchste Bit ist das Vorzeichenbit.

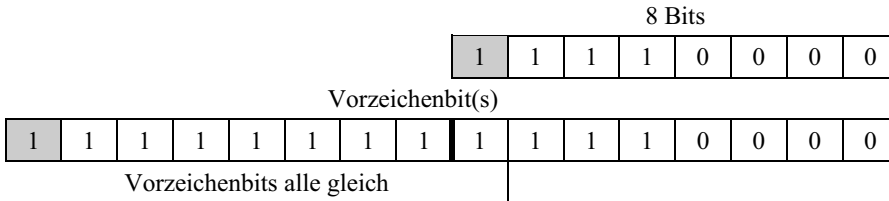
Funktionswert ist vom Typ BYTE, WORD oder LONG.

In den folgenden Beispielen wird je eine positive und eine negative Zahl mit unterschiedlicher Bitanzahl in ein BYTE vorzeichenerweitert:

BYTE	B vorher	SIGNEXT (B, n)
SIGNEXT (B, 2)	00100101	00000001
SIGNEXT (B, 2)	00101111	11111111
SIGNEXT (B, 3)	00101001	00000001
SIGNEXT (B, 3)	00101101	11111101
SIGNEXT (B, 4)	10110001	00000001
SIGNEXT (B, 4)	10011001	11111001
SIGNEXT (B, 5)	11000111	00000111
SIGNEXT (B, 5)	11010111	11110111
SIGNEXT (B, 6)	10001111	00001111
SIGNEXT (B, 6)	11101111	11101111
SIGNEXT (B, 7)	10101101	00101101
SIGNEXT (B, 7)	01101101	11101101
BYTE nach WORD		
SIGNEXT (L, 5)	11000111	00000000 00000111
SIGNEXT (L, 5)	11010111	11111111 11110111

SIGNEXT

W = SIGNEXT (B, 8)



3

Die Vorzeichenerweiterung wird angewendet, wenn vorzeichenbehaftete Werte unterschiedlicher Wortbreite, z.B. 5-Bit-Meßwerte eines Multimeters oder 12-Bit-Werte eines A/D-Wandlers in Tiger-BASIC[®] weiterverarbeitet werden. Ebenso können die 4-Byte großen LONG-Werte vorzeichenrichtig mit weniger Platzverbrauch im Flash abgelegt oder z.B. seriell übertragen werden, wenn nur kleinere Werte vorkommen, indem die LONG-Zahl einfach abgeschnitten wird. Wenn solche Werte weiterverarbeitet werden, müssen sie ihr Vorzeichen wieder bekommen.

Hier ein Beispiel, was aus einem negativen LONG-Wert in einer WORD-Variablen wird:

```
WORD W
LONG L
L = -87
W = L
PRINT #1, L, W           \ L = -87, W = 65449
```

Programmbeispiel:

```

-----
' Name: SIGNEXT.TIG
-----
' Dieses Programm zeigt Beispiele fuer die Funktion SIGNEXT(var,pos).
' Dabei wird der Wert des Bits 'pos' in alle hoeherwertigeren Bits
' der Variablen 'var' uebernommen.
-----

TASK Main                                     ' Beginn Task MAIN
  BYTE B                                     ' BYTE-Variable deklarieren
  WORD W                                     ' WORD-Variable deklarieren
  LONG L                                     ' LONG-Variable deklarieren

' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"

' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  USING "UH<8><8> 0.0.0.4 4"                ' Formatstring fuer Ausgabe

  B = 70h                                   ' B vorbesetzen (112)
  PRINT USING #1, "<1>Bit: 7, B: "; B        ' Ausgabe auf LC-Display
  B = SIGNEXT (B,7)                         ' zu BYTE: F0h (240)
  W = SIGNEXT (B,7)                         ' zu WORD: FFFF0h (65520)
  L = SIGNEXT (B,7)                         ' zu LONG: FFFFFFF0h (-16)
  PRINT USING #1, "BYTE = "; B              ' Ausgabe auf LC-Display
  PRINT USING #1, "WORD = "; W              ' Ausgabe auf LC-Display
  PRINT USING #1, "LONG = "; L              ' Ausgabe auf LC-Display
  WAIT DURATION 3000                        ' 3 Sek warten

  W = 7000h                                 ' W vorbesetzen (28672)
  PRINT USING #1, "<1>Bit:14, W: "; W         ' Ausgabe auf LC-Display
  B = SIGNEXT (W,14)                        ' zu BYTE: 00h (0)
  W = SIGNEXT (W,14)                        ' zu WORD: F000h (61440)
  L = SIGNEXT (W,14)                        ' zu LONG: FFFF000h (-4096)
  PRINT USING #1, "BYTE = "; B              ' Ausgabe auf LC-Display
  PRINT USING #1, "WORD = "; W              ' Ausgabe auf LC-Display
  PRINT USING #1, "LONG = "; L              ' Ausgabe auf LC-Display
  WAIT DURATION 3000                        ' 3 Sek warten

  L = 47654321h                             ' L vorbesetzen (1197818657)
  PRINT USING #1, "<1>Bit:19, L: "; L         ' Ausgabe auf LC-Display
  B = SIGNEXT (L,19)                        ' zu BYTE: 21h (33)
  W = SIGNEXT (L,19)                        ' zu WORD: 4321h (17185)
  L = SIGNEXT (L,19)                        ' zu LONG: FFFD4321h (-179423)
  PRINT USING #1, "BYTE = "; B              ' Ausgabe auf LC-Display
  PRINT USING #1, "WORD = "; W              ' Ausgabe auf LC-Display
  PRINT USING #1, "LONG = "; L              ' Ausgabe auf LC-Display
END                                           ' Ende Task MAIN

```

3

Siehe auch: SGN, INT, ABS

SIN

SIN

SIN (rad)

SIN: SINUS, Real-Werte

Funktion: Liefert den Sinus von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $-4.5 \cdot 10^{15}$ bis $4.5 \cdot 10^{15}$ und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von -1 bis 1. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
'-----
' Name: SIN.TIG
'-----
#define PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                             ' Beginn Task MAIN
  LONG DEG                           ' LONG-Variable deklarieren
  REAL RAD                           ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                    ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                  ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>SIN ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; SIN (RAD)     ' Sinuswert ausgeben
  WAIT DURATION 1000                 ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                   ' Ende Task MAIN
```

Siehe auch: COS, TAN, ASIN, ACOS, ATAN, SINH

SINH

SINH (rad)

SINH: SINUS hyperbolicus, Real-Werte

Funktion: Liefert den Sinushyperbolicus von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von -709 bis 709 und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```

'-----
' Name: SINH.TIG
'-----
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                              ' Beginn Task MAIN
  LONG DEG                             ' LONG-Variable deklarieren
  REAL RAD                             ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                     ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                   ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>SINH ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; SINH (RAD)     ' Sinus hyperbolicus ausgeben
  WAIT DURATION 1000                  ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                  ' Ende Task MAIN

```

Siehe auch: COS, TAN, ASIN, ACOS, ATAN, COSH

SQRT

SQRT

SQRT (Wert)

SQRT: Wurzel

Funktion: Liefert die Quadratwurzel von **Wert** zurück.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL größer oder gleich 0.

Funktionswert ist vom Typ REAL. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Programmbeispiel:

```
'-----  
' Name: Sqrt.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
  LONG L                                ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"  
  FOR L = 0 TO 100                        ' Werte von 0 bis 100  
    PRINT #1, "<1>Square root of "; L    ' Ausgabe der Quadratwurzel  
    PRINT USING #1, "="; Sqrt(LTR(L))  ' auf das LC-Display (in REAL)  
    WAIT DURATION 1000                  ' 1 Sek warten  
  NEXT                                  ' naechster Wert  
END                                    ' Ende Task MAIN
```

Siehe auch: LN, LOG, LD, EXP

START_FIFO

START_FIFO Name, Integralweite, Startwert[, Fehlerbehandlung]

START_FIFO: Fifo löschen + vorbereiten für INTEGRAL_FIFO

Instruktion: Setzt die Startbedingungen des FIFO-Puffers **Name** für die anschließende Integralbildung mit INTEGRAL_FIFO. Füllt den **Startwert** in die ersten durch **Integralweite** bestimmten Elemente des FIFO-Puffers.

Integralweite ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und stimmt mit der Größe des zur Integralbildung verwendeten FIFO-Puffers überein. Die Integralweite ist immer eine Potenz von 2.

Startwert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und füllt den FIFO-Puffer mit diesem Wert.

Um einen zur Integralbildung verwendeten FIFO-Puffer zu initialisieren, muß dieser vor der ersten Benutzung mit der Instruktion START_FIFO gefüllt werden. Eventuell kann der erste Meßwert als Startwert dienen. Steht kein Startwert zur Verfügung, so beginnt der erste Integralwert mit einem zufälligen Wert, der sich erst nach einer Anzahl von Integralwertbildungen auf den richtigen Wert einstellt.

Näheres zur Integralbildung finden Sie bei der Beschreibung der Funktion INTEGRAL_FIFO auf Seite 259.

Weitere Informationen über FIFO-Puffer finden Sie auf Seite 30.

START_FIFO

Programmbeispiel:

```
' -----
' Name: START_FIFO.TIG
' -----
TASK Main                                ' Beginn Task MAIN
  FIFO BUFFER(32) of LONG                ' FIFO-Buffer definieren
  WORD W1                                ' WORD-Variable deklarieren
  LONG L1                                ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 0, 80h, 8
  CLEAR_FIFO BUFFER                      ' FIFO-Buffer loeschen
  START_FIFO BUFFER,32,0                 ' Startwert setzen
  FOR W1 = 0 TO 31                       ' Einrichten der FOR-Schleife
    L1 = RND(1)                          ' Zufallswert generieren
    PRINT #1, "<1>Elements:";W1           ' W1 auf LC-Display ausgeben
    PRINT #1, "          L1:";L1         ' L1 auf LC-Display ausgeben
    INTEGRAL_FIFO BUFFER,0,L1            ' Integral bilden
    PRINT #1, "Integral:";L1             ' L1 auf LC-Display ausgeben
    WAIT_DURATION 1000                   ' 1 Sek warten
  NEXT                                  ' naechster Wert fuer W1
END                                      ' Ende Task MAIN
```

Siehe auch: PUT_FIFO, UNGET_FIFO, GET_FIFO.

STOP_TASK

STOP_TASK Name

STOP_TASK: Stoppt Task

Instruktion: Stoppt die Task **Name**.

Die Ausführung dieses Prozesses wird unmittelbar nach Abarbeitung dieser Instruktion unterbrochen. Es ist zu beachten, daß alle ungepufferten I/Os dieses Prozesses zum Erliegen kommen. Statische, quasistatische, sowie alle gepufferten I/Os (wie z.B. serielle Ein/Ausgänge) laufen ggf. noch weiter.

Eine Task kann sich selbst sowie jede andere Task stoppen. Alle lokalen Variablen dieser Task bleiben erhalten. Die Task kann mit der Instruktion CONT_TASK fortgeführt werden.

3

STOP_TASK

Programmbeispiel:

```
'-----
' Name: STOP_TASK.TIG
'-----
' Zaehlt die Variablen I (in Task MAIN) und J (in Task T1) hoch.
' Ist I zwischen 300 und 500, ist die Task T1 gestoppt und J
' wird in dieser Zeit nicht weiter hochgezählt.
'-----
LONG I, J                                ' Globale Var. deklarieren

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
RUN_TASK T1                              ' TASK T1 starten
FOR I = 0 TO 1000                          ' Zaehlschleife fuer I
  IF I > 300 THEN                          ' Ist I > 300 ?
    STOP_TASK T1                          ' Bei I > 300 Task T1 stoppen
  ENDIF                                  ' Ende der IF-Abfrage
  IF I > 500 THEN                          ' Ist I > 500 ?
    CONT_TASK T1                          ' Bei I > 500 Task T1 starten
  ENDIF                                  ' Ende der IF-Abfrage
  PRINT #1, "<1>I ="; I                    ' I anzeigen
  PRINT #1, "J ="; J                      ' J anzeigen
  WAIT DURATION 100                       ' 100 ms warten
NEXT                                      ' naechster Wert
END                                        ' Ende Task MAIN

'-----
' TASK T1 zaehlt globale Variable J hoch
'-----
TASK T1                                  ' Beginn Task T1
  FOR J = 0 TO 1000                        ' Zaehlschleife fuer J
    WAIT DURATION 100                     ' 100 ms warten
  NEXT                                    ' naechster Wert
END                                        ' Ende Task T1
```

STOS\$

STOS\$ (*String1*, *Pos1*, *String2*, *Anzahl*)

STOS\$ (*String1*, *Pos1*, *String2*, *Pos2*, *Anzahl*)

STOS\$: Schnelle Ausführung: String nach String.

Funktion: **String-To-String** schreibt **Anzahl** Bytes aus **String2** in den **String1** an die angegebene **Position**.

String1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und entspricht dem Quellstring für den Funktionswert.
Pos1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition in String1 an. Wenn die angegebene Position außerhalb des Strings liegt, wird String1 verlängert.
String2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die einzufügenden Zeichen.
Pos2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition in String2 an. Wenn nur 4 Parameter angegeben werden, dann ist die letzte Zahl die Anzahl und Pos2 entfällt und wird als 0 angenommen.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der abzulegenden Bytes an. Ist die Anzahl negativ, werden die Zeichen aus String2 rückwärts in String1 geschrieben. Ist Anzahl nicht angegeben, wird der String bis zum Ende verwendet.
Funktionswert	ist vom Typ STRING und kann mit String1 identisch sein, jedoch nicht mit String2 . Der Funktionswert enthält String1 mit den eingebauten Bytes aus String2.

STOS\$ legt in einem String eine angegebene Anzahl von Bytes aus String2 ab. Dort vorhandene Bytes werden überschrieben. Wird die Anzahl positiv angegeben, legt STOS\$ die Bytes in der Reihenfolge wie in String2, bei negativer Anzahl in umgekehrter Reihenfolge ab. Ist die Anzahl zu groß, werden maximal soviele Bytes geschrieben, wie String2 enthält.

Der String wird verlängert, wenn die angegebene Position außerhalb des Strings liegt. Die maximale Stringlänge wird jedoch nie überschritten. Es kann dadurch auch eine "Lücke" im String entstehen, die undefinierte Bytes enthält. Wird die maximale Länge überschritten, entsteht als Ergebnis ein Leerstring.

STOSS

Die Verarbeitungsgeschwindigkeit dieser Funktion ist wesentlich höher, wenn die Zuweisungsvariable, die den Funktionswert aufnimmt, mit der Variablen String1 identisch ist (**A\$=STOSS(A\$,0,B\$,LEN(B\$))**).

Tip

Die Funktionen NTOSS\$, NFROMS, RTOSS\$, RFROMS, STOSS\$ erhöhen die Verarbeitungsgeschwindigkeit und sparen Speicherplatz, wenn Zahlenwerte zwischengespeichert oder übertragen werden sollen.

Der Ablage-String entspricht in etwa einer Struktur in C. Es können verschiedene Datentypen in einer Variablen abgelegt und auf diese Weise zu einem Datensatz (Record) zusammengefaßt werden.

3

```
` Beispiele zu STOSS$
` A$ besteht aus 15 Punkten
A$ = STOSS$ ( ".....", 0, "Hello", 5 ) ` "Hello....."
A$ = STOSS$ ( ".....", 1, "Hello", 5 ) ` ".Hello....."
A$ = STOSS$ ( ".....", 2, "Hello", 5 ) ` "..Hello....."
A$ = STOSS$ ( ".....", 3, "Hello", 5 ) ` "...Hello....."
A$ = STOSS$ ( ".....", 15, "Hello", 5 ) ` ".....Hello"
A$ = STOSS$ ( ".....", 16, "Hello", 5 ) ` ".....Hello"
A$ = STOSS$ ( ".....", 17, "Hello", 5 ) ` ".....Hello"
A$ = STOSS$ ( ".....", 3, "Hello", -5 ) ` "...olleH....."
A$ = STOSS$ ( ".....", 2, "Hello", -5 ) ` "..olleH....."
A$ = STOSS$ ( ".....", 1, "Hello", -5 ) ` ".olleH....."
A$ = STOSS$ ( ".....", 0, "Hello", -5 ) ` "olleH....."
A$ = STOSS$ ( ".....", -1, "Hello", -5 ) ` ""
A$ = STOSS$ ( ".....", 0, "Hello", 5 ) ` "Hello....."
A$ = STOSS$ ( ".....", 1, "Hello", 4 ) ` ".Hell....."
A$ = STOSS$ ( ".....", 2, "Hello", 3 ) ` "..Hel....."
A$ = STOSS$ ( ".....", 3, "Hello", 2 ) ` "...He....."
A$ = STOSS$ ( ".....", 4, "Hello", 1 ) ` "....H....."
A$ = STOSS$ ( ".....", 4, "Hello", 0 ) ` "....."
```

Programmbeispiel:

```

'-----
' Name: STOSS.TIG
'-----

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
PRINT #1,STOSS$(".....",1,"hello",5) ' Ausgabe auf LC-Display
PRINT #1,STOSS$(".....",1,"hello",-5) ' Ausgabe auf LC-Display
PRINT #1,STOSS$(".....",5,"hello",3) ' Ausgabe auf LC-Display

A$ = ""                                ' A$ nicht initialisiert:
A$ = STOSS$(A$,3,"hello",5)            ' Laenge waere Zufall!
CALL SHOW_STRING_AS_HEX ( A$ )         ' String in A$ einbauen
                                        ' Bytes sind geschrieben
                                        ' die ersten 3 Bytes Zufall

A$ = "<1>I am a teststring"            ' neues A$
B$ = "bird"                            ' B$ mit 4 Bytes
A$ = STOSS$(A$,8,B$,12)                ' String B$ in A$ einbauen
                                        ' aber mehr Bytes als B$ hat
PRINT #1, A$                          ' 4 Bytes sind geschrieben

A$ = STOSS$(A$,100,B$,4)                ' String B$ in A$ einbauen
                                        ' aber jenseits der max.
                                        ' Laenge von A$
PRINT #1, len(A$)                     ' 4 Bytes sind geschrieben
END                                    ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----

SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2"           ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1         ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                         ' neue Zeile
END

```

3

Siehe auch: MID\$, NTOSS\$, NFROMS

STR\$

STR\$

STR\$ (Wert)

STR\$: Wandelt numerischen Wert um in String

Funktion: Wandelt den **Wert** in einen String um.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL.

Funktionswert ist vom Typ STRING und enthält die eingefügte Zahl.

STR\$ wandelt einen Zahlenwert in einen String um.

Programmbeispiel:

```
'-----  
' Name: STR$.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
    STRING A$                            ' STRING-Variable deklarieren  
    REAL R                                ' REAL-Variable deklarieren  
    LONG L                                ' LONG-Variable deklarieren  
' LCD-Treiber installieren (BASIC-Tiger)  
    INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
    R = -12345.6789                        ' R vorbes. (-1,23456789E004)  
    L = -54321                             ' L vorbesetzen  
    A$=LEFT$(STR$(R),6)+RIGHT$(STR$(L),5) ' String zusammensetzen  
    PRINT #1, "A$ = "; A$                 ' Ausgabe: A$ = -1,23454321  
END                                         ' Ende Task MAIN
```

Siehe auch: STRI\$, VAL_NUM

STRIS

STRIS (Wert, Formatstring)

STRIS: Wandelt numerischen Wert um in String, Format-String

Funktion: Wandelt den **Wert** in einen String um. Dabei wird der **Formatstring** wie bei der Ausgabe mit PRINT_USING zur Formatierung der Zahl verwendet.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG oder REAL.

Formatstring ist vom Typ STRING und enthält den Formatstring zur Formatierung des Wertes. Weitere Informationen zum Aufbau der Formatstrings finden Sie auf der Seiten 499ff.

Funktionswert ist vom Typ STRING und enthält die eingefügte Zahl.

STRIS wandelt einen Zahlenwert in einen String um. Dabei wird wie bei der Ausgabe mit PRINT_USING ein Formatstring zur Formatierung herangezogen. Ist der Formatstring leer oder falsch, wird der Standard-Formatstring verwendet.

Programmbeispiel:

```

' -----
' Name: STRIS.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$,B$,C$,D$                     ' STRING-Variablen deklarieren
  REAL R                                 ' REAL-Variable deklarieren
  LONG L                                 ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "NF<7><7>*<5>V0.0.0.0.0.1.3.3,5.0.0.0.0.0.0" '
  B$ = "EF<1><1>*<6>V0.0.0.0.0.0.0.1,6.0.0.0.0.0.0" '
  C$ = "SD<0><1> V3.3.3.3"               '
  D$ = "UH<9><9> kv0.0.0.0 9"             ' Formatstrings fuer STRIS
  R = -123456.7829                        ' Variablen besetzen
  L = -9876543                            '
  PRINT #1, "A$="; STRIS (R, A$)          ' Ausgabe auf LC-Display
  PRINT #1, "B$="; STRIS (R, B$)          '
  PRINT #1, "C$="; STRIS (L, C$)          '
  PRINT #1, "D$="; STRIS (L, D$)          '
END                                        ' Ende Task MAIN

```

Siehe auch: VAL_NUM

SUB

SUB

SUB Unterprogrammname [([*Parameterliste*])]

SUB...END: Unterprogramm definieren

Instruktion: Beginnt die Definition des Unterprogramms.

Parameterliste kann von 0 bis zu 16 Parameter umfassen und Konstanten, oder Variable enthalten.

3

Beachte: Die Parameterliste kann keine Ausdrücke enthalten.

Die Datentypen der übergebenen Parameter müssen mit den im Aufruf (siehe CALL) übergebenen Argumenten übereinstimmen. Unstimmigkeiten meldet der Compiler als Fehler.

Unterprogramme sind ein sehr leistungsfähiges Werkzeug in Tiger-BASIC®-Programmen. Die systematische Verwendung von Unterprogrammen trägt zu modularer, verständlicher Programmierung bei und reduziert sowohl die resultierende Programmlänge als auch den Debug-Aufwand zum Teil erheblich. Weder die Anzahl von Unterprogrammen, noch ihre Länge sind begrenzt, so lange ausreichend Programm-Speicherplatz (FLASH) zur Verfügung steht. Auch dürfen Unterprogramme grundsätzlich von jedem anderen Unterprogramm und jeder Task aus aufgerufen werden, ohne sich zu behindern. Auch rekursive Konstruktionen - Unterprogramm ruft sich selbst auf - sind erlaubt.

Unterprogramme können mit Variablen unterschiedlichen Geltungsbereichs arbeiten:

Lokale Variable werden im Unterprogramm deklariert. Ihr Geltungsbereich ist begrenzt auf genau den jeweiligen Unterprogramm-Aufruf. Wird ein Unterprogramm von mehreren Tasks gleichzeitig verwendet, so erzeugt jeder Unterprogramm-Aufruf einen eigenen Satz von lokalen Variablen, die nicht miteinander in Beziehung stehen. Keine solcher lokalen Variablen kann von einer anderen Task oder einem anderen Unterprogramm-Aufruf gelesen oder verändert werden. Lokale Variable haben also den Vorteil der „Geheimhaltung“ und „Sicherheit“ nach außen. Lokale Variable werden zur Laufzeit auf dem Stack angelegt (siehe Festlegung der Stack-Größe USER_STACK_SIZE, Seite 40). Sie werden wie alle Variablen im BASIC-Tiger® durch die Deklaration noch nicht initialisiert, besitzen also zunächst einen zufälligen Wert. Die Zahl und Größe von lokalen Variablen, sowie der Verschachtelungsgrad von Unterprogrammen erfordert eine bestimmte Mindestgröße des Stacks. Tritt zur Laufzeit ein Stack-Überlauf ein, so hat dies einschneidende Wirkung auf den Programm-Ablauf (Task wird gestoppt.). Bis

zur Fertigstellung eines Projektes ist die tatsächlich erforderliche Stack-Größe als Worst-Case Betrachtung festzulegen, damit sich in der Anwendung niemals ein Stack-Überlauf ereignen kann.

Will man mit dem erforderlichen Speicherplatz für den Stack haushalten, so führe man sich vor Augen, daß lokale Variablen in Unterprogrammen stets Stack belegen. Insbesondere lokale Strings und Arrays sollten dann möglichst sparsam eingesetzt werden. Strings sollten jeweils explizit auf die wirklich benötigte Größe hin deklariert werden. „STRING A\$ (10) belegt nur 12 Bytes auf dem Stack, während STRING B\$ stets mit der Default-Länge (z.B. 64+2 Bytes) deklariert wird. Bei ARRAYS kann durch geeignete Wahl des Typs, möglichst BYTE oder WORD statt LONG, der Platzbedarf reduziert werden.

Globale Variable können von jedem Unterprogramm und jeder Task aus angesprochen werden. Globale Variable eignen sich zur Kommunikation zwischen Tasks und als universelle Variable, Puffer, Arrays oder Strings mit weitreichender Bedeutung. Werden globale Variable von einem Unterprogramm verändert, ist sorgfältig auf die Plausibilität von Programm-Ablauf und Dateninhalten zu achten! Werden globale Variablen von verschiedenen Tasks aus manipuliert, kann z.B. der Task-Switch-Mechanismus kurzzeitig (!!) angehalten werden, bis die kritische Phase der Manipulation beendet ist (siehe: DI-SABLE_TSW / ENABLE_TSW).

Ein-/Ausgangs-Parameter

Bis zu 16 Werte und Variablen können als Parameter an ein Unterprogramm übergeben und von ihr zurückgegeben werden. Ein-/Ausgangs-Parameter werden durch Referenz weitergegeben, d.h. das Unterprogramm erhält Zeiger auf den Ort dieser Werte und Variablen. Daher belasten lange Strings, die als Parameter übergeben werden, den Stack nur mit wenigen Referenz-Bytes. Man beachte, daß diese Parameter nicht den Charakter von lokalen Variablen besitzen. Wird eine Ausgangsvariable in einem Unterprogramm verändert, so ändert sich sofort der Wert der entsprechenden Original-Variablen. Reine Eingangs-Variablen sind vor Veränderungen in der Subroutine geschützt.

Bei einem Programm mit intensiver Unterprogrammstruktur, vielen lokalen Variablen, Ein/Ausgabeparametern, rekursiven Schritten kann es vorkommen, daß der Stack überläuft. Die Folge ist dann ein Runtime-Error, welcher zum Beenden der betroffenen Task führt. Die Task MAIN wird jedoch neu gestartet. (Siehe auch 'Fehlerbehandlung während der Laufzeit auf Seite 50 und Compiler-Anweisung USER_STACK_SIZE auf Seite 40).

SUB

Anhaltspunkte für die Stackbelegung (ca. Werte):

	Stackbedarf
Unterprogramm-Aufruf:	17 Bytes
BYTE (lokal)	1 Byte
WORD (lokal)	2 Bytes
LONG (lokal)	4 Bytes
REAL (lokal)	8 Bytes
STRING (lokal)	Länge + 2
ARRAY (lokal)	Länge=Anzahl der Elemente * Elementlänge
Ein-Ausgangs-Parameter:	6 Bytes
Globale Variablen:	0 Bytes

3

Bei einer Stackgröße von 512 Bytes können also z.B. 8 Unterprogramm-Level mit bis zu 64 Byte Stack-Bedarf aktiv sein. Die Stackgröße läßt sich, soweit Speicherplatz vorhanden ist, mit der Compiler-Anweisung `USER_STACK_SIZE` festlegen. Der Defaultwert ist 2048 Bytes.

Einige Beispiele für die Parameterangabe Unterprogrammen:

SUB UPRG ()

Ein Unterprogramm ohne Parameterübergabe.

SUB UPRG (LONG N)

Eine numerische Variable N wird übergeben. Die Variable kann im Unterprogramm nicht verändert werden. N kann ein Element eines Arrays sein.

SUB UPRG (STRING A\$)

Eine Stringvariable wird übergeben. Der String kann im Unterprogramm nicht verändert werden. Der String kann ein Element eines Arrays sein.

SUB UPRG (LONG N1, N2, N3; VAR STRING S1\$, S2\$)

N1, N2 und N3 können im Unterprogramm 'UPRG' nicht verändert werden.

S1\$ und S2\$ werden im aufrufenden Programm verändert.

Variable, die innerhalb eines Unterprogramms deklariert werden, sind stets lokal. Bei jedem Aufruf dieser Unterprogramms, wird ein neuer Satz lokaler Variablen erzeugt. Dadurch ist gewährleistet, daß es auch bei gleichzeitiger Verwendung aus verschie-

denen Tasks heraus oder bei rekursivem Aufruf nicht zur Konflikten zwischen den Variablen kommt.

Wird eine Variable benutzt, ohne vorher deklariert worden zu sein, wird sie automatisch als Typ LONG angenommen, wenn es eine numerische Variable ist, und automatisch vom Typ STRING, wenn es eine Stringvariable ist.

Die Größe des Unterprogramms ist nur durch den vorhandenen Speicher begrenzt.

Die Definition des Unterprogramms endet mit **END**. Die END-Instruktion hat gleichzeitig die Wirkung einer RETURN-Instruktion.

Programmbeispiel:

```

'-----
' Name: SUB.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG I,J                               ' LONG-Variablen deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  I = 5                                  ' I vorbesetzen
  CALL FAC (I,J)                         ' Aufruf der Subroutine FAC
  PRINT #1, "FAC (; I; ") ="; J          ' Ausgabe von J
END                                       ' Ende Task MAIN

'-----
' Subroutine FAC berechnet Fakultaeet von I und uebergibt Ergebnis in J
'-----
SUB FAC (LONG K; VAR LONG L)             ' Beginn Subroutine FAC
  BYTE B                                 ' BYTE-Variable deklarieren
  L = 1                                  ' L initialisieren
  FOR B = 1 TO K                          ' Zaehlschleife fuer B (1..K)
    L = L * B                             ' Mit B multiplizieren
  NEXT                                    ' naechster Wert fuer B
END                                       ' Ende Subroutine FAC

```

SWITCH...CASE

SWITCH...CASE

SWITCH *Auswahlvariable*

CASE *Auswahlwert_1:*

Anweisungsblock

CASE *Auswahlwert_n:*

Anweisungsblock

[**DEFAULT:**

Anweisungsblock]

ENDSWITCH

SWITCH..CASE..ENDCASE: Bedingte Programmausführung, Standard

Instruktion: Führt eine Anweisung oder einen Anweisungsblock in Abhängigkeit von dem Wert der angegebenen **Auswahlvariablen** aus.

Auswahlvariable ist eine Variable vom Typ BYTE, WORD, LONG oder STRING und gibt die auszuwertende Variable an.

Auswahlwert_x ist eine Konstante vom Typ BYTE, WORD, LONG, wenn die Auswahlvariable numerischen Typs ist, oder vom Typ STRING, wenn die Auswahlvariable vom Typ STRING ist. Hinter dem Auswahlwert steht ein Doppelpunkt.

Hinter jeder Zeile mit der CASE-Instruktion folgt ein Anweisungsblock, der eine oder mehrere Anweisungen enthält. Der zu der CASE-Instruktion zugehörige Anweisungsblock wird dann ausgeführt, wenn die Auswahlvariable den Wert enthält, der als Parameter der CASE-Instruktion angegeben ist. Nach der Ausführung des Anweisungsblocks wird die auf 'ENDSWITCH' folgende Instruktion ausgeführt. Weder die Anzahl der CASE-Instruktionen noch die Größe der Anweisungsblöcke sind begrenzt.

Trifft der Wert keiner CASE-Instruktion zu, so wird der Anweisungsblock ausgeführt, der zur Instruktion **DEFAULT:** gehört, sofern diese angegeben ist. Ist **DEFAULT:** nicht angegeben, fährt das Programm sofort hinter **ENDSWITCH** fort. **DEFAULT:** ist nur einmal am Ende der CASE-Kette angegeben.

Im Vergleich zwischen einer Konstruktion mit **IF...ELSE...ENDIF** und **SWITCH...CASE...ENDSWITCH** ist **SWITCH...CASE** wesentlich schneller, da hier die Verzweigung abhängig von einer Konstanten erfolgt, wo bei **IF...ELSE** verknüpfte Ausdrücke als Bedingung möglich sind. Anweisungsblöcke weiter oben in der CASE-Kette werden eher ausgeführt als die Anweisungsblöcke am Ende der Kette.

SWITCH...CASE

Einen weiteren Geschwindigkeitsvorteil bietet die indizierte Verzweigung über eine Tabelle mit der Instruktion SWITCH...CASE.

Beispiele:

```
SWITCH N
  CASE 10:
    PRINT #1, "zehn"
  CASE 100:
    PRINT #1, "hundert"
  CASE 1000:
    PRINT #1, "tausend"
  DEFAULT
    PRINT #1, "weder zehn noch hundert noch tausend"
ENDSWITCH
```

3

```
SWITCH YN$
  CASE "y":
    PRINT #1, "yes"
  CASE "n":
    PRINT #1, "no"
  DEFAULT
    PRINT #1, "please enter y for yes or n for no"
ENDSWITCH
```

SWITCH...CASE

Programmbeispiel:

3

```

'-----
' Name: SWITCH_CASE.TIG
'-----
' Die Zahlen 16 und 4 werden addiert, subtrahiert oder multipliziert
' oder dividiert. Auswahl mit den Tasten <Up> und <Down>, ausfuehren
' mit <Return>. Die Tasten-Abfrage wird mit SWITCH durchgefuehrt,
' wobei ein "echter" Vergleich stattfindet. Die Auswahl der Berechnung
' wird mit SWITCHI durchgefuehrt, wobei der Index einer Sprungtabelle
' die naechste auszufuehrende Anweisung angibt.
'-----
#INCLUDE KEYB_PP.INC                                ' dt. Tastatur Plug & play-Lab

TASK MAIN                                            ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 80h, 8
CALL INIT_KEYB (1)                                ' Tastatur deutsch
STRING Choice$,Key$                                ' STRING-Variable deklarieren
LONG L1, L2, M, N, GOON                            ' LONG-Variablen deklarieren
Choice$ = ":+-:*/:::"                              ' Choice$ mit Auswahl-Elementen
L1 = 16                                              ' L1 vorbesetzen
L2 = 4                                              ' L2 vorbesetzen
M = 0                                              ' Menue-Index = 0
PRINT #1, "<1>  Keys: Up,Down,Ret"                ' Ausgabe auf LC-Display
LOOP 999999999                                     ' sehr viele Loops
GOON = 1
WHILE GOON = 1
    PRINT #1, "<2><10>"; L1; " ";                    ' aktuelle Wahl darstellen
    PRINT #1, SELECT$ (M,Choice$); L2;
    FOR N=0 TO 0 STEP 0
        RELEASE_TASK
        GET #1, #0, #1, 1, N
    NEXT
    GET #1, 1, Key$
    SWITCH Key$
        CASE _CR:
            GOON = 0
        CASE _UP:
            M = (M - 1) BITAND 3
        CASE _DO:
            M = (M + 1) BITAND 3
    END SWITCH
END WHILE
SWITCHI M
CASE 0:
    PRINT #1, " ="; L1 + L2; " "
CASE 1:
    PRINT #1, " ="; L1 - L2; " "
CASE 2:
    PRINT #1, " ="; L1 * L2; " "
CASE 3:
    PRINT #1, " ="; L1 / L2; " "
END SWITCH
END LOOP
END
'-----
' Ende Task MAIN

```

Instruktion	Vorteil
IF...THEN...ELSE	Ausdrücke und verknüpfte Bedingungen möglich
SWITCH...CASE	beliebige Konstante hinter CASE, schneller als IF...THEN...ELSE
SWITCHI...CASE	höchste Verzweigungsgeschwindigkeit, da über indizierte Sprungtabelle

Siehe auch: SWITCHI...CASE, IF...THEN

SWITCHI...CASE

SWITCHI...CASE

SWITCHI *Auswahlvariable*

CASE *Auswahlwert_1:*

Anweisungsblock

CASE *Auswahlwert_n:*

Anweisungsblock

[**DEFAULT:**

Anweisungsblock]

ENDSWITCH

SWITCHI..CASE..ENDCASE: Bedingte Programmausführung, high-speed

Instruktion: Führt eine Anweisung oder einen Anweisungsblock in Abhängigkeit von dem Wert der angegebenen **Auswahlvariablen** aus.

Auswahlvariable ist eine Variable vom Typ BYTE, WORD, LONG und gibt die auszuwertende Variable an. Der Wertebereich dieser Variablen ist 0...65535.

Auswahlwert_x ist eine Konstante vom Typ BYTE, WORD, LONG. Hinter dem Auswahlwert steht ein Doppelpunkt. Auch hier gelten Werte von 0...65535.

SWITCHI arbeitet mit einem Index in eine Sprungtabelle und ist daher sehr schnell. Hinter jeder Zeile mit 'CASE' folgt ein Anweisungsblock, der eine oder mehrere Anweisungen enthält. Der zu der CASE-Instruktion zugehörige Anweisungsblock wird dann ausgeführt, wenn die Auswahlvariable den Wert enthält, der als Parameter der CASE-Instruktion angegeben ist. Nach der Ausführung des Anweisungsblocks wird die auf 'ENDSWITCH' folgende Instruktion ausgeführt.

Trifft der Wert keiner CASE-Instruktion zu, so wird der Anweisungsblock ausgeführt, der zur Instruktion DEFAULT: gehört, sofern diese angegeben ist. Ist DEFAULT: nicht angegeben, fährt das Programm sofort hinter ENDSWITCH fort. DEFAULT: ist nur einmal am Ende der CASE-Kette angegeben.

Im Unterschied zu SWITCH...CASE...ENDSWITCH verzweigt diese Instruktion nicht über eine Verzweigungskette, sondern über eine Tabelle. Die Tabelle wird abhängig von dem größten und kleinsten Wert gebildet, der in der CASE-Kette vorkommt. Haben Sie z.B. nur 10 Verzweigungen mit möglichen Werten zwischen 10 und 10.000, dann wird eine sehr große Tabelle aufgebaut, die sehr viel Speicher beansprucht, jedoch nur 10 Werte enthält. Liegen die Werte jedoch zwischen 10.000

und 10.010, so läßt sich der Geschwindigkeitsvorteil ohne Speicherplatzeinbußen voll ausnutzen.

Im Vergleich zwischen einer Konstruktion mit IF...ELSE...ENDIF und SWITCH...CASE...ENDSWITCH ist SWITCH...CASE wesentlich schneller, da hier die Verzweigung abhängig von einer Konstanten erfolgt, wo bei IF...ELSE verknüpfte Ausdrücke als Bedingung möglich sind. Die schnellste Konstruktion ist die indizierte Verzweigung über eine Tabelle mit der Instruktion SWITCHI...CASE.

Beispiel für eine CASE-Kette mit Verzweigungen für die 5 Werte für N 38, 47, 48, 63, 69. Eine Sprungtabelle wird für den Bereich von dem kleinsten bis zum größten Wert gebildet, also 32 Sprungziele, von denen jedoch nur 5 Werte benutzt werden:

SWITCHI N

38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61
62	63	64	65	66	67	68	69

Anweisungen für den Fall N=38

Anweisungen für den Fall N=47

Anweisungen für den Fall N=48

Anweisungen für den Fall N=63

Anweisungen für den Fall N=69

SWITCHI...CASE

Programmbeispiel:

3

```

'-----
' Name: SWITCHI_CASE.TIG
'-----
' Eine Schleife, die eine Variable von 1 bis 10 hochzaehlt, wird 200
' mal durchlaufen. Die Variable legt fest, welcher Ausweisungsblock
' ausgefuehrt werden soll. Dazu werden erst IF..THEN-Verschachtelungen
' verwendet, dann SWITCH bzw. SWITCHI-Instruktionen. Die Dauer der
' Abarbeitung wird jeweils angezeigt. i der Index einer Sprungtabelle
'-----
TASK MAIN                                ' Beginn Task MAIN
    LONG n, t1                          ' LONG-Variablen deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
'-----
' hier findet die Abfrage ueber IF..THEN-Verschachtelung statt
'-----
t1=TICKS()                               ' speichere Anfangszeit
LOOP 200                                 ' 200 Schleifendurchlaeufer
    FOR n=1 TO 10                        ' Zaehle n von 1 bis 10
        IF n=1 THEN                     ' Falls n=1 mache nichts
            ELSE                         ' sonst:
                IF n=2 THEN             ' Falls n=2 mache nichts
                    ELSE                 ' sonst:
                        IF n=3 THEN      ' Falls n=3 mache nichts
                            ELSE         ' sonst:
                                IF n=4 THEN ' Falls n=4 mache nichts
                                    ELSE   ' sonst:
                                        IF n=5 THEN ' Falls n=5 mache nichts
                                            ELSE   ' sonst:
                                                IF n=6 THEN ' Falls n=6 mache nichts
                                                    ELSE   ' sonst:
                                                        IF n=7 THEN ' Falls n=7 mache nichts
                                                            ELSE   ' sonst:
                                                                IF n=8 THEN ' Falls n=8 mache nichts
                                                                    ELSE   ' sonst:
                                                                        IF n=9 THEN ' Falls n=9 mache nichts
                                                                            ELSE   ' sonst:
                                                                                IF n=10 THEN ' Falls n=10 mache nichts
                                                                                    ENDIF
                                                                                ENDIF
                                                                            ENDIF
                                                                        ENDIF
                                                                    ENDIF
                                                                ENDIF
                                                            ENDIF
                                                        ENDIF
                                                    ENDIF
                                                ENDIF
                                            ENDIF
                                        ENDIF
                                    ENDIF
                                ENDIF
                            ENDIF
                        ENDIF
                    ENDIF
                ENDIF
            ELSE
                SWITCHI n, t1
            ENDIF
        ENDIF
    NEXT n                               ' naechster Wert for n
ENDLOOP                                 ' Ende der Schleife
PRINT #1, "IF..THEN: ", TICKS()-t1      ' Ausgabe der Dauer
'-----
' hier findet die Abfrage ueber die SWITCH..CASE-Anweisung statt
'-----
t1=TICKS()                               ' speichere Anfangszeit
LOOP 200                                 ' 200 Schleifendurchlaeufer

```

```

FOR n=1 TO 10                                ' Zaehle n von 1 bis 10
  SWITCH n                                    ' Bedingte Auswahl nach Inhalt
    CASE 1:                                  ' n=1: mache nichts
    CASE 2:                                  ' n=2: mache nichts
    CASE 3:                                  ' n=3: mache nichts
    CASE 4:                                  ' n=4: mache nichts
    CASE 5:                                  ' n=5: mache nichts
    CASE 6:                                  ' n=6: mache nichts
    CASE 7:                                  ' n=7: mache nichts
    CASE 8:                                  ' n=8: mache nichts
    CASE 9:                                  ' n=9: mache nichts
    CASE 10:                                 ' n=10: mache nichts
  ENDSWITCH                                   ' Ende der bedingten Auswahl
NEXT                                           ' naechster Wert for n
ENDLOOP                                       ' Ende der Schleife
PRINT #1, "SWITCH :"; TICKS()-t1             ' Ausgabe der Dauer
-----
' hier findet die Abfrage ueber die SWITCHI..CASE-Anweisung statt
-----
t1=TICKS()                                    ' speichere Anfangszeit
LOOP 200                                      ' 200 Schleifendurchlaeufer
  FOR n=1 TO 10                              ' Zaehle n von 1 bis 10
    SWITCHI n                                ' Bedingte Auswahl nach Index
      CASE 1:                                ' n=1: mache nichts
      CASE 2:                                ' n=2: mache nichts
      CASE 3:                                ' n=3: mache nichts
      CASE 4:                                ' n=4: mache nichts
      CASE 5:                                ' n=5: mache nichts
      CASE 6:                                ' n=6: mache nichts
      CASE 7:                                ' n=7: mache nichts
      CASE 8:                                ' n=8: mache nichts
      CASE 9:                                ' n=9: mache nichts
      CASE 10:                               ' n=10: mache nichts
    ENDSWITCH                                 ' Ende der bedingten Auswahl
  NEXT                                       ' naechster Wert for n
ENDLOOP                                     ' Ende der Schleife
PRINT #1, "SWITCHI :"; TICKS()-t1           ' Ausgabe der Dauer
END                                          ' Ende Task MAIN

```

Instruktion	Vorteil
IF...THEN...ELSE	Ausdrücke und verknüpfte Bedingungen möglich
SWITCH...CASE	beliebige Konstante hinter CASE, schneller als IF...THEN...ELSE
SWITCHI...CASE	höchste Verzweigungsgeschwindigkeit, da über indizierte Sprungtabelle

Siehe auch: SWITCH...CASE, IF...THEN

SYSVAR\$

SYSVAR\$

SYSVAR\$ (FunktionsNr, Parameter2)

SYSVAR\$: Liest String-Systemvariable

Funktion: Liefert den Wert einer STRING-Systemvariablen zurück.

FunktionsNr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG und ist die Nummer der Abfrage.

Parameter2 kann verschiedene Bedeutungen haben oder ist manchmal irgendeine Zahl (Dummy).

Funktionswert ist vom Typ STRING.

Systemvariable des Typs STRING werden mit dieser Funktion abgefragt.

Benutzen Sie die Namen, dann brauchen Sie bei Änderungen im Nummernsystem Ihre Programme nicht anzupassen. 'THIS_TASK' wird für die aktuelle Task eingesetzt.

Tip

Binden Sie die Datei 'DEFINE_A.INC' ein, um die Funktionsnamen zu verwenden, da sich die Funktionsnummern im Rahmen der Weiterentwicklung von Tiger-BASIC® ändern können. Funktionen von SYSVAR\$:

Funktionsname	Nr	2. Parameter	Beschreibung
RE_T_STRI	<00>	Taskname	Fehlermeldungstext des letzten Laufzeitfehlers in Task mit der Nummer TaskNr
RE_C_STRI	<01>	ErrCode	Fehlermeldungstext zu dem Fehler mit der Nr. ErrCode

Programmbeispiel:

```

'-----
' Name: SYSVAR$.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER (32) OF BYTE                ' FIFO-Buffer einrichten
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  ON ERRTASK CALL Error                  ' Bei Fehler aufrufen
  FOR I = 0 TO 35                        ' 35 Durchlaufe
    PRINT #1, "<1>I ="; I                ' I auf LC-Display ausgeben
    PUT FIFO BUFFER, I                  ' Fehler herbeifuehren (>31)
    WAIT DURATION 500                   ' 500 ms warten
  NEXT                                  ' Ende FOR...NEXT
END                                      ' Ende Task MAIN

'-----
' Fehlerbehandlungsroutine
'-----
SUB Error                                ' Beginn Subroutine Error
  PRINT #1, "Error!"                    ' Ausgabe auf LC-Display
  PRINT #1, SYSVAR$ (0,0)               ' Ausgabe Fehlermeldungstext
  WAIT DURATION 2000                    ' 2 Sek warten
  ON_ERROR_RESET                        ' Fehler Flag Reset
END                                      ' Ende Subroutine Error

```

SYSVARN

SYSVARN

SYSVARN (FunktionsNr, Parameter2)

SYSVARN: Liest numerische Systemvariable

Funktion: Liefert den Wert einer LONG-Systemvariablen zurück. Systemvariable des numerischen Typs (außer REAL) werden mit dieser Funktion abgefragt. Die Abfrage kann auch Systemfunktionen auslösen.

FunktionsNr ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD, LONG und ist die Nummer der Abfrage.

Parameter2 kann verschiedene Bedeutungen haben oder ist manchmal irgendeine Zahl (Dummy).

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Den Funktionsnummern sind in der Include-Datei DEFINE_A.INC Namen zugeordnet, die Sie in untenstehender Tabelle finden.

Benutzen Sie die Namen, dann brauchen Sie bei Änderungen im Nummernsystem Ihre Programme nicht anzupassen. 'THIS_TASK' wird für die aktuelle Task eingesetzt.

Tip

Binden Sie die Datei 'DEFINE_A.INC' ein, um die Funktionsnamen zu verwenden, da sich die Funktionsnummern im Rahmen der Weiterentwicklung von Tiger-BASIC® ändern können. Funktionen von SYSVARN:

Funktionsname	Nr	2. Parameter	Beschreibung
RE_ID	<00>	Taskname	Letzter Run-Time-Error-Code in der angegebenen Task
RE_CNT	<01>	Taskname	Run-Time-Error-Zählerstand
RE_SEMA	<02>	Taskname	Akt. Error-Flag (0=ok, 1=error)
RE_IDRES	<15>	Taskname	setzt Error-Flag zurück und liefert Error-Code zurück
CALL_LEVEL	<16>	Taskname	Stack-Verschachtelungstiefe
DSTACK_FILL	<17>	Taskname	Stack-Füllung in Bytes
DSTACK_FREE	<18>	Taskname	freier Stack-Platz in Bytes
DSTACK_SIZE	<19>	Dummy	Stack-Größe in Bytes
DRAM	<30>	0	Gesamt DRAM im System

Funktionsname	Nr	2. Parameter	Beschreibung
SRAM	<31>	0	Gesamt SRAM im System
FLASH_CHIPS	<32>	Dummy	Anzahl der Flash-Chips
FLASH_SIZE	<33>	Dummy	Größe der Flash-Chips in Bytes
FLASH_SEC	<34>	Dummy	Anzahl Sektoren pro Chip
FLASH_SSIZE	<35>	Dummy	Flash-Sektorgröße
FLASH_ASEC	<36>	Dummy	Anzahl der Flash-Sektoren
FLASH_GSIZE	<37>	Dummy	Größe des Flashspeichers in Bytes
FLASH_DSEC	<38>	Dummy	Anzahl der Flash-Sektoren für User-Daten
FLASH_DSIZE	<39>	Dummy	Größe des Flashspeichers in Bytes für User-Daten
FLASH_DMODE	<40>	Dummy	0=während Flash-Operationen wartet das System 1=während Flash-Operationen läuft das System weiter
ACT_TASK	<48>	Dummy	aktuelle Task-Nummer
TASK_PRIO	<49>	Taskname	aktuelle Task-Priorität dieser Task
SUM_PRIO	<50>	Dummy	Summe aller Task-Prioritäten
NR_OF_TASKS	<51>	Dummy	Anzahl Tasks im Programm
NR_ACT_TASKS	<52>	Dummy	Anzahl aktiver Tasks
TIGER_MODE	<67>	Dummy	Tiger-Modus: 0: RUN-Mode 1: PC-Mode =Debug-MODE
TIGER_VERS	<68>	Dummy	Tiger-Version
TIGER_MODULE	<69>	Dummy	Tiger-Modul Typ: 0 = Modul-Familie A

SYSVARN

Programmbeispiel:

3

```
'-----
' Name: SYSVARN.TIG
'-----
#INCLUDE DEFINE_A.INC                                ' globale Def. einbinden

TASK MAIN                                             ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
CALL TEST                                           ' Subroutine TEST aufrufen
PRINT #1, "<1>Stack-Level =";                       ' Ausgabe auf LC-Display
PRINT #1, SYSVARN (CALL_LEVEL, -1)                 ' Stack-Versch.-tiefe abfragen
PRINT #1, "Stack-free =";                           ' Ausgabe auf LC-Display
PRINT #1, SYSVARN (DSTACK_FREE, -1)                 ' freien Stack-Platz abfragen
END                                                  ' Ende Task MAIN

'-----
' Subroutine fragt Stack ab (rekursiv bis Verschachtelungstiefe 10)
'-----
SUB TEST                                             ' Beginn Subroutine TEST
PRINT #1, "<1>Stack-Level =";                       ' Ausgabe auf LC-Display
PRINT #1, SYSVARN (CALL_LEVEL, -1)                 ' Stack-Versch.-tiefe abfragen
PRINT #1, "Stack-free =";                           ' Ausgabe auf LC-Display
PRINT #1, SYSVARN (DSTACK_FREE, -1)                 ' freien Stack-Platz abfragen
WAIT DURATION 2000                                  ' 2 Sek warten
IF SYSVARN (CALL_LEVEL, -1) < 10 THEN               ' Falls Versch.-Tiefe < 10,
CALL TEST                                           ' Sub TEST rekursiv aufrufen
ENDIF
END                                                  ' Ende Subroutine TEST
```


TABS

TABS [*Taskname*] *String*

TABS: Setzt Tabulatorpositionen für PRINT_USING

Instruktion: Legt Tabulatorpositionen für die Ausgabe mit PRINT_USING in der Task **Taskname** fest.

Taskname gibt an, für welche Task die Tabulatorpositionen gesetzt werden sollen. Jede Task kann eigene TABS-Festlegungen treffen, ohne daß es eine Beeinflussung gibt oder zur Laufzeit ständig zwischen verschiedenen Einstellungen gewechselt werden muß. Wird Taskname nicht angegeben, wird die aktuelle Task angenommen.

String besteht aus den Angaben der Tabulatorpositionen in binärer Form gefolgt von der Angabe des Tabulatortyps in Buchstaben. Die erste Position ist Position 0:

Buchstabe	Tabulatortyp
L	linksbündig
R	rechtsbündig
C	zentriert (center)

Beispiel:

TABS "<1>L<0AH>C<19>R"

Setzt einen linksbündigen Tabulator an Position 1 (Position 0 wäre die erste Position), einen zentrierten Tabulator an Position 10 (0A Hex) und eine rechtsbündigen Tabulator an Position 19 (z.B. die rechteste Position auf dem 4x20-LC-Display). In der Instruktion PRINT_USING wird die erste Tabulatorposition durch ein Komma erreicht.

PRINT_USING #1,, "Hallo Welt"

mit zwei Kommata hinter '#1'. Druckt den Text 'Hallo Welt' an der ersten Tabulatorposition. Würde das zweite Komma fehlen, würde 'Hallo Welt' an den Anfang der Zeile gedruckt.

TABS

Programmbeispiel:

```
'-----
' Name: TABS.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  TABS MAIN, "<05>L"                      ' linksbueendig ab Pos.2
  PRINT USING #1, "A", "Hello World"      ' -> "A...Hello World..."
  TABS MAIN, "<19>R"                      ' rechtsbueendig an Pos.19
  PRINT USING #1, "B", "Hello World"      ' -> "B.....Hello World"
END                                         ' Ende Task MAIN
```

Siehe auch: USING, FRAME

Ein weiteres Beispiel zeigt, wie die mit TABS gesetzten Tabulatoren genutzt werden. Das Ergebnis im Ausdruck finden Sie auf den darauffolgenden Seiten.

Programmbeispiel:

```

-----
' Name: TABS1.TIG
-----
#INCLUDE DEFINE_A.INC          ' allg. Symbol-Definitionen

TASK MAIN                      ' Start TASK Main
  STRING A$,F$,T1$,T2$         ' String-Variablen deklarieren
  INSTALL DEVICE #2, "SER1B_K1.TDD",& ' ser. Treiber installieren
  BD_19_200, DP_8N, JA,&        ' Einstellungen Kanal 0
  BD_38_400, DP_8E, JA        ' Einstellungen Kanal 1

  F$ = "|          |          |          |"
  T1$ = "<10>R<30>R<50>R"
  T2$ = "<<10>R<<30>R<<50>R"
  PRINT #2, #0, "String for FRAME:"
  PRINT #2, #0, CHR$(34);F$;CHR$(34)
  PRINT #2, #0, "String for TABS:"
  PRINT #2, #0, CHR$(34);T2$;CHR$(34)
  PRINT #2, #0, ""
  PRINT #2, #0, "<<----0---><<----1---><<----2---><<----3--->";
  PRINT #2, #0, "<<----4---><<----5--->"
  PRINT #2, #0, ".123456789.123456789.123456789.123456789";
  PRINT #2, #0, ".123456789.123456789"
  PRINT #2, #0, "          R          R          ";
  PRINT #2, #0, "          R          "          ";
  TABS T1$
  FRAME F$
  A$ = ""
  FOR X = 1 TO 15
    A$ = A$ + "A"
    PRINT USING #2, #0,,A$,A$,A$
  NEXT
  PRINT #2, #0, "<12>";

  F$ = "|          |          |          |"
  T1$ = "<10>L<30>L<50>L"
  T2$ = "<<10>L<<30>L<<50>L"
  PRINT #2, #0, "String for FRAME:"
  PRINT #2, #0, CHR$(34);F$;CHR$(34)
  PRINT #2, #0, "String for TABS:"
  PRINT #2, #0, CHR$(34);T2$;CHR$(34)
  PRINT #2, #0, ""
  PRINT #2, #0, "<<----0---><<----1---><<----2---><<----3--->";
  PRINT #2, #0, "<<----4---><<----5--->"
  PRINT #2, #0, ".123456789.123456789.123456789.123456789";
  PRINT #2, #0, ".123456789.123456789"
  PRINT #2, #0, "          L          L          ";
  PRINT #2, #0, "          L          "          ";
  TABS T1$
  FRAME F$
  A$ = ""
  FOR X = 1 TO 15
    A$ = A$ + "A"
    PRINT USING #2, #0,,A$,A$,A$

```

TABS

3

```
NEXT
PRINT #2, #0, "<12>";

F$ = "|
T1$ = "<10>C<30>C<50>C"
T2$ = "<<10>C<<30>C<<50>C"
PRINT #2, #0, "String for FRAME:"
PRINT #2, #0, CHR$(34);F$;CHR$(34)
PRINT #2, #0, "String for TABS:"
PRINT #2, #0, CHR$(34);T2$;CHR$(34)
PRINT #2, #0, ""
PRINT #2, #0, "<<----0---><----1---><----2---><----3--->";
PRINT #2, #0, "<<----4---><----5--->"
PRINT #2, #0, ".123456789.123456789.123456789.123456789";
PRINT #2, #0, ".123456789.123456789"
PRINT #2, #0, "          C          C          ";
PRINT #2, #0, "          C          "
TABS T1$
FRAME F$
A$ = ""
FOR X = 1 TO 15
  A$ = A$ + "A"
  PRINT USING #2, #0,,A$,A$,A$
NEXT
PRINT #2, #0, "<12>";

END                                     ' Ende TASK Main
```

467

TAN

TAN

TAN (rad)

TAN: TANGENS, Real-Werte

Funktion: Liefert den Tangens von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $-4.5 \cdot 10^{15}$ bis $4.5 \cdot 10^{15}$ und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von $4.19 \cdot 10^{-307}$ bis $1.67 \cdot 10^{308}$. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```
'-----
' Name: TAN.TIG
'-----
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                             ' Beginn Task MAIN
  LONG DEG                            ' LONG-Variable deklarieren
  REAL RAD                            ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
FOR DEG = 0 TO 360                    ' Winkel von 0 bis 360 Grad
  RAD = DEG * PI_180                  ' Winkel: Grad- zu Bogenmass
  PRINT #1, "<1>TAN ("; DEG; " Grad)" ' Ausgabe auf LC-Display
  PRINT USING #1, "="; TAN (RAD)     ' Tangenswert ausgeben
  WAIT DURATION 1000                 ' 1 Sek warten
NEXT                                  ' naechster Winkelwert
END                                   ' Ende Task MAIN
```

Siehe auch: SIN, COS, ASIN, ACOS, ATAN, SINH

TAN – Trans-Aktions-Nummern

Eine Trans-Aktions-Nummer (TAN) kommt zur Anwendung, wenn sichergestellt sein soll, daß nur eine bestimmte Personen, die im Besitz einer gültigen Trans-Aktions-Nummer ist, einen Vorgang durchführen kann. Bekannt sind diese Nummern vor allem aus dem Online-Banking. Jede Nummer ist nur einmal verwendbar. Der Berechtigte braucht also einen Vorrat an TANs, die in einer Liste stehen. Im System wird bei der Einleitung des Vorgangs nach der Nummer gefragt und die eingegebene Nummer mit der im System vorhandenen Nummernliste verglichen. Wenn die Nummer vorhanden ist, dann wird der Vorgang durchgeführt und die Nummer als ‚verbraucht‘ markiert. Sind alle Nummern der Liste verbraucht, dann muß eine neue Liste besorgt werden. Verliert die Person die Berechtigung, so sollten unverbrauchte TANs gelöscht werden.

Auch in einer Tiger-BASIC® - Anwendung können solche Nummern eingesetzt werden. Es gibt nun eine Reihe von Funktionen, die TANs unterstützen:

- TAN_CHK prüft TAN und markiert sie als verbraucht.
- TAN_COUNT zählt unverbrauchte oder verbrauchte, gelöschte TANs
- TAN_DEL löscht TANs
- TAN_FIRST sucht erste unverbrauchte oder verbrauchte, gelöschte TAN

In Tiger-BASIC® wird im FLASH eine Tabelle angelegt, in der die TANs enthalten sind. Die Funktion TAN_CHK prüft, ob eine bestimmte TAN dort vorkommt oder nicht. Ein gefundener TAN-Eintrag wird sofort als verbraucht markiert. Wenn eine TAN bereits als verbraucht markiert wurde oder gelöscht wurde oder nicht in der Liste vorkommt, dann wird eine Fehlernummer zurückgegeben.

Um TAN-Listen effektiv zu nutzen, d.h. mit möglichst hoher Sicherheit, denn dafür sind TANs ja da, und mit möglichst günstiger Flash-Ausnutzung, sollten die folgenden Regeln bedacht werden:

- Eine lange TAN ist sicherer als eine kurze.
- Die kürzeste Codierung ist die binäre (spart Flash-Speicher).
- Den ganzen Codebereich ausnutzen, also binär und nicht ASCII.
- Lange Listen sollten unterteilt werden. Wenn eine kleine Auswahl aus einer langen Liste zur Zeit gültig ist, dann erhöht das die Sicherheit.
- Sicherheitsmechanismen in die Anwendung einbauen, z.B. nach drei falschen Nummern ist die Liste gesperrt.

Eine TAN-Liste darf beliebig viele Einträge besitzen. Durch die Wahl der Startadresse und Anzahl können die TAN-Funktionen gezielt Teil-Listen verwalten.

Eine TAN-Liste besteht aus Einträgen von TANs. Jeder Eintrag besteht aus den Bytes der TAN-Nummer (4...12 bytes) und dem nachfolgenden Flag-Byte des Eintrags. In dem Flag-Byte wird vermerkt, ob eine TAN unverbraucht, verbraucht oder

TAN – Trans-Aktions-Nummern

gelöscht ist. Es hat eine vorgegebene Struktur mit festgelegten Bits 2...7. Bei einer unverbrauchten TAN hat dieses Flag-Byte den Wert 10011011b.

Aufbau eines TAN-Flag-Bytes:

7	6	5	4	3	2	1	0	Bitnummer
1	0	0	1	1	0			Bedeutung der Bits 0 und 1
						1	1	TAN noch nicht verbraucht
						1	0	TAN verbraucht
						0	1	TAN gelöscht
						0	0	TAN verbraucht und gelöscht

verbraucht

bedeutet, die TAN-Nummer wurde mit der Funktion TAN_CHK gefunden und auch gleich als ‚verbraucht‘ markiert. Diese TAN ist also "verbraucht" worden.

gelöscht

bedeutet, diese TAN-Nummer wurde vom BASIC-Programm mit der Funktion TAN_DEL gelöscht. Auf diese Weise wird eine alte TAN-Liste für ungültig erklärt, auch wenn noch gültige TAN-Nummern in dieser Liste waren.

Zur besseren Unterscheidung werden die Attribute ‚verbraucht‘ und ‚gelöscht‘ getrennt geschrieben.

TAN_CHK

TAN_CHK (TAN\$, Start_adr, Anzahl)

TAN_CHK: Prüft ob gültige TAN, setz „benutzt“
(nur in Projektmodell 'PM_FULL')

Funktion: Sucht in einer FLASH-Tabelle, ob eine bestimmte TAN dort vorkommt oder nicht. Ein gefundener TAN-Eintrag wird sofort als verbraucht markiert. Wenn eine TAN bereits als verbraucht markiert wurde, so wird als Funktionswert ein Fehler zurückgegeben.

TAN\$	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die aktuelle TAN-Nummer. Die Länge von TAN\$ beträgt 4...12 Bytes und wird als die Länge der Nummern in der Tabelle angenommen.
Start_adr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und der höchsten User-Flash-Adresse und gibt die Startadresse in der TAN-Liste im Flash an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der TANs in der TAN-Liste im Flash an.
Funktionswert	ist vom Typ LONG und gibt das Ergebnis der Funktion folgendermaßen an: 0...nnnn: diese TAN war gültig und ist nun als verbraucht markiert. Der Funktionswert gibt den INDEX an. -1: TAN kommt nicht in der Tabelle vor. -2: TAN kommt zwar in der Tabelle vor, ist jedoch bereits als verbraucht markiert. -3: TAN kommt zwar in der Tabelle vor, ist jedoch gelöscht. Benutzer benötigt neue TAN-Liste. -4: TAN-Tabelle ist leer. Alle TANs sind verbraucht oder gelöscht. -5: TAN-Tabelle ist ungültig. Sicherheit nicht mehr garantiert. Das Flag-Byte eines TAN-Eintrags hat einen unzulässigen Wert. -6: FLASH-Error. Diese TAN-Nummer war o.k., aber beim Schreiben des ‚verbraucht‘-Flags trat ein Flash-Schreibfehler auf. TAN-Sicherheit nicht mehr garantiert. -7: Parameter-Fehler: Adresse außerhalb des User-Flash Anzahl TANs zu groß TAN-Platz wäre größer als

TAN_CHK

FLASH-Bereich

TAN-Stringlänge nicht zwischen 4 und 16 Byte

TAN_CHK wird ab Modulversion 1.06n unterstützt.

Die Modulversion ist per Software Laufzeit abfragbar mit der Funktion SYSVAR_N, über das Kommando Tiger-Status aus dem Menü Anzeige sowie durch einen Aufkleber auf dem Modul gekennzeichnet.

3

TAN_CHK prüft, ob die angegebene TAN in der Liste der TANs im Flash vorkommt und ob die TAN noch unverbraucht ist. Da die Länge der Liste sowie ein Offset angegeben wird, kann es sich auch nur um einen Teil der Gesamtliste handeln. Kommt die Nummer dort vor und ist unverbraucht, dann wird sie sofort als verbraucht markiert. Dieser Vorgang kann nicht rückgängig gemacht werden, außer durch Löschen des gesamten Sektors im Flash, wobei die TAN-Liste auch gelöscht würde.

Tritt ein Fehler ein, wird eine negative Zahl zurückgegeben, während im Erfolgsfall eine Zahl größer oder gleich Null (Indexnummer) als Funktionswert erscheint.

Programmbeispiel:

```

'-----
' Name: TAN_CHK.TIG
' sucht TAN-Nummer und markiert sie als benutzt
'-----
user_var_strict                                ' Vars deklarieren!
#include DEFINE A.INC                          ' allgemeine Definitionen
#define TAN_LEN 4                             ' Laenge einer TAN ohne Flag
#define TANS 5                                 ' Anzahl TANS in der Liste

DATALABEL TAN_LIST                            ' Adressangabe im Flash

TASK MAIN                                     ' Beginn Task MAIN
TAN_LIST::                                   ' 4 Bytes TAN + Flag
    DATA BYTE "20 43 55 FF 9B"%              ' 0. TAN
    DATA BYTE "16 67 89 55 9B"%              ' 1. TAN
    DATA BYTE "34 98 72 43 9B"%              ' 2. TAN
    DATA BYTE "74 92 06 49 9B"%              ' 3. TAN
    DATA BYTE "12 99 47 52 9B"%              ' 4. TAN

    BYTE NTAN                                ' Anzahl TANS
    LONG R                                    ' Ergebnis
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

' zaehle unverbrauchte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, " unused: ";NTAN
R = TAN_CHK ( "42 42 42 42"% , TAN_LIST, TANS ) ' falsche TAN
PRINT #1, "1. check: ";R
R = TAN_CHK ( "20 43 55 FF"% , TAN_LIST, TANS ) ' richtige TAN
PRINT #1, "2. check: ";R

' zaehle unverbrauchte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, " unused: ";NTAN
END                                           ' Ende Task MAIN

```

Das folgende Beispiel verwendet INPUT und eine Wandlung mittels VAL_NUM ins binäre Format, um den vollen möglichen Codebereich der TANs auszunutzen. Das Beispiel ist auf dem Plug & Play Lab ausführbar. Beachten Sie, daß die 9Bh das Flag-Byte darstellt und nicht zur TAN gehört.

Programmbeispiel:

```

'-----
' Name: TAN_CHK1.TIG
' Eingabe TAN-Nummer per INPUT
' sucht TAN-Nummer und markiert sie als benutzt oder zeigt Fehler
' zum Test geben Sie eine der TANs aus DATA BYTE ein (ohne die 9B)
'-----
user_var_strict                                ' Vars deklarieren!

```

TAN_CHK

3

```
#include DEFINE A.INC      ' allgemeine Definitionen
#include UFUNC3.INC        ' User-Function-Codes
#include KEYB_PP.INC       ' dt. Keyboard Belegung

#define TAN_LEN 4          ' Laenge einer TAN ohne Flag
#define TANS 5             ' Anzahl TANS in der Liste

DATALABEL TAN_LIST        ' Adressangabe im Flash

TASK MAIN
TAN_LIST::
    DATA BYTE "20 43 55 FF 9B"% ' 0. TAN
    DATA BYTE "16 67 89 55 9B"% ' 1. TAN
    DATA BYTE "34 98 72 43 9B"% ' 2. TAN
    DATA BYTE "74 92 06 49 9B"% ' 3. TAN
    DATA BYTE "12 99 47 52 9B"% ' 4. TAN

    BYTE NTAN              ' Anzahl TANS
    LONG T, R              ' TAN num., Ergebnis
    STRING T$              ' INPUT-String
    ' LCD-Treiber installieren (BASIC-Tiger)
    INSTALL DEVICE #1, "LCD1.TDD"
    ' LCD-Treiber installieren (TINY-Tiger)
    ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
    CALL INIT KEYB (1)      ' setze Keyboard + div.
    PUT #1, #0, #UFCO_KB_ECHO, YES ' Echo der Eingabe auf LCD

AGAIN:
    PRINT #1, "<1>input TAN:"; ' Ausgabe auf LC-Display
    INPUT #1, T$            ' STRING-Input
    PRINT #1, "<10>";          ' auf neue Zeile
    T$ = "$" + T$          ' wandele nun in String
                             ' mit HEX-Ziffern
    T = VAL_NUM ( T$ )      ' Wert der Eingabe
    T$ = " "               ' initialisiere zu 4 Bytes
    T$ = NTOS$ ( T$, 0, -4, T ) ' Zahl reinlegen (high erst)

                             ' zaehle verbrauchte
    NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_USED )
    PRINT #1, "          used:";NTAN

    R = TAN_CHK ( T$, TAN_LIST, TANS ) ' pruefe TAN
    PRINT #1, "<1>checked result:";R
    IF R < 0 THEN            ' wenn Fehler
        WAIT_DURATION 1000 ' lasse Result sehen
        GOTO AGAIN         ' versuche es nochmal
    ENDIF

                             ' zaehle verbrauchte
    NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_USED )
    PRINT #1, "          used:";NTAN
END                          ' Ende Task MAIN
```

Siehe auch: TAN_COUNT, TAN_DEL, TAN_FIRST

TAN_COUNT

TAN_COUNT (*Start_adr, TAN_Len, Anzahl, Typ*)

TAN_COUNT: Zählt TANs: freie, benutzte, gelöschte
(nur in Projektmodell 'PM_FULL')

Funktion: Zählt das Auftreten von TANs eines bestimmtem Typs in einer TAN-Liste.

Start_adr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und der höchsten User-Flash-Adresse und gibt die Startadresse der TAN-Liste im Flash an.
TAN_Len	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der TAN-Nummern (ohne Flag-Byte) in der TAN-Liste im Flash an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der TANs in der TAN-Liste im Flash an.
Typ	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welcher Typ TAN in der Liste gesucht werden soll (Bit 0 und Bit 1): 11: unverbraucht 10: verbraucht 01: gelöscht 00: verbraucht und gelöscht
Funktionswert	ist vom Typ LONG und gibt das Ergebnis der Funktion folgendermaßen an: 0...nnnn: Anzahl der gefundenen TANs des gesuchten Typs. -7: Parameter-Fehler: Adresse außerhalb des User-Flash Anzahl TANs zu groß TAN-Platz wäre größer als FLASH-Bereich TAN-Stringlänge nicht zwischen 4 und 16 Byte

TAN_COUNT wird ab Modulversion 1.06n unterstützt.
Die Modulversion ist per Software Laufzeit abfragbar mit der Funktion SYSVARn, über das Kommando Tiger-Status aus dem Menü Anzeige sowie durch einen Aufkleber auf dem Modul gekennzeichnet.

TAN_COUNT zählt alle TANs innerhalb des angegebenen Bereichs der TAN-Liste, die dem gesuchten Status entsprechen: unverbraucht, verbraucht, gelöscht oder verbraucht und gelöscht. Da die Länge der Liste sowie ein Offset angegeben wird, kann es sich um die Gesamtliste oder einen Teil der Liste handeln.

TAN_COUNT

Programmbeispiel:

```
'-----
' Name: TAN_COUNT.TIG
' zaehlt TANS eines Typs der Liste ab Startadresse
'-----
user_var_strict                               ' Vars deklarieren!
#include DEFINE_A.INC                         ' allgemeine Definitionen
#define TAN_LEN 4                             ' Laenge einer TAN ohne Flag
#define TANS 5                                ' Anzahl TANS in der Liste

DATALABEL TAN_LIST                           ' Adressangabe im Flash

TASK MAIN                                    ' Beginn Task MAIN
TAN_LIST::                                  ' 4 Bytes TAN + Flag
    DATA BYTE "20 43 55 FF 9B"%             ' 0. TAN
    DATA BYTE "16 67 89 55 9B"%             ' 1. TAN
    DATA BYTE "34 98 72 43 9B"%             ' 2. TAN
    DATA BYTE "74 92 06 49 9B"%             ' 3. TAN
    DATA BYTE "12 99 47 52 9B"%             ' 4. TAN

    BYTE NTAN                                ' Anzahl TANS
    LONG R, ADR                              ' Ergebnis, Adresse
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
' zaehle unverbrauchte

NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, "no. of unused: ";NTAN             ' sollte 5 sein
' andere Startadresse
ADR = TAN_LIST + 2 * (TAN_LEN+1)             ' 2* (TAN-Laenge + Flagbyte)
' zaehle unverbrauchte

NTAN = TAN_COUNT ( ADR, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, "no. of unused: ";NTAN             ' sollte 3 sein
END                                           ' Ende Task MAIN
```

Siehe auch: TAN_CHK, TAN_DEL, TAN_FIRST

TAN_DEL

TAN_DEL (Start_adr, TAN_Len, Anzahl)

TAN_DEL: Löscht TAN
(nur in Projektmodell 'PM_FULL')

Funktion: Löscht eine **Anzahl** von TANs aus einer TAN-Liste ab der **Start_adr**esse.

Start_adr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und der höchsten User-Flash-Adresse und gibt die Startadresse der TAN-Liste im Flash an.
TAN_Len	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der TAN-Nummern (ohne Flag-Byte) in der TAN-Liste im Flash an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der TANs an, die aus der TAN-Liste im Flash gelöscht werden.
Funktionswert	ist vom Typ LONG und gibt das Ergebnis der Funktion folgendermaßen an: 0...nnnn: Anzahl der TANs, die gelöscht wurden. TANs, die bereits gelöscht waren, werden nicht gezählt. -5: TAN-Tabelle ist ungültig. Sicherheit nicht mehr garantiert. Das Flag-Byte eines TAN-Eintrags hat einen unzulässigen Wert. -6: FLASH-Error. Diese TAN-Nummer war o.k., aber beim Schreiben des ‚verbraucht‘-Flags trat ein Flash-Schreibfehler auf. TAN-Sicherheit nicht mehr garantiert. -7: Parameter-Fehler: Adresse außerhalb des User-Flash Anzahl TANs zu groß TAN-Platz wäre größer als FLASH-Bereich TAN-Stringlänge nicht zwischen 4 und 16 Byte

TAN_DEL wird ab Modulversion 1.06n unterstützt.
Die Modulversion ist per Software Laufzeit abfragbar mit der Funktion SYSVARn, über das Kommando Tiger-Status aus dem Menü Anzeige sowie durch einen Aufkleber auf dem Modul gekennzeichnet.

TAN_DEL löscht alle TANs innerhalb des angegebenen Bereichs der TAN-Liste, die dem gesuchten Status entsprechen: unverbraucht, verbraucht, gelöscht oder ver-

TAN_DEL

braucht und gelöscht. Da die Länge der Liste sowie ein Offset angegeben wird, kann es sich um die Gesamtliste oder einen Teil der Liste handeln.

Programmbeispiel:

```

-----
' Name: TAN_DEL.TIG
' loescht 3 TAN aus der Liste
' Wenn das Programm neu gestartet wird, bleibt es bei 3 geloeschten
-----

user_var_strict                                ' Vars deklarieren!
#include DEFINE_A.INC                          ' allgemeine Definitionen
#define TAN_LEN 4                              ' Laenge einer TAN ohne Flag
#define TANS 5                                 ' Anzahl TANS in der Liste

DATALABEL TAN_LIST                            ' Adressangabe im Flash

TASK MAIN                                     ' Beginn Task MAIN
TAN_LIST::                                   ' 4 Bytes TAN + Flag
    DATA BYTE "20 43 55 FF 9B"%              ' 0. TAN
    DATA BYTE "16 67 89 55 9B"%              ' 1. TAN
    DATA BYTE "34 98 72 43 9B"%              ' 2. TAN
    DATA BYTE "74 92 06 49 9B"%              ' 3. TAN
    DATA BYTE "12 99 47 52 9B"%              ' 4. TAN

    BYTE NTAN                                ' Anzahl TANS
    LONG R, ADR                              ' Ergebnis, Adresse
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8

                                           ' zaehle geloeschte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_DELETED )
PRINT #1, "no. of deleted: ";NTAN
ADR = TAN_LIST + 2 * (TAN_LEN+1)              ' 2* (TAN-Laenge + Flagbyte)
R = TAN_DEL ( ADR, TAN_LEN, 2 )               ' loesche 2.+3. TAN
PRINT #1, " delete result: ";R

                                           ' zaehle geloeschte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_DELETED )
PRINT #1, "no. of deleted: ";NTAN              ' sollte 2 sein
WAIT_DURATION 2000

ADR = TAN_LIST + 2 * (TAN_LEN+1)              ' dieselbe Adresse
R = TAN_DEL ( ADR, TAN_LEN, 2 )               ' 2* (TAN-Laenge + Flagbyte)
PRINT #1, "<1> delete result: ";R              ' hat keine TAN geloescht
                                           ' kein Fehler
                                           ' zaehle geloeschte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_DELETED )
PRINT #1, "no. of deleted: ";NTAN              ' sollte 2 sein
WAIT_DURATION 2000

ADR = TAN_LIST + 1 * (TAN_LEN+1)              ' Adresse + Anz=2 ueberlappt
R = TAN_DEL ( ADR, TAN_LEN, 2 )               ' 1* (TAN-Laenge + Flagbyte)
PRINT #1, "<1> delete result: ";R              ' falsch: 5. TAN gibt es nicht
                                           ' hat eine TAN geloescht
                                           ' zaehle geloeschte
NTAN = TAN_COUNT ( TAN_LIST, TAN_LEN, TANS, TAN_DELETED )
PRINT #1, "no. of deleted: ";NTAN              ' sollte 3 sein
WAIT_DURATION 2000

END                                           ' Ende Task MAIN

```

TAN_DEL

Siehe auch: TAN_CHK, TAN_COUNT, TAN_FIRST

TAN_FIRST

TAN_FIRST (*Start_adr*, *TAN_Len*, *Anzahl*, *Typ*)

TAN_FIRST: Sucht 1.TAN: freie, benutzte oder gelöschte
(nur in Projektmodell 'PM_FULL')

Funktion: Sucht das erste Auftreten einer TAN eines bestimmtem Typs in einer TAN-Liste ab der **Start_adresse**.

Start_adr	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG zwischen 0 und der höchsten User-Flash-Adresse und gibt die Startadresse der TAN-Liste im Flash an.
TAN_Len	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Länge der TAN-Nummern (ohne Flag-Byte) in der TAN-Liste im Flash an.
Anzahl	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der TANs in der TAN-Liste im Flash an.
Typ	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt an, welcher Typ TAN in der Liste gesucht werden soll (Bit 0 und Bit 1): 11: unverbraucht 10: verbraucht 01: gelöscht 00: verbraucht und gelöscht
Funktionswert	ist vom Typ LONG und gibt das Ergebnis der Funktion folgendermaßen an: 0...(TAN-Anzahl-1): Index der gefundenen TAN des gesuchten Typs. -7: Parameter-Fehler: Adresse außerhalb des User-Flash Anzahl TANs zu groß TAN-Platz wäre größer als FLASH-Bereich TAN-Stringlänge nicht zwischen 4 und 16 Byte

TAN_FIRST

Programmbeispiel:

3

```
-----
' Name: TAN_FIRST.TIG
' sucht TAN-Nummer und markiert sie als benutzt
-----
user_var_strict                               ' Vars deklarieren!
#include DEFINE_A.INC                         ' allgemeine Definitionen
#define TAN_LEN 4                             ' Laenge einer TAN ohne Flag
#define TANS 5                                ' Anzahl TANS in der Liste

DATALABEL TAN_LIST                           ' Adressangabe im Flash

TASK MAIN                                     ' Beginn Task MAIN
TAN_LIST::                                   ' 4 Bytes TAN + Flag
    DATA BYTE "20 43 55 FF 9B"%             ' 0. TAN
    DATA BYTE "16 67 89 55 9B"%             ' 1. TAN
    DATA BYTE "34 98 72 43 9B"%             ' 2. TAN
    DATA BYTE "74 92 06 49 9B"%             ' 3. TAN
    DATA BYTE "12 99 47 52 9B"%             ' 4. TAN

    BYTE NTAN                                ' Anzahl TANS
    LONG R, ADR                              ' Ergebnis, Adresse
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

                                     ' suche erste unverbrauchte
R = TAN_FIRST ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, " first unused:";R
WAIT_DURATION 2000

                                     ' brauche TAN Nr. 0
R = TAN_CHK ( "20 43 55 FF"% , TAN_LIST, TANS ) ' richtige TAN
PRINT #1, "<1>checked result:";R

                                     ' suche erste unverbrauchte
R = TAN_FIRST ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, " first unused:";R

ADR = TAN_LIST + 1 * (TAN_LEN+1)             ' 2* (TAN-Laenge + Flagbyte)
R = TAN_DEL ( ADR, TAN_LEN, 1 )              ' loesche TAN Nr. 1
PRINT #1, " delete result:";R

                                     ' suche erste unverbrauchte
R = TAN_FIRST ( TAN_LIST, TAN_LEN, TANS, TAN_UNUSED )
PRINT #1, " first unused:";R
END                                           ' Ende Task MAIN
```

Siehe auch: TAN_CHK, TAN_COUNT, TAN_DEL

TANH

TANH (rad)

TANH: TANGENS hyperbolicus, Real-Werte

Funktion: Liefert den Tangenshyperbolicus von **rad** zurück.

rad ist eine Variable, eine Konstante oder ein Ausdruck vom Typ REAL im Bereich von $-4.5 \cdot 10^{15}$ bis $4.5 \cdot 10^{15}$ und entspricht einem Winkel im Bogenmaß in der Einheit **rad**. Eine Typwandlung findet automatisch statt.

Funktionswert ist vom Typ REAL und liegt im Bereich von -1 bis 1. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Umrechnung von Grad in Bogenmaß: Gradwert * PI/180

Programmbeispiel:

```

'-----
' Name: TANH.TIG
'-----
#DEFINE PI_180 0.0174532925199433      ' Konstante PI/180 definieren

TASK MAIN                              ' Beginn Task MAIN
  LONG DEG                             ' LONG-Variable deklarieren
  REAL RAD                             ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "NF<0><1> <15>V0.0.0.0.0.0.0.0,0.0.0.0.0.0.0"
  FOR DEG = 0 TO 360                   ' Winkel von 0 bis 360 Grad
    RAD = DEG * PI_180                 ' Winkel: Grad- zu Bogenmass
    PRINT #1, "<1>TANH ("; DEG; " Grad)" ' Ausgabe auf LC-Display
    PRINT USING #1, "="; TANH (RAD)   ' Tangens hyperbolicus ausgeben
    WAIT DURATION 1000                 ' 1 Sek warten
  NEXT                                ' naechster Winkelwert
END                                    ' Ende Task MAIN

```

Siehe auch: COS, TAN, ASIN, ACOS, ATAN, SINH

TASK

TASK

TASK Name

TASK ... END: Task Definition

Instruktion: Beginn der Definition der Task **Name**.

Der Task MAIN kommt besondere Bedeutung zu. Wichtige Hinweise finden Sie auf Seite 17.

Die Task-Definition endet mit der Anweisung **END**.

Programmbeispiel:

```
' -----  
' Name: TASK.TIG  
' -----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, "Hello World"                ' Auf LC-Display ausgeben  
END                                       ' Ende Task MAIN
```


THIS_VERS

THIS_VERS ()

THIS_VERS: Gibt die aktuelle Version dieses Runtime-Systems zurück

Funktion: Liefert die aktuellen Version des Runtime-Systems zurück. Die Versionsnummer ist codiert.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

THIS_VERS gibt Auskunft über die Version des Runtime-Systems im Modul. Gemeint ist das System, aus welchem der Compiler die Anwendung aufgebaut hat. Folgende Versionen sind im Gesamtsystem im Spiel und zu unterscheiden:

- **Compiler**, das Programm auf dem PC, welches die Anwendung, die am Ende im Modul läuft, aus allen anderen Komponenten erzeugt. Die Komponenten sind Quelltext, Runtime-System, Device-Treiber und Daten, die eventuell ins Flash mit eingebunden werden. Versionsnummer ist die des Produktes Tiger-BASIC[®] und über das Menü Hilfe zu erfahren.
- **Runtime-System**, befindet sich auf dem PC und ist unabhängig vom Compiler update-fähig. Die Versionsnummer ist mit THIS_VERS zu erfahren.
- **Device-Treiber**, liegen als Dateien vor und werden mit in die Anwendung nach Bedarf eingebunden. Device-Treiber können unabhängig von den anderen Komponenten erneuert, modernisiert und hinzugefügt werden. Ihre Versionsnummer kann sowohl von der PC-Oberfläche abgefragt werden (Anzeige Gerätetreiberliste) als auch mit Hilfe von User-Function-Code zur Laufzeit im Modul.
- **Modul**, jedes BASIC-Tiger[®]-Modul hat eine Versionsnummer, die auf dem Modul-Aufkleber steht sowie zur Laufzeit mit der Funktion SYSVARN abgefragt werden kann.

THIS_VERS fragt die Version des Runtime-Systems ab und kann dadurch bei langlebigen, sich weiter entwickelnden Produkten einen Anhaltspunkt über den Stand der Entwicklung zum Zeitpunkt des Ausliefern geben.

Bei Versionssprüngen (z.B. von 4.x auf 5.0) kann es vorkommen, daß die Kompatibilität des neuen Runtime-Systems bzw. der Device-Treiber mit dem älteren Compiler nicht mehr gegeben ist. Grundsätzlich gilt jedoch, das die Komponenten sich unabhängig entwickeln und von Zeit zu Zeit eventuell neuere Versionen im Internet zur Verfügung stehen.

THIS_VERS

Programmbeispiel:

```
'-----
' Name: THIS_VERS.TIG
' Liefert die Version des Run-Time-Systems
'-----
TASK MAIN                                ' Beginn Task MAIN

' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

A = THIS_VERS                            ' Version des Runtime-Systems
CALL CVT_VERS ( A, VRS$ )                ' konvertiere Version in String
PRINT #1, VRS$                           ' Ausgabe konvertiert
END                                       ' Ende Task MAIN

'-----
' Subroutine: Erzeuge Versions-String
'-----
SUB CVT_VERS ( LONG VRS; VAR STRING V$ )
  LONG V, V1                             ' Variablendeklaration
  STRING C$, V1$                          '
  V = VRS                                ' bearbeite in V
  V1$ = ""
  IF V > 0FFFFh THEN                      ' pruefe ob beta-Version
    V1$ = "b"
  ENDIF
  V = V BITAND 0FFFFh                    ' maskiere Versionsnummer
  V$ = STRI$(V,"UH<4><4> 0.0.0.0")        ' wandele Nummer in String
  C$ = RIGHT$( V$, 1 )                  ' wandele rechte Ziffer
  V = ASC ( C$ ) + 49                    ' in Buchstabe
  V$ = V1$+LEFT$(V$,1)+"."+MID$(V$,1,2)+CHR$(V)
END                                       ' Ende subroutine
```

Siehe auch: SYSVARN, User-Function-Codes der Device-Treiber

TICKS

TICKS ()

TICKS: Liefert akt. Zeit des 1 ms System-Taktes

Funktion: Liefert den aktuellen Wert des internen Systemzählers zurück. Der Systemzähler zählt in Millisekunden und beginnt nach Reset mit 0.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

3

Ticks ist die Anzahl der seit dem letzten Reset vergangenen Millisekunden. Der Tick-Zähler läuft nach 2.147.483.647 Ticks, also nach 24 Tagen über. Der Zähler bleibt im positiven Wertebereich der LONG-Zahlen. Die Differenz zwischen zwei Tick-Zählerständen ermittelt die Funktion DIFF_TICKS und berücksichtigt dabei, daß der Zähler gerade übergelaufen sein kann. Mit SET_TICKS wird der Zähler auf einen neuen Wert gesetzt.

Programmbeispiel:

```

' -----
' Name: TICKS.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  LONG T                                ' LONG-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  T = TICKS()                            ' Speichert Ticks in T
  LOOP 100000                            ' 100000 leere Loops
  ENDLOOP
  T = TICKS() - T                          ' Zeitdifferenz nach LOOP
  PRINT #1, 100000000/T; " Loops/Sec."    ' Anzeige von LOOPs/Sekunde
END                                         ' Ende Task MAIN

```

Siehe auch: DIFF_TICKS

TRIM\$

TRIM\$

TRIM\$ (*String1*, *String2*, *Flag*)

TRIM\$: Entfernt führende und/oder nachfolgende Zeichen

Funktion: Entfernt aus **String1** in Abhängigkeit von **Flag** führende und/oder anhängende Zeichen, die in **String2** enthalten sind.

3

String1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist der zu bearbeitende String.
String2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Zeichen, die in String1 bearbeitet werden.
Flag	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt, ob führende Zeichen, anhängende Zeichen oder sowohl führende als auch anhängende Zeichen entfernt werden.
Funktionswert	ist vom Typ STRING und enthält String1 ohne Bytes, die in String2 vorkommen.

Mit der Funktion TRIM\$ können auf einfache Weise Zeichen, die vor oder hinter den zu verarbeitenden Zeichen stehen und die bei der weiteren Verarbeitung stören, aus einer Variablen entfernen. Z.B. sollen führende Nullen einer Ziffernfolge oder anhängende Leerzeichen eines Strings entfernt werden - oder beides zusammen, dann ist TRIM\$ die geeignete Funktion.

Programmbeispiel:

```

'-----
' Name: TRIM$.TIG
'-----
TASK Main                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "  Hello World  "                ' A$ besetzen
  PRINT #1, "<1>";                        ' Display loeschen
  PRINT #1, "***;TRIM$(A$, " ", 1);***" ' links trimmen
  PRINT #1, "***;TRIM$(A$, " ", -1);***" ' rechts trimmen
  PRINT #1, "***;TRIM$(A$, " ", 0);***" ' links & rechts trimmen
  WAIT DURATION 5000                    ' 5 Sek warten
  A$ = ".!.-! Hello World --!.."        ' A$ besetzen
  PRINT #1, "<1>";                        ' Display loeschen
  PRINT #1, "***;TRIM$(A$, ".!- ", 1);***" ' links trimmen
  PRINT #1, "***;TRIM$(A$, ".!- ", -1);***" ' rechts trimmen
  PRINT #1, "***;TRIM$(A$, ".!- ", 0);***" ' links & rechts trimmen
END                                      ' Ende Task MAIN

```

3

Siehe auch: REMDOUBLE\$, REMOVE\$, LEFT\$, RIGHT\$

UNGET_FIFO

UNGET_FIFO

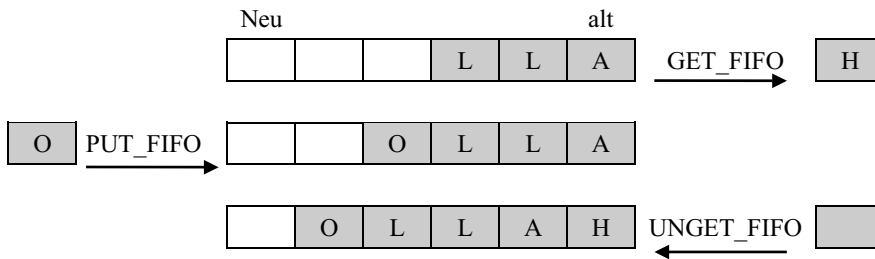
UNGET_FIFO Name, Element[, Fehlerbehandlung]

UNGET_FIFO: Mache GET_FIFO rückgängig (wenn möglich)

Instruktion: Schiebt ein mit GET_FIFO gelesenes **Element** wieder in den FIFO-Puffer **Name** zurück. (FIFO=First-In-First-Out)

Element ist vom Typ BYTE, WORD, oder LONG.

Diese Operation ist z.B. erforderlich, wenn das Endezeichen einer Zeichenkette zwar erkannt, aber selbst nicht gelesen werden soll.



Programmbeispiel:

```

' -----
' Name: UNGET_FIFO.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
  FIFO BUFFER (512) OF WORD              ' FIFO-Buffer definieren
  WORD I,J1,J2                          ' WORD-Variablen definieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  FOR I = 0 TO 511                      ' 512 Werte in den
    PUT_FIFO BUFFER, I                 ' FIFO-Buffer schreiben
  NEXT I                                '
  FOR I = 0 TO 18                      ' 19 mal jeweils
    GET_FIFO BUFFER, J1                ' 2 Werte aus dem
    GET_FIFO BUFFER, J2                ' FIFO-Buffer auslesen,
    PRINT #1, "<1>I  ="; I             ' Schleifenzaehler ausgeben,
    PRINT #1, "J1 ="; J1               ' FIFO-Inhalt 1 ausgeben,
    PRINT #1, "J2 ="; J2               ' FIFO-Inhalt 2 ausgeben,
    WAIT DURATION 1000                 ' 1 Sek warten und
    UNGET_FIFO BUFFER, J2              ' J2 zurueck in den Buffer
  NEXT I                                '
END                                     ' Ende Task MAIN

```

3

Siehe auch: GET_FIFO, PUT_FIFO

UNPACK_BYTE\$

UNPACK_BYTE\$

UNPACK_BYTE\$ (*String*, *Start*, *Anzahl*, *Flag*, *Typ*)

UNPACK_BYTE\$: De-Komprimiert Strings
(nur in Projektmodell 'PM_FULL')

Funktion: Dekomprimiert die angegebene **Anzahl** Bytes aus **String** ab der Position **Start**. Dies ist die inverse Funktion zu PACK_BYTE\$.

3

- String** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING oder eine Adresse im Flash und enthält die zu dekomprimierenden Zeichen.
- Start** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anfangsposition der Dekomprimierung an.
- Anzahl** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und legt die Anzahl der Bytes fest, die bei der Dekomprimierung entstehen.
- Flag** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG im Bereich von 0...255 und legt das Kompressionsflag fest.
- Typ** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und bestimmt die Kompressionsart (zur Zeit ist Typ immer = 0).
- Funktionswert** ist vom Typ STRING und enthält den dekomprimierten String.

UNPACK_BYTE\$ mit Kompressionstyp 0: ersetzt die bei der Komprimierung eingesetzten Sequenzen durch die entsprechende Anzahl aufeinanderfolgender gleicher Codes, z.B.:

"<Flag><6>A" wird zu "AAAAAA"

Die Anzahl (hier 6) liegt im Bereich 4...255. Es gibt folgende Sondersequenzen:

Sequenz	Bedeutung
<Flag><0>	Endesequenz
<Flag><1>	1x Flag
<Flag><2>	2x Flag
<Flag><3>	3x Flag

Programmbeispiel:

```

-----
' Name: UNPACK_BYTE$.TIG
-----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$, B$                          ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  USING "UH<2><2>*k 2 2 2 2 2"          ' fuer Ausgabe als HEX
  A$ = "<11H><25>A"                      ' das is A$ (25xA)
  PRINT #1, "A$=";                      ' zeige A$
  FOR I = 0 TO LEN(A$)-1                ' hier: 4 Bytes
    PRINT_USING #1, ASC(MID$(A$,I,1));
  NEXT
  B$ = UNPACK_BYTE$ ( A$, 0, 3, 11H, 0 ) ' dekomprimiere 3 Bytes
  PRINT #1, "<13><10>B$=";                ' zeige Ergebnis in B$
  FOR I = 0 TO 4                        ' erste 5 Bytes
    PRINT_USING #1, ASC(MID$(B$,I,1));
  NEXT
  PRINT #1, "<13><10>LEN A$=";LEN(A$);", B$=";LEN(B$)
END                                    ' Ende Task MAIN

```

3

Siehe auch: PACK_BYTE\$, BYTE_ANIB\$, Incremental-Conversion BYTE_INIB\$, INIB_BYTE\$

UPPER\$

UPPER\$

UPPER\$ (*String*)

UPPER\$: Konvertiert Buchstaben in String zu Großbuchstaben.

Funktion: Liefert den in Großbuchstaben umgewandelten **String** zurück.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und entspricht dem Quellstring für die Umwandlung in Großbuchstaben.

Funktionswert ist vom Typ STRING.

Die Zuweisungsvariable (Funktionswert) kann mit der Argumentvariablen identisch sein. In diesem Falle ist die Verarbeitungsgeschwindigkeit höher als bei der Zuweisung zu einer anderen Variablen.

Programmbeispiel:

```
'-----  
' Name: UPPER$.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, UPPER$ ("Hello World")      ' Ausgabe auf LC-Display  
END                                      ' Ende Task MAIN
```

Siehe auch: UPPER7\$

UPPER7\$

UPPER7\$ (*String*)

UPPER7\$: Entfernt ev. gesetztes Bit-7 und Konvertiert Buchstaben in String zu Großbuchstaben.

Funktion: Liefert den in Großbuchstaben umgewandelten **String** zurück. Setzt gleichzeitig das höchste Bit jedes Zeichens zurück, so daß reiner 7-Bit-Code entsteht.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und entspricht dem Quellstring für die Umwandlung in Großbuchstaben.

Funktionswert ist vom Typ STRING und enthält nur ASCII-Zeichen<128.

Die Zuweisungsvariable (Funktionswert) kann mit der Argumentvariablen identisch sein. In diesem Falle ist die Verarbeitungsgeschwindigkeit höher als bei der Zuweisung zu einer anderen Variablen.

Programmbeispiel:

```

' -----
' Name: UPPER7$.TIG
' -----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #1, UPPER7$ ("F4 E5 F3 F4"%)    ' Ausgabe auf LC-Display
END                                     ' Ende Task MAIN

```

Siehe auch: UPPER\$

USING

USING

USING [*Taskname*] *String*

USING: Format-Anweisung für PRINT_USING

Instruktion: Gibt einen oder mehrere Formatstrings an, die für mit PRINT_USING ausgegebene numerische Werte in der Task **Taskname** verwendet werden.

3

Taskname gibt an, für welche Task die Formatstrings gesetzt werden sollen. Jede Task kann eigene USING-Festlegungen treffen, ohne daß es eine Beeinflussung gibt oder zur Laufzeit ständig zwischen verschiedenen Einstellungen gewechselt werden muß. Wird Taskname nicht angegeben, wird die aktuelle Task angenommen.

String besteht aus einem oder mehreren Formatanweisungen.

Die Formatanweisungen werden in der im String angegebenen Reihenfolge verwendet und anschließend von vorne beginnend wiederholt. Jeder numerische Wert, der ausgegeben wird, verwendet die nächste gerade anstehende Formatanweisung.

Die Ausgabe findet mit system-internen Standard-Formatstrings statt, wenn unpassende Formatstrings vorgefunden werden, z.B. ein Formatstring für LONG-Zahlen bei der Ausgabe einer REAL-Zahl (=Float-Zahl).

Strings mit USING formatieren geht so ganz einfach:

1. „using“ hinschreiben.
2. F1 drücken (Hilfe).
3. Link auf Beispiele anklicken, passend zu BYTE, WORD, LONG oder REAL.
4. Gewünschten Formatstring markieren
5. Mit Strg-C in die Zwischenablage kopieren.
6. Im Quelltext mit Strg-V einfügen.

Eine Beschreibung der Formatstrings finden Sie auf den folgenden Seiten.

Formatstrings

Formatstring für Dezimale und Hexadezimale numerische Ausgabe von BYTE, WORD und LONG-Zahlen:

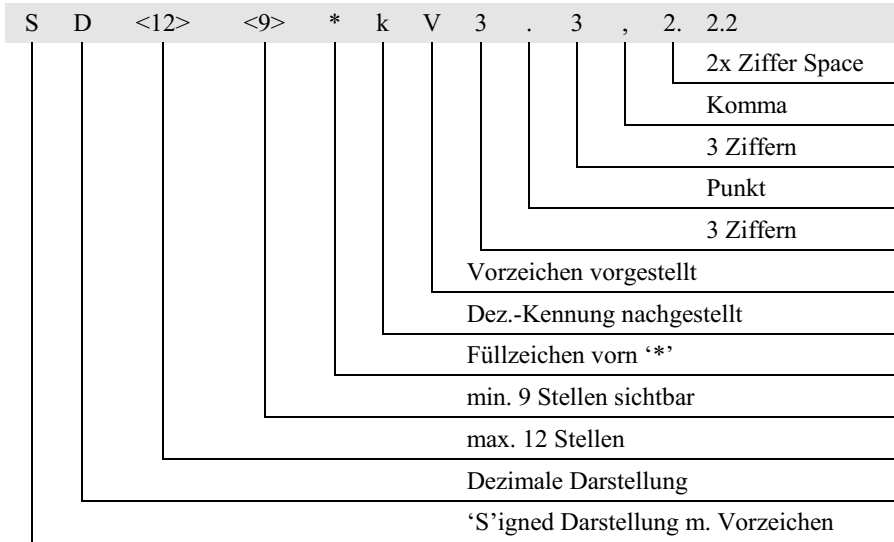
Pos	Parameter	Bedeutung
0	U	Zahl ist nicht vorzeichenbehaftet (unsigned)
	S	Zahl ist vorzeichenbehaftet (signed)
1	D	Dezimalzahl
	H	Hexadezimalzahl
2	0	variable Länge
	<1>...<31>	Länge maximal wie angegeben
3	<Zahl>	mindestens soviel Ziffern darstellen
4	ASCII-Zeichen	Auffüllzeichen vor der Zahl
5	Leerzeichen	keine Kennung Dez/Hex
	K	Kennung Dez/Hex wird vorgestellt
	k	Kennung Dez/Hex wird nachgestellt
6	Leerzeichen	ohne Vorzeichen
	V	Vorzeichen wird vorgestellt
	v	Vorzeichen wird nachgestellt
7, 8	<Zahl><Trenn>	Segmentierung: Anzahl Ziffern (0...9), Trennzeichen
9, 10	<Zahl><Trenn>	Segmentierung: Anzahl Ziffern (0...9), Trennzeichen
11, 12	<Zahl><Trenn>	Segmentierung: Anzahl Ziffern (0...9), Trennzeichen
13, 14	<Zahl><Trenn>	Segmentierung: Anzahl Ziffern (0...9), Trennzeichen
15	<Zahl>	Segmentierung: Anzahl Ziffern (0...9) (insgesamt 5x Ziffernangabe, 4x Trennzeichen)

Beachte: das Füllzeichen ist nie eine Ziffer. Für führende Nullen erhöhen Sie den Parameter 3 'mindestens soviel Ziffern darstellen'.

USING

Die Segmentierung der Ausgabe legt fest, nach wieviel Ziffern welches Trennzeichen eingefügt wird und muß stets vollständig angegeben werden.

Zur besseren Übersicht ist in der folgenden Abbildung der Formatstring etwas auseinandergezogen. Er enthält in diesem Beispiel jedoch keine Leerzeichen.



```
USING "SD<12><9>*kV3.3,2.2"  
PRINT_USING #1, -50123456
```

Die LONG-Zahl `'-50123456'` wird mit obigem Formatierungsstring durch den `PRINT_USING`-Instruktion so ausgegeben:

```
-.***05.012,34 56#
```

Vorzeichenbehaftete (S) Dezimalzahl (D) mit maximal 12 Ziffern, mindestens 9 Ziffern sichtbar, Füllzeichen (*), Kennzeichen nachgestellt (k), Vorzeichen vorgestellt (V), Trennzeichen sind nach 3 Ziffern der Punkt, nach weiteren 3 Ziffern das Komma, dann 2 mal nach jeweils 2 Ziffern ein Punkt.

Hier finden Sie eine andere Darstellung des Formatstrings und die Zuordnung der Parameter zu zwei konkreten Zahlen '-12345678' und '654321', die jeweils 4 künstlich erzeugte Nachkommastellen haben sollen:

SD<12><9>*kV3.3.3,2 2 'SD<12><9>' bedeutet:

Eine vorzeichenbehaftete (**S**igned) **D**ezimalzahl mit **12** Stellen, davon mindestens **9** als Ziffern sichtbar. So sehen die Zahlen aus, wenn sie mit PRINT_USING ausgegeben werden:

-*01.234,56 78#** (-12345678)

*******65,43 21#** (+654321)

Da BYTE-, WORD- und LONG-Zahlen immer ganze Zahlen sind, wird das Komma in der Ausgabe durch den Formatstring erzeugt.

SD<12><9>*kV3.3.3,2 2 '*kV' bedeutet:

Das Füllzeichen in obigen Zahlen ist '*', die Kennung für Dezimalzahlen ('#') ist nachgestellt (kleines 'k'), und das Vorzeichen ist vorgestellt (großes 'V'). Das positive Vorzeichen erscheint als Leerzeichen.

SD<12><9>*kV3.3.3,2 2 '3.3.3,2 2' bedeutet:

Die Formatierung mit Trennzeichen wird bei den Ganzzahlen BYTE, WORD und LONG von rechts her vorgenommen: 2 Ziffern, dann ein Leerzeichen, wieder 2 Ziffern und das Komma, dann jeweils 3 Ziffern, die durch Punkte getrennt sind. Die 4 Nachkommastellen werden durch die Formatierungsanweisung in der Ausgabe dargestellt.

USING

Ein kleines Beispiel zeigt die Anwendung von USING:

Programmbeispiel:

```
'-----
' Name: USING.TIG
' Tip: Schneiden Sie USING-Strings aus der Hilfe aus und fuegen
' in Ihren Quelltext ein.
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
  REAL R                                ' REAL-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  L = -2012345671                        ' LONG-Variable besetzen
  R = -1501.23456                        ' REAL-Variable besetzen
  USING "SD<11><1>*kv3.3.3.3"            ' Format-String fuer LONG
  PRINT_USING #1, "L ="; L              ' Ausgabe auf LC-Display
  USING "NF<7><3>*<4>V0.0.0.&          ' Format-String fuer REAL
0.0.1.3.3,4.0.0.0.0.0.0"              ' Ausgabe auf LC-Display
  PRINT_USING #1, "R ="; R
END                                      ' Ende Task MAIN
```

Das folgende Beispiel zeigt, wie 2 Tasks (oder mehr) für ihre Ausgaben verschiedene USING-Formate gleichzeitig benutzen, ohne daß durch das Multitasking mal das falsche Format verwendet würde. Die Formate werden jeweils nur für die entsprechenden Task gesetzt:

Programmbeispiel:

```
'-----
' Name: USING2.TIG
' Tip: Schneiden Sie USING-Strings aus der Hilfe aus und fuegen
' in Ihren Quelltext ein.
' Hier: demonstriert, dass USINGs taskweise gelten.
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG EVER, L                          ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8

  RUN_TASK ALSO_USING, 3                ' 2. Task, die USING benutzt
'
  L = -20123                            ' LONG-Variable besetzen
  FOR EVER = 0 TO 0 STEP 0
    USING "SD<7><1> V 0.0.0.3.3"         ' Format-String fuer LONG
    PRINT_USING #1, "<1Bh>A<0><0><0F0h>L ="; L' Ausgabe auf LC-Display
    L = L + 1
  NEXT
END                                      ' Ende Task MAIN
```



```

' -----
' Diese Task verwendet ein anderes Ausgabeformat.
' Die Formate vermischen sich jedoch nicht.
' -----
TASK ALSO_USING
  LONG EVER, L                                ' LONG-Variable deklarieren

  L = 134744                                  ' LONG-Variable besetzen
  FOR EVER = 0 TO 0 STEP 0
    USING "SD<6><1>  0.0.2:2:2"                ' Format-String fuer LONG
    PRINT_USING #1, "<1Bh>A<0><1><0F0h>L ="; L' Ausgabe auf LC-Display
    L = L + 1
  NEXT
END

```

USING

Ein weiteres Beispiel zeigt, wie unter Verwendung der Formatstrings numerische Integerwerte ausgegeben werden. Das Ergebnis im Ausdruck finden Sie auf den darauffolgenden Seiten.

Programmbeispiel:

```
'-----
' Name: USING_INT.TIG
'-----
#include DEFINE_A.INC                                ' allg. Symbol-Definitionen

TASK MAIN                                            ' Start Task MAIN
  LONG L                                            ' LONG-Variable deklarieren
  INSTALL DEVICE #2, "SER1B_K1.TDD",&              ' ser. Treiber installieren
  BD_19_200, DP_8N, JA,&                            ' Einstellungen Kanal 0
  BD_38_400, DP_8E, JA                            ' Einstellungen Kanal 1
  TABS "<10>L"                                       ' setze TAB

'-----
  L = 12345678                                     ' setze LONG-Variable

  PRINT #2, #0, "-----";
  PRINT #2, #0, "-----";
  PRINT #2, #0, "Ausgabe: unsigned, dezimal, ohne Fuellzeichen, kons";
  PRINT #2, #0, "t. Laenge,"
  PRINT #2, #0, "      ohne Kennung, ohne Vorzeichen, mit Segment";
  PRINT #2, #0, "ierung"
  PRINT #2, #0, "Wert:      12345678 (LONG)"
  PRINT #2, #0, "-----";
  PRINT #2, #0, "-----";

  USING "UD<8><5>    0.0.0.7.1"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.7.1";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.6.2"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.6.2";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.5.3"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.5.3";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.4.4"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.4.4";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.3.5"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.3.5";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.2.6"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.2.6";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"

  USING "UD<8><5>    0.0.0.1.7"
  PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>    0.0.0.1.7";CHR$(34);
  PRINT USING #2, #0, , "|";L;"|"
```

```

USING "UD<8><5> 0.0.0.0.8"
PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5> 0.0.0.0.8";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

PRINT #2, #0, ""
PRINT #2, #0, ""
PRINT #2, #0, ""

'-----
L = 12345                                     ' setze LONG-Variable

PRINT #2, #0, "-----";
PRINT #2, #0, "-----"
PRINT #2, #0, "Ausgabe: unsigned, dezimal, mit Fuellzeichen, var. ";
PRINT #2, #0, "Laenge"
PRINT #2, #0, "      ohne Kennung, ohne Vorzeichen, ohne Segmen";
PRINT #2, #0, "tierung"
PRINT #2, #0, "Wert:      12345 (LONG)"
PRINT #2, #0, "-----";
PRINT #2, #0, "-----"

USING "UD<9><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<9><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<8><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<7><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<7><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<6><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<6><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<5><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<5><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<4><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<4><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

USING "UD<3><5>* 0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<3><<5>* 0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"| "

PRINT #2, #0, ""
PRINT #2, #0, ""
PRINT #2, #0, ""

'-----
L = 123456789                                 ' setze LONG-Variable

PRINT #2, #0, "-----";
PRINT #2, #0, "-----"
PRINT #2, #0, "Ausgabe: unsigned, dezimal, Leerz.als Fuellz., var.";
PRINT #2, #0, "Laenge"

```

USING

3

```

PRINT #2, #0, "           ohne Kennung, ohne Vorzeichen, ohne Segmen";
PRINT #2, #0, "tierung"
PRINT #2, #0, "Wert:      123456789 (LONG)"
PRINT #2, #0, "-----";
PRINT #2, #0, "-----"

USING "UD<9><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<9><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<8><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<8><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<7><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<7><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<6><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<6><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<5><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<5><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<4><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<4><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<3><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<3><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<2><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<2><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<1><1>      0.0.0.0.0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<1><<1>      0.0.0.0.0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

PRINT #2, #0, ""
PRINT #2, #0, ""
PRINT #2, #0, ""

'-----
L = 123456789                                ' setze LONG-Variable

PRINT #2, #0, "-----";
PRINT #2, #0, "-----"
PRINT #2, #0, "Ausgabe: unsigned, hex, mit Fuellzeichen, konst. La";
PRINT #2, #0, "enge"
PRINT #2, #0, "           mit Kennung, ohne Vorzeichen, mit Segmenti";
PRINT #2, #0, "erung"
PRINT #2, #0, "Wert:      123456789 (LONG)"
PRINT #2, #0, "-----";
PRINT #2, #0, "-----"

```

```

USING "UH<8><8>*k 0 2 2 2 2"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*k 0 2 2 2 2";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*k 0 0 0 4 4"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*k 0 0 0 4 4";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*k 0 0 0 0 8"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*k 0 0 0 0 8";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*K 0 2 2 2 2"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*K 0 2 2 2 2";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*K 0 0 0 4 4"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*K 0 0 0 4 4";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*K 0 0 0 0 8"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*K 0 0 0 0 8";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

PRINT #2, #0, ""
PRINT #2, #0, ""
PRINT #2, #0, ""

-----
L = -0ABCh                                ' setze LONG-Variable

PRINT #2, #0, "-----";
PRINT #2, #0, "-----"
PRINT #2, #0, "Ausgabe: unsigned, hex mit/dez ohne Fuellzeichen, v";
PRINT #2, #0, "ar. Laenge"
PRINT #2, #0, "mit/ohne Kennung/Vorzeichen, ohne Segmenti";
PRINT #2, #0, "erung"
PRINT #2, #0, "Wert:      -2748 (LONG)"
PRINT #2, #0, "-----";
PRINT #2, #0, "-----"

USING "UH<8><8>*kV0 0 0 0 8"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*kV0 0 0 0 8";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UH<8><8>*k 0 0 0 0 8"
PRINT #2, #0, "USING ";CHR$(34);"UH<<8><<8>*k 0 0 0 0 8";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<10><4> V0 0 0 0 0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<10><<4> V0 0 0 0 0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

USING "UD<10><4> 0 0 0 0 0"
PRINT #2, #0, "USING ";CHR$(34);"UD<<10><<4> 0 0 0 0 0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

PRINT #2, #0, ""
PRINT #2, #0, ""

```

USING

3

```
PRINT #2, #0, ""
```

```
-----  
L = -123456789                                ' setze LONG-Variable
```

```
PRINT #2, #0, "-----";  
PRINT #2, #0, "-----"  
PRINT #2, #0, "Ausgabe: signed, dezimal, ohne Fuellzeichen, konst.";  
PRINT #2, #0, " Laenge"  
PRINT #2, #0, "          mit Kennung, mit Vorzeichen, ohne Segmenti";  
PRINT #2, #0, "erung"  
PRINT #2, #0, "Wert:      -12345789 (LONG)"  
PRINT #2, #0, "-----";  
PRINT #2, #0, "-----"
```

```
USING "SD<9><5> kv0.0.0.0.0"  
PRINT #2, #0, "USING ";CHR$(34);"SD<<9><<5> kv0.0.0.0.0";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SD<9><5> kv0.0.0.0.0"  
PRINT #2, #0, "USING ";CHR$(34);"SD<<9><<5> kv0.0.0.0.0";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SD<9><5> Kv0.0.0.0.0"  
PRINT #2, #0, "USING ";CHR$(34);"SD<<9><<5> Kv0.0.0.0.0";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SD<9><5> KV0.0.0.0.0"  
PRINT #2, #0, "USING ";CHR$(34);"SD<<9><<5> KV0.0.0.0.0";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
PRINT #2, #0, ""  
PRINT #2, #0, ""  
PRINT #2, #0, ""
```

```
-----  
L = -0ABCh                                ' setze LONG-Variable
```

```
PRINT #2, #0, "-----";  
PRINT #2, #0, "-----"  
PRINT #2, #0, "Ausgabe: signed, hex mit/dez ohne Fuellzeichen, var";  
PRINT #2, #0, ". Laenge"  
PRINT #2, #0, "          mit/ohne Kennung/Vorzeichen, ohne Segmenti";  
PRINT #2, #0, "erung"  
PRINT #2, #0, "Wert:      -2748 (LONG)"  
PRINT #2, #0, "-----";  
PRINT #2, #0, "-----"
```

```
USING "SH<8><8>*kv0 0 0 0 8"  
PRINT #2, #0, "USING ";CHR$(34);"SH<<8><<8>*kv0 0 0 0 8";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SH<8><8>*k 0 0 0 0 8"  
PRINT #2, #0, "USING ";CHR$(34);"SH<<8><<8>*k 0 0 0 0 8";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SD<10><4> V0 0 0 0 0"  
PRINT #2, #0, "USING ";CHR$(34);"SD<<10><<4> V0 0 0 0 0";CHR$(34);  
PRINT USING #2, #0, , "|";L;"|"
```

```
USING "SD<10><4>    0 0 0 0 0"
PRINT #2, #0, "USING ";CHR$(34);"SD<<10><<4>    0 0 0 0 0";CHR$(34);
PRINT USING #2, #0, , "|";L;"|"

PRINT #2, #0, ""
PRINT #2, #0, ""
PRINT #2, #0, ""

END
```

USING

3

Ausgabe: unsigned, dezimal, ohne Füllzeichen, konst. Laenge, ohne Kennung, ohne Vorzeichen, mit Segmentierung		
Wert: 12345678 (LONG)		

USING "UD<8><5>	0.0.0.8.0"	12345678
USING "UD<8><5>	0.0.0.7.1"	1234567.8
USING "UD<8><5>	0.0.0.6.2"	123456.78
USING "UD<8><5>	0.0.0.5.3"	12345.678
USING "UD<8><5>	0.0.0.4.4"	1234.5678
USING "UD<8><5>	0.0.0.3.5"	123.45678
USING "UD<8><5>	0.0.0.2.6"	12.345678
USING "UD<8><5>	0.0.0.1.7"	1.2345678
USING "UD<8><5>	0.0.0.0.8"	12345678

Ausgabe: unsigned, dezimal, mit Fuellzeichen, var. Laenge ohne Kennung, ohne Vorzeichen, ohne Segmentierung		
Wert: 12345 (LONG)		

USING "UD<9><5>0	0.0.0.0.0"	000012345
USING "UD<8><5>0	0.0.0.0.0"	00012345
USING "UD<7><5>0	0.0.0.0.0"	0012345
USING "UD<6><5>0	0.0.0.0.0"	012345
USING "UD<5><5>0	0.0.0.0.0"	12345

Ausgabe: unsigned, dezimal, ohne Fuellzeichen, var. Laenge ohne Kennung, ohne Vorzeichen, ohne Segmentierung		
Wert: 123456789 (LONG)		

USING "UD<9><1>	0.0.0.0.0"	123456789
USING "UD<8><1>	0.0.0.0.0"	23456789
USING "UD<7><1>	0.0.0.0.0"	3456789
USING "UD<6><1>	0.0.0.0.0"	456789
USING "UD<5><1>	0.0.0.0.0"	56789
USING "UD<4><1>	0.0.0.0.0"	6789
USING "UD<3><1>	0.0.0.0.0"	789
USING "UD<2><1>	0.0.0.0.0"	89
USING "UD<1><1>	0.0.0.0.0"	9


```

-----
● Ausgabe: unsigned, hex, mit Fuellzeichen, konst. Laenge
  mit Kennung, ohne Vorzeichen, mit Segmentierung
● Wert:      123456789 (LONG)
-----
● USING "UH<8><8>0k 0 2 2 2 2"      |07 5B CD 15H|
● USING "UH<8><8>0k 0 0 0 4 4"      |075B CD15H|
● USING "UH<8><8>0k 0 0 0 0 8"      |075BCD15H|
● USING "UH<8><8>0K 0 2 2 2 2"      |$07 5B CD 15|
● USING "UH<8><8>0K 0 0 0 4 4"      |$075B CD15|
● USING "UH<8><8>0K 0 0 0 0 8"      |$075BCD15|
-----
● Ausgabe: unsigned, hex mit/dez ohne Füllzeichen, var. Laenge
  mit/ohne Kennung/Vorzeichen, ohne Segmentierung
● Wert:      -2748 (LONG)
-----
● USING "UH<8><8>0kV0 0 0 0 8"      |-FFFFF544H|
● USING "UH<8><8>0k 0 0 0 0 8"      |FFFFF544H|
● USING "UD<10><4> V0 0 0 0 0"      |-4294964548|
● USING "UD<10><4> 0 0 0 0 0"      |4294964548|
-----
● Ausgabe: signed, dezimal, ohne Fuellzeichen, konst. Laenge
  mit Kennung, mit Vorzeichen, ohne Segmentierung
● Wert:      -12345789 (LONG)
-----
● USING "SD<9><5> kv0.0.0.0.0"      |123456789#-|
● USING "SD<9><5> kV0.0.0.0.0"      |-123456789#|
● USING "SD<9><5> Kv0.0.0.0.0"      |#123456789-|
● USING "SD<9><5> KV0.0.0.0.0"      |-#123456789|
-----
● Ausgabe: signed, hex mit/dez ohne Fuellzeichen, var. Laenge
  mit/ohne Kennung/Vorzeichen, ohne Segmentierung
● Wert:      -2748 (LONG)
-----
● USING "SH<8><8>0kV0 0 0 0 8"      |-00000ABCH|
● USING "SH<8><8>0k 0 0 0 0 8"      |00000ABCH|
● USING "SD<10><4> V0 0 0 0 0"      |- 2748|
● USING "SD<10><4> 0 0 0 0 0"      | 2748|

```

USING

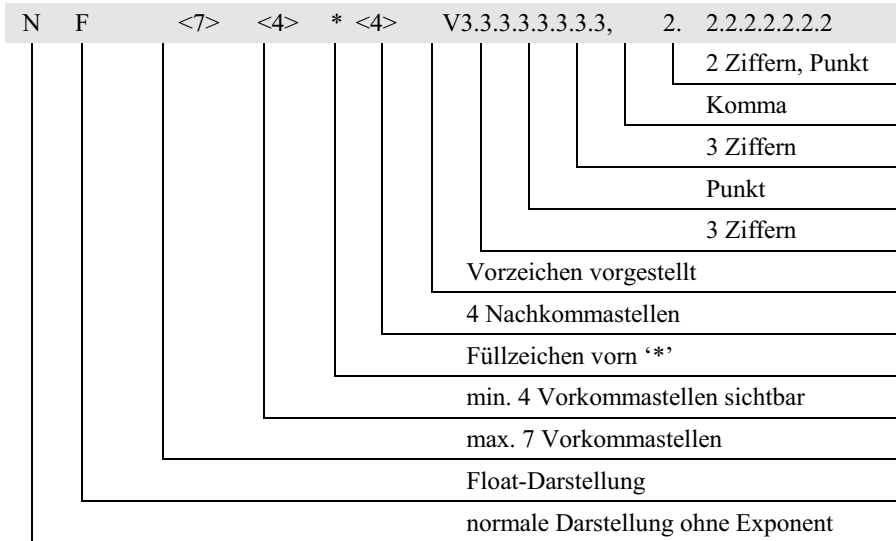
Formatstring für die Ausgabe von REAL-Zahlen:

Pos	Parameter	Bedeutung
0	N	normale, nicht wissenschaftlich
	E	wissenschaftlich, exponentiell
1	F	REAL-Zahl (Floating-Point)
2	0	Vorkomma: variable Länge
	<1>...<31>	Vorkomma: Länge wie angegeben
3	<Zahl>	Vorkomma: mindestens soviel Ziffern
4	ASCII-Zeichen	Auffüllzeichen
5	<0>...<31>	Nachkomma: Anzahl Stellen
	<250>	Nachkomma: variable Länge
6	Leerzeichen	ohne Vorzeichen
	V	‘V’=Vorzeichen vorgestellt, ‘v’=Vorzeichen nachgestellt
7, 8	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
9, 10	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
11, 12	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
13, 14	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
15, 16	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
17, 18	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
19, 20	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
21, 22	<Zahl><Dez.tr.>	Segmentierung: Anz. Z. (0...9), Dezimaltrennzeichen
23, 24	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
25, 26	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
27, 28	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
29, 30	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
31, 32	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
33, 34	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
35, 36	<Zahl><Trenn>	Segmentierung: Anz. Ziffern (0...9), Trennzeichen
37	<Zahl>	Segmentierung: Anz. Ziffern (16x Ziffer, 15x Trennz.)

Beachte: das Füllzeichen ist niemals eine Ziffer. Für führende Nullen erhöhen Sie den Parameter 3 ‘Vorkomma: mindestens soviel Ziffern’.

Die Segmentierung der Ausgabe legt fest, nach wieviel Ziffern welches Trennzeichen eingefügt wird und muß stets vollständig angegeben werden. In dem Formatstring für REAL-Zahlen steht nach den 8 Vorkomma-Segmentierungsziffern das Dezimaltrennzeichen. Von diesem Zeichen ausgehend wird die Zahl vor dem Dezimaltrennzeichen nach links und nach dem Dezimaltrennzeichen nach rechts bearbeitet.

Zur besseren Übersicht ist in der folgenden Abbildung der Formatstring etwas auseinandergezogen. Er enthält in diesem Beispiel jedoch keine Leerzeichen.



3

Die REAL-Zahl '-501,23456' wird mit obigem Formatierungsstring durch den PRINT_USING-Instruktion so ausgegeben:

_*_*_*0.501,23.45.6

Normale (N) Darstellung einer Float-Zahl (F) mit 7 Stellen vor dem Komma, mindestens 4 als Ziffern sichtbar, Füllzeichen (*), 4 Nachkommastellen, Vorzeichen vorgestellt, Trennzeichen vor dem Komma ist nach jeweils 3 Ziffern der Punkt, nach dem Komma dann nach jeweils 2 Ziffern ein Punkt.

Hier finden Sie eine andere Darstellung des Formatstrings und die Zuordnung der Parameter zu zwei konkreten Zahlen '-123456,789012' und '654321,012345':

NF<10><7>*<4>V3.3.3.3.3.3.3.3,2 2 2 2 2 2 2 2

'NF<10><7> und <4>' bedeuten:

Eine **N**ormale **F**loatzahl mit **10** Stellen Vorkommastellen, davon mindestens **7** als Ziffern sichtbar, und **4** Nachkommastellen. So sehen die Zahlen aus, wenn sie mit PRINT_USING ausgegeben werden:

USING

-*0.123.456,78 90#** (-123456,789012)

*****00654321,01 23#** (654321,012345)

Float-Zahlen sind immer Vorzeichenbehaftet und immer Dezimal. Anstelle der Kennung Dezimal/Hexadezimal steht hier die Anzahl der Nachkommastellen.

NF<10><7>***<4>V3.3.3.3.3.3.3.3,2 2 2 2 2 2 2 2**

'*' und 'V' bedeuten:

Das Füllzeichen in obigen Zahlen ist '*', das Vorzeichen ist vorgestellt (großes 'V'). Das positive Vorzeichen erscheint als Leerzeichen.

NF<10><7>***<4>V3.3.3.3.3.3.3.3,2 2 2 2 2 2 2 2**

'3.3.3.3.3.3.3.3,2 2 2 2 2 2 2 2' bedeutet:

Bei der Formatierung wird von dem Dezimaltrennzeichen (das 8. Trennzeichen) ausgehend der Vorkomma-Anteil nach links formatiert, dann der Nachkomma-Anteil ab dem Dezimaltrennzeichen nach rechts entsprechend dem Formatstring mit Trennzeichen versehen: Das Dezimaltrennzeichen ist hier das Komma. Vor dem Komma stehen 3 Ziffern, dann ein Punkt, wieder 3 Ziffern und ein Punkt, usw. Hinter dem Dezimaltrennzeichen stehen jeweils 2 Ziffern, die durch Leerzeichen getrennt sind.

```
-----
Ausgabe: normal, ohne Fuellzeichen, var. Vorkomma,
         var. Nachkomma, ohne Vorzeichen, ohne Segmentierung
Wert:    12345.6789 (REAL)
-----
```

```
USING "NF<1><1> <4> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |56789|
USING "NF<2><1> <4> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |456789|
USING "NF<3><1> <4> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |3456789|
USING "NF<4><1> <4> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |23456789|
USING "NF<5><1> <4> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |123456789|
USING "NF<5><1> <3> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |12345678|
USING "NF<5><1> <2> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |1234567|
USING "NF<5><1> <1> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |123456|
USING "NF<5><1> <0> 0.0.0.0.0.0.0.0,0.0.0.0.0.0.0.0" |12345|
```

```
-----
Ausgabe: normal, ohne Fuellzeichen, var. Vorkomma,
         var. Nachkomma, ohne Vorzeichen, mit Segmentierung
Wert:    12345.6789 (REAL)
-----
```

```
USING "NF<1><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |5,6789|
USING "NF<2><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |45,6789|
USING "NF<3><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |345,6789|
USING "NF<4><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |2.345,6789|
USING "NF<5><1> <4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,6789|
USING "NF<5><1> <3> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,678|
USING "NF<5><1> <2> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,67|
USING "NF<5><1> <1> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,6|
USING "NF<5><1> <0> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345|
```

USING

3

```

-----
Ausgabe: normal, mit Fuellzeichen, var. Vorkomma,
        konst. Nachkomma, ohne Vorzeichen, mit Segmentierung
Wert:    12345.6789 (REAL)
-----
USING "NF<5><1>0<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,6789|
USING "NF<6><1>0<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |012.345,6789|
USING "NF<7><1>0<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |0.012.345,6789|
USING "NF<8><1>0<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |00.012.345,6789|
USING "NF<9><1>0<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |000.012.345,6789|
-----

Ausgabe: normal, mit Fuellzeichen (<>0), var. Vorkomma,
        var. Nachkomma, ohne Vorzeichen, mit Segmentierung
Wert:    12345.6789 (REAL)
-----
USING "NF<5><1>=<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |12.345,6789|
USING "NF<6><1>=<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |=12.345,6789|
USING "NF<7><1>=<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |===12.345,6789|
USING "NF<8><1>=<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |====12.345,6789|
USING "NF<9><1>=<4> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0.0" |=====12.345,6789|
-----

Ausgabe: exp., ohne Fuellzeichen, konst. Vorkomma,
        var. Nachkomma, ohne Vorzeichen, mit Segmentierung
Wert:    12345.6789 (REAL)
-----
USING "EF<1><1> <1> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,2E+004|
USING "EF<1><1> <2> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,23E+004|
USING "EF<1><1> <3> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,234E+004|
USING "EF<1><1> <4> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,2345E+004|
USING "EF<1><1> <5> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,23456E+004|
USING "EF<1><1> <6> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,234567E+004|
USING "EF<1><1> <7> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,2345678E+004|
USING "EF<1><1> <8> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0.0" |1,23456789E+004|

```

```
-----
Ausgabe: exp./normal, ohne Fuellzeichen, var. Vorkomma,
         var. Nachkomma, mit/ohne Vorzeichen/Segmentierung
Wert:    -12345.6789 (REAL)
-----
```

```
USING "NF<5><1> <1>v0.0.0.0.0.0.3.3,4.0.0.0.0.0.0" |12.345,6789-|
USING "NF<5><1> <1>V0.0.0.0.0.0.3.3,4.0.0.0.0.0.0" |-12.345,6789|
USING "NF<5><1> <1> 0.0.0.0.0.0.3.3,4.0.0.0.0.0.0" |12.345,6789|
USING "EF<1><1> <8>v0.0.0.0.0.0.0.1,8.0.0.0.0.0.0" |1,23456789-E+004|
USING "EF<1><1> <8>V0.0.0.0.0.0.0.1,8.0.0.0.0.0.0" |-1,23456789E+004|
USING "EF<1><1> <8> 0.0.0.0.0.0.0.1,8.0.0.0.0.0.0" |1,23456789E+004|
```

VAL_NUM

VAL_NUM

VAL_NUM (String)

VAL_NUM: Lies Integer Wert aus einem String

Funktion: Liefert den Wert der ersten Ziffernfolge aus der Zeichenfolge **String** als BYTE, WORD oder LONG-Zahl zurück. VAL_NUM erkennt auch Hexadezimalzahlen, die mit '\$' beginnen.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die Zahl. Vor oder hinter der Zahl dürfen andere Zeichen stehen. Das '\$'-Zeichen beeinflusst VAL_NUM.

Funktionswert ist vom Typ LONG. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Der String darf sowohl vor als auch hinter der Zahl weitere Zeichen enthalten. Sind mehrere Zahlen im String enthalten, so wird die erste als Funktionswert zurückgeliefert. Das Zeichen '\$' leitet eine hexadezimale Zahl ein. Ein '\$'-Zeichen ohne folgende Ziffern wird als Wert 0 zurückgeliefert.

Bei der Verarbeitung mit VAL_NUM störende Zeichen lassen sich bequem mit der Funktion CONVERT\$ entfernen.

Tip

Beispiele:

```
X = VALNUM ( " 12 13" )      ` Wert ist 12
X = VALNUM ( "xyz 12 DM" )    ` Wert ist 12
X = VALNUM ( "$0C" )          ` Wert ist 12
X = VALNUM ( "US$12" )        ` Wert ist 12Hex = 18
X = VALNUM ( "US$ 12" )       ` Wert ist 0
```


Programmbeispiel:

```
'-----  
' Name: VAL_NUM.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, 435 + VAL_NUM ("= 120 mm") ' Ausgabe auf LC-Display  
END                                     ' Ende Task MAIN
```

3

Siehe auch: VAL_REAL, ASC, LEN, STR\$, STRIS

VAL_REAL

VAL_REAL

VAL_REAL (*String*, *Deztrenn*)

VAL_REAL: Lies Real Wert aus einem String

Funktion: Liefert den Wert der ersten Ziffernfolge in der Zeichenfolge **String** zurück.

3

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING.

Deztrenn ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING mit der Länge 1 und gibt das Dezimaltrennzeichen an.

Funktionswert ist vom Typ REAL. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Beispiele:

```
X = VAL_REAL ( " 12 13", "." )      ` Wert ist 12.0
X = VAL_REAL ( "xyz 12 DM", "." )    ` Wert ist 12.0
X = VAL_REAL ( "xyz 12.50 DM", "." )  ` Wert ist 12.5
x = VAL_REAL ( "US$12.00 DM", "." )  ` Wert ist 12.0
x = VAL_REAL ( "12,50 DM", "," )     ` Wert ist 12.5
x = VAL_REAL ( "1.234E4", "." )      ` Wert ist 12340
```

Programmbeispiel:

```
' -----
' Name: VAL_REAL.TIG
' -----

TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
PRINT #1, VAL_REAL ("123.45 DM", ".") ' "." ist Dezimal-Trenner !
END                                     ' Ende Task MAIN
```

Siehe auch: VAL_NUM, ASC, LEN

WAIT_CLOCK

WAIT_CLOCK [Wert]

WAIT_CLOCK: Ausführung dieser Task wartet bis Ticks Wert erreicht ist

Instruktion: Wartet, bis die Systemticks à 1 Millisekunde den Stand **Wert** erreicht haben. Die CPU-Zeit dieser Task wird den anderen Tasks solange zur Verfügung gestellt.

Wert ist eine Zahl vom Typ BYTE, WORD oder LONG.

WAIT_CLOCK wartet, bis der angegebene Zeitpunkt erreicht ist. Der Systemtick-Zähler beginnt nach Reset mit 0 und läuft nach 2.147.483.647 Ticks, also nach 24 Tagen über. Der Zähler bleibt im positiven Wertebereich der LONG-Zahlen. Der Zählstand der Systemticks kann mit der Funktion Ticks ausgelesen werden.

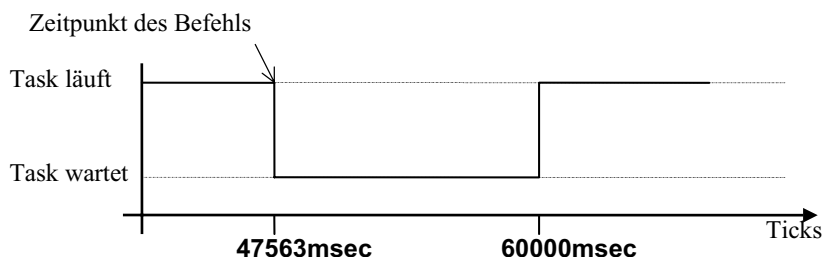


Beachte: die Instruktionen WAIT_CLOCK, WAIT_DURATION und WAIT_NEXT müssen im zeitlichen Programmablauf eine Chance haben, innerhalb der gesetzten Zeit im Multitaskingsystem wieder die CPU zugeteilt zu bekommen, um ordnungsgemäß zu funktionieren.

Das heißt zum Beispiel, daß eine Task mit kurzen WAIT_DURATION von z.B. 100msec nicht für eine Sekunde oder mehr gestoppt werden darf. Wenn eine Task länger gestoppt werden soll, dann werden Wartezeiten mit Ticks und DIFFTicks programmiert.

Beispiel:

WAIT_CLOCK 60000



WAIT_CLOCK

Programmbeispiel:

```
'-----
' Name: WAIT_CLOCK.TIG
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG L                                ' LONG-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  L = TICKS ()                          ' Systemticks lesen
  PRINT #1, "Hello at"; L                ' Ausgabe auf LC-Display
  WAIT_CLOCK (L+2000)                   ' Warte bis Systemticks+2000
  L = TICKS ()                          ' Systemticks lesen
  PRINT #1, "Hello at"; L                ' Ausgabe auf LC-Display
END                                      ' Ende Task MAIN
```

Siehe auch: WAIT_NEXT, WAIT_DURATION

WAIT_DURATION

WAIT_DURATION Anzahl

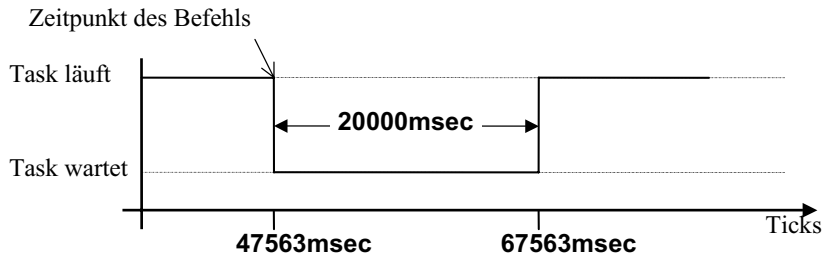
WAIT_DURATION: Ausführung dieser Task wartet Zeit-DAUER

Instruktion: Wartet **Anzahl** Millisekunden. Der folgende Befehl in dieser Task wird nach der angegebenen Wartezeit ausgeführt. Die CPU-Zeit dieser Task wird den anderen Tasks solange zur Verfügung gestellt.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Millisekunden an, die gewartet werden soll.
Wertebereich: 3 bis zur größten positiven LONG-Zahl (2.147.483.647).

WAIT_DURATION stellt die CPU-Zeit anderen Tasks zur Verfügung.

WAIT_DURATION 20000



WAIT_DURATION

Beachte: die Instruktionen WAIT_CLOCK, WAIT_DURATION und WAIT_NEXT müssen im zeitlichen Programmablauf eine Chance haben, innerhalb der gesetzten Zeit im Multitaskingsystem wieder die CPU zugeteilt zu bekommen, um ordnungsgemäß zu funktionieren.

Das heißt zum Beispiel, daß eine Task mit kurzen WAIT_DURATION von z.B. 100msec nicht für eine Sekunde oder mehr gestoppt werden darf. Wenn eine Task länger gestoppt werden soll, dann werden Wartezeiten mit TICKS und DIFFTICKS programmiert.



3

Programmbeispiel:

```
'-----  
' Name: WAIT_DURATION.TIG  
'-----  
TASK MAIN                                ' Beginn Task MAIN  
' LCD-Treiber installieren (BASIC-Tiger)  
  INSTALL DEVICE #1, "LCD1.TDD"  
' LCD-Treiber installieren (TINY-Tiger)  
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8  
  PRINT #1, "<l>Output in 5 Sec."          ' Ausgabe auf LC-Display  
  WAIT DURATION 5000                     ' 5 Sek warten  
  PRINT #1, "5 Sec. counted down"        ' Ausgabe auf LC-Display  
END                                       ' Ende Task MAIN
```

Siehe auch: WAIT_NEXT, WAIT_CLOCK

WAIT_NEXT

WAIT_NEXT [Wert]

WAIT_NEXT: Ausführung dieser Task wartet Zeit-RASTER

Instruktion: Wenn **Wert** angegeben ist, wird die Länge der Zeitscheibe in Millisekunden gesetzt. Wenn **Wert** nicht angegeben ist, wartet WAIT_NEXT bis zum Ende der aktuellen Zeitscheibe. Während der Wartezeit wird die CPU-Zeit dieser Task den anderen Tasks zur Verfügung gestellt.

Wert ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Anzahl der Millisekunden an, die einer Zeitscheibe entsprechen.
Wertebereich: 3 bis zur größten positiven LONG-Zahl (2.147.483.647).

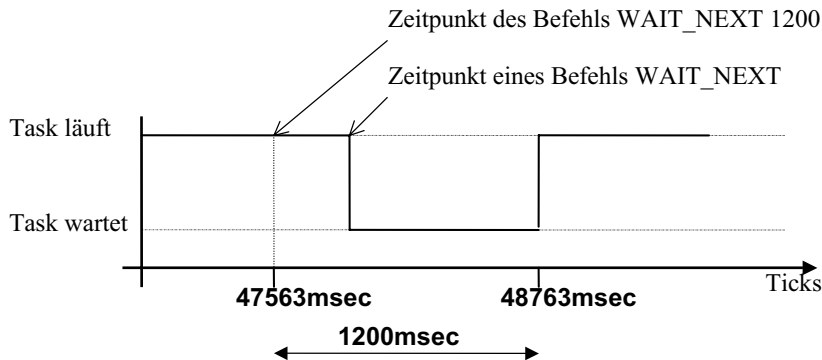
Mit der Instruktion WAIT_NEXT kann eine Task exakt auf ein Zeitraster synchronisiert werden. Zunächst wird **Wert** angegeben und das Zeitraster damit gesetzt. Weitere WAIT_NEXT-Instruktionen warten, bis erneut eine Zeitrastergrenze erreicht ist.



Beachte: die Instruktionen WAIT_CLOCK, WAIT_DURATION und WAIT_NEXT müssen im zeitlichen Programmablauf eine Chance haben, innerhalb der gesetzten Zeit im Multitaskingsystem wieder die CPU zugeteilt zu bekommen, um ordnungsgemäß zu funktionieren.

Das heißt zum Beispiel, daß eine Task mit kurzen WAIT_DURATION von z.B. 100msec nicht für eine Sekunde oder mehr gestoppt werden darf. Wenn eine Task länger gestoppt werden soll, dann werden Wartezeiten mit TICKS und DIFFTICKS programmiert.

WAIT_NEXT 1200



WAIT_NEXT

Programmbeispiel:

```
-----
' Name: WAIT_NEXT.TIG
-----
' Der Zaehler Time1 laeuft etwas schneller als der Zaehler Time2,
' da die Wartezeit in Task T1 etwas kuerzer ist als die in Task T2.
-----
LONG Time1,Time2                                ' Globale LONG-Variablen

TASK MAIN                                        ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
Time1 = 0                                        ' Time1 initialisieren
Time2 = 0                                        ' Time2 initialisieren
RUN_TASK T1                                    ' Task T1 starten
RUN_TASK T2                                    ' Task T2 starten
LOOP 99999999                                  ' sehr viele Loops
    PRINT #1, "<1>Time1 ="; Time1                ' Time1 auf Display ausgeben
    PRINT #1, "Time2 ="; Time2                  ' Time2 auf Display ausgeben
    WAIT_DURATION 100                           ' 100 ms warten
ENDLOOP
END                                              ' Ende Task MAIN

-----
' Task T1 inkrementiert Time1, wenn naechste 100 ms erreicht sind
-----
TASK T1                                        ' Beginn Task T1
WAIT_NEXT 100                                ' 100ms warten initialisieren
LOOP 99999999                                ' sehr viele Loops
    WAIT_NEXT                                  ' Rest der 100 ms
    Time1 = Time1 + 1                          ' Time1 inkrementieren
ENDLOOP
END                                              ' Ende Task T1

-----
' Task T2 inkrementiert Time2 alle 100 ms
-----
TASK T2                                        ' Beginn Task T2
LOOP 99999999                                ' sehr viele Loops
    WAIT_DURATION 100                          ' 100 ms warten
    Time2= Time2 + 1                           ' Time2 inkrementieren
ENDLOOP
END                                              ' Ende Task T2
```

Siehe auch: WAIT_CLOCK, WAIT_DURATION

WHILE

WHILE [Bedingung]

[Anweisungsblock]

ENDWHILE

WHILE...ENDWHILE: Schleife mit Abbruchbedingung am Anfang

Instruktion: Führt den Anweisungsblock solange aus, wie die **Bedingung** erfüllt ist. Der Anweisungsblock ist mit ENDWHILE abgeschlossen. Wenn die **Bedingung** bereits am Anfang nicht erfüllt ist, wird der Anweisungsblock kein einziges Mal ausgeführt.

Bedingung besteht aus mindestens einer Vergleichs-Operation, die als wahr oder falsch ausgewertet werden kann. Mehrere Bedingungen können durch AND, OR, XOR oder NOT miteinander verknüpft werden (Siehe Operatoren Seite 34).

Die Schleife mit WHILE bietet die Möglichkeiten, den Abbruch der Schleife durch komplizierte Bedingungen zu definieren. Schleifen, die von dieser Möglichkeit keinen Gebrauch machen, laufen schneller ab, wenn Sie mit FOR...NEXT oder LOOP realisiert sind.

Vergleichsoperatoren	
=	gleich
<>	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich

Beispiele:

WHILE A=0	Solange A=0 ist.
WHILE A=1 AND B=0	Solange A=1 und B=0 ist.
WHILE NOT (A=1)	Solange A nicht gleich 1 ist.
WHILE NOT ((A=1) AND (B=0))	Solange die Bedingungen 'A=1' und gleichzeitig 'B=0' nicht erfüllt sind.
WHILE NOT ((A=1)) AND (B=0))	Solange A nicht gleich 1, aber B=0 ist.

WHILE

Folgende Konstruktion mit WHILE ist programmtechnisch richtig, hat aber im Multitaskingsystem den Nachteil, entsprechend der Priorität ständig CPU-Zeit zu benötigen. Der Tastaturpuffer wird viel zu oft auf das Vorhandensein von Zeichen überprüft:

```
A$=""  
WHILE A$=""  
    GET #1, 1, A$           ' auf Tastatureingabe warten  
ENDWHILE  
    ' Eingabe verarbeiten
```

3

Besser ist die folgende Konstruktion, die durch die Instruktion WAIT_DURATION CPU-Zeit wieder freigibt:

```
A$=""  
WHILE A$=""  
    GET #1, 1, A$           ' auf Tastatureingabe warten  
    WAIT_DURATION 50        ' gebe CPU-Zeit frei  
ENDWHILE  
    ' Eingabe verarbeiten
```

Programmbeispiel:

```

'-----
' Name: WHILE.TIG
'-----
#INCLUDE KEYB_PP.INC           ' deutsche Keyboard-Belegung

TASK MAIN                     ' Beginn Task MAIN
  STRING A$                   ' STRING-Variable deklarieren
  BYTE N                      ' BYTE-Variable deklarieren
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 80h, 8
  CALL INIT_KEYB (1)          ' setze Keyboard + div. Params
  A$ = ""                     ' A$ initialisieren
  PRINT #1, "<l>press ESC"      ' Ausgabe auf LC-Display
  WHILE A$ <> CHR$(27)          ' Solange A$ nicht "ESC"
    FOR N = 0 TO 0 STEP 0      ' Endlosschleife bis N=1(GET!)
      RELEASE_TASK            ' Rest der Task-Zeit freigeben
      GET #1, #0, #1, 1, N     ' N=Zeichen im Tastatur-Buffer
    NEXT                       ' Ende Endlosschleife
    GET #1, 1, A$              ' Aus Tastaturbuffer einlesen
  ENDWHILE                    ' Ende der While-Schleife
  PRINT #1, "ESC pressed"      ' Ausgabe auf LC-Display
END                             ' Ende Task MAIN

```

3

Siehe auch: LOOP, IF...THEN, FOR...NEXT

XBUS_INR\$

XBUS_INR\$

XBUS_INR\$ (*Anzahl*, *Clock_Flag*)

XBUS_INR\$: Schnelles Einlesen von Daten über XPort-Bus
(nur in Projektmodell 'PM_FULL')

Funktion: Liest Bytes von dem Datenbus des externen Portsystems und erzeugt lediglich ACLK-Signale.

3

- Anzahl** ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und gibt die Anzahl der zu lesenden Bytes an.
- Clock_Flag** ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und gibt an, ob vor dem ersten Clock oder nach dem ersten Clock das erste Mal gelesen wird:
0: erst ein Byte einlesen, dann den ersten Clock
1: erst den ersten Clock, dann das erste Byte einlesen.
- Funktionswert** ist vom Typ STRING und enthält die gelesenen Bitmuster.

Diese schnelle Art des Datenlesens kann z.B. mit einem externen, parallel angeschlossenen Chip kommunizieren, z.B. einem Speicherchip, A/D-Wandler, ASIC.

XBUS_INR\$ liest Bytes von dem Datenbus des externen Portsystems und erzeugt lediglich ACLK-Signale. Auf diese Weise kann auf dem Daten-Adreßbus des erweiterten Portsystems ein externes Gerät angeschlossen sein, welches auf einem parallelen 8-Bit-Bus Daten getaktet zur Verfügung stellt. Zugriffe auf den Bus von Device-Treibern werden unterbunden, solange das CE-Signal aktiv ist. Das BASIC-Tiger[®]-EPORT-System kann parallel dazu weiterverwendet werden.

Es werden soviel Bytes gelesen, wie angegeben, jedoch maximal so viele, wie der Ergebnisstring aufnehmen kann.

Programmbeispiel:

```

' -----
' Name: XBUS_INR$.TIG
' Direktes Einlesen vom Daten-Adressbus des erweiterten Portsystems
' Nur ACLK-Signale werden erzeugt.
' Beispiel verwendet die Adressen der Tastatur des Plug & Play Labs
' -----
TASK MAIN                                ' Beginn Task MAIN
  STRING A$(16k)                          ' grosser String
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  ' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  A$ = XBUS_INR$ ( 0, 1k )                '
END                                        ' Ende Task MAIN

' -----
' zeigt den String als HEX-Bytes
' -----
SUB SHOW_STRING_AS_HEX ( STRING S$; LONG LENGTH )
  LONG I
  USING "UH<2><2> 0.0.0.0.2"              ' Format HEX-Ausgabe
  FOR I = 0 TO LENGTH                     ' Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) ); ' als HEX
  NEXT
  PRINT #1, ""                            ' neue Zeile
END

```

3

Siehe auch: XBUS_OUTR, XIN\$, XOUT, XSETUP

XBUS_OUTR

XBUS_OUTR (Var [, Länge])

XBUS_OUTR: Schnelles Ausgeben von Daten über XPort-Bus
(nur in Projektmodell 'PM_FULL')

Funktion: Gibt Byte(s) aus **Var** auf dem Port des erweiterten Portsystems aus und erzeugt dabei ACLK- und CE-Signale.

3

Var

ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG, ein STRING oder eine Flash-Adresse (DATALABEL).

BYTE, WORD, LONG: es werden alle Bytes des numerischen Wertes auf den Port geschrieben.

STRING: es werden die Bytes des STRINGS auf den Port geschrieben.

DATALABEL: es werden die Bytes ab der Flash-Adresse auf den Port geschrieben. In diesem Fall ist die Angabe der Länge erforderlich.

Länge

ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und ist nur anzugeben, wenn Var eine Flash-Adresse ist.

Diese Funktion liefert keinen Funktionswert zurück.

Diese schnelle Art der Datenausgabe kann z.B. mit einem externen, parallel angeschlossenen Chip kommunizieren, z.B. einem Speicherchip, A/D-Wandler, ASIC.

XBUS_OUTR schreibt Bytes auf den Datenbus des externen Portsystems und erzeugt ACLK-Signale und ein 'Chip-enable'-Signal auf dem Pin, der mit XSETUP für die CE-Funktion bestimmt wurde (Standard: L40). Auf diese Weise kann auf dem Adreß-Datenbus des erweiterten Portsystems ein externes Gerät angeschlossen sein, welches auf einem parallelen 8-Bit-Bus Daten getaktet entgegennimmt. Zugriffe auf den Bus von Device-Treibern werden nicht unterbunden. Daher muß das Chip-enable-Signal herangezogen werden, um das gewünschte externe Gerät nur dann mit ACLK und Daten zu versorgen, wenn es auch beabsichtigt ist. Das BASIC-Tiger®-EPORT-System kann parallel dazu weiterverwendet werden.

Programmbeispiel:

```

'-----
' Name: XBUS_OUTR.TIG
' Direktes Ausgeben auf dem Daten-Adressbus (erweiterten Portsystems)
' Data, ACLK-Signale und Chip-enable werden erzeugt.
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

OUT 4, 00000001b, 255                    ' CE-Pin idle=high
DIR_PIN 4, 0, 0                          ' setze CE-Pin als Ausgang

R = XSETUP ( &
    6, &                                ' Daten-Port
    3, &                                ' Port fuer ACLK, DCLK, INE
    3, &                                ' Pin-Nr. fuer ACLK
    4, &                                ' Pin-Nr. fuer DCLK
    5, &                                ' Pin-Nr. fuer INE
    4, &                                ' Port fuer CE
    0 )                                  ' Pin-Nr. fuer CE
IF R <> 0 THEN                            ' wenn XSET nicht erfolgreich
    GOTO ERROR                            ' melde Fehler
ENDIF

A$ = "this is a test for<10><13> XBUS_OUTR"
XBUS_OUTR ( A$ )                          ' ausgeben auf Daten/Adressbus
                                           ' (L60...L67), ACLK (L33), CE(L40)

ERROR:
    PRINT #1, "error occured"
END                                         ' Ende Task MAIN

```

3

Siehe auch: XBUS_INR\$, XIN\$, XOUT, XSETUP

XIN, XIN\$

XIN, XIN\$

XIN (*Phys_Portadr*)

XIN\$ (*Phys_Portadr*, *Anzahl*)

XIN, XIN\$: Liest Byte(s) von XPort I/O-Erweiterungs-System
(nur in Projektmodell 'PM_FULL')

3

Funktion: Liefert ab der Adresse **Phys_Portadr** mit aufsteigenden Adressen die momentan anliegenden 8-Bit-Werte von den externen Ports.

Phys_Portadr ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die physikalische Portadresse an.

Anzahl ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt bei XIN\$ die Anzahl der zu lesenden Ports an.

Funktionswert ist im Fall von XIN vom Typ BYTE, WORD, oder LONG und im Fall von XIN\$ vom Typ STRING und enthält die von den Ports gelesenen Bitmuster.

XIN und XIN\$ sind 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber Lesen mit der IN-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: XIN und XIN\$ prüfen nicht die Zulässigkeit des Portzugriffs.

Die Trennung in Adreßbereiche für Output und Input gilt für XIN und XIN\$ nicht. Es können sich Input-Adressen und Output-Adressen im gesamten Adreßraum von 0 bis 255 befinden und auch beliebig überlagern. XIN liest für WORD 2 Ports und für LONG 4 Ports.

Programmbeispiel:

```

'-----
' Name: XIN.TIG
' Direktes Einkesen von phys. Adressen der erweiterten Ports
' Beispiel verwendet die Adressen der Tastatur des Plug & Play Labs
' Das Byte entspricht dem oberen DIP-Schalter
'-----
TASK MAIN                                ' Beginn Task MAIN
  LONG L
  ' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
  ' LCD-Treiber installieren (TINY-Tiger)
  '  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  LOOP 9999999                          ' viele loops
    L = XIN ( 97h )                      ' liest von 97h 1 Byte
    USING "UH<8><8> 0.0.0.4.4"           ' Format HEX-Ausgabe
    PRINT USING #1, L
    WAIT_DURATION 200                    ' 200ms warten
  ENDLOOP
END                                      ' Ende Task MAIN

```

3

XIN, XIN\$

Programmbeispiel:

3

```
'-----
' Name: XIN$.TIG
' direktes Einkesen von phys. Adressen der erweiterten Ports
' Beispiel verwendet die Adressen der Tastatur des Plug & Play Labs
'-----
TASK MAIN                                ' Beginn Task MAIN
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  LOOP 9999999                          ' viele loops
    A$ = XIN$ ( 88h, 16 )                ' Erw. Inputs 88h..97h
    PRINT #1, "<1>";                      ' loesche LCD
    CALL SHOW_STRING_AS_HEX ( A$ )       ' zeige String an
    WAIT_DURATION 200                    ' 200ms warten
  ENDLOOP                                '
END                                      ' Ende Task MAIN

'-----
' zeigt den String als HEX-Bytes
'-----
SUB SHOW_STRING_AS_HEX ( STRING S$ )
  BYTE I
  USING "UH<2><2> 0.0.0.0.2"              ' Format HEX-Ausgabe
  FOR I = 0 TO LEN ( S$ ) - 1            ' alle Bytes des Strings
    PRINT_USING #1, ASC ( MID$ ( S$, I, 1 ) );' als HEX
  NEXT
  PRINT #1, ""                           ' neue Zeile
END
```

Siehe auch: XBUS_INR\$, XBUS_OUTR, XOUT, XSETUP

XOR\$

XOR\$ (*String*, *Bitmuster*)

XOR\$: Verknüpfe Wert mit allen Bytes eines Strings per XOR
(nur in Projektmodell 'PM_FULL')

Funktion: Führt mit allen Bytes in **String** eine XOR-Operation aus.

String ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält die zu bearbeitenden Zeichen.

Bitmuster ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und enthält das 8-Bit breite Bitmuster für die XOR-Operation.

Funktionswert ist vom Typ STRING und enthält den konvertierten String.

Die Zeichen in **String** werden byteweise mit einer XOR-Operation bearbeitet. Dabei werden alle Bits, die im Quellstring mit den Bits im Bitmuster übereinstimmen, im Zielbyte zu '0'. Bits, die im Quellstring nicht mit den Bits im Bitmuster übereinstimmen, werden im Zielbyte zu '1'.

Beispiel:

Quellstring (2 Bytes):	01100111	01100110
Bitmuster:	01010101	01010101
Ergebnis-String:	00110010	00110011

XOR\$

Programmbeispiel:

```
' -----
' Name: XOR$.TIG
' -----

TASK MAIN                                ' Beginn Task MAIN
  STRING A$                              ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
  INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
'  INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  A$ = "<67H><66H>"                        ' das is A$
  PRINT #1, "0:";                         ' zeige A$
  CALL DISP_BIN ( 2, A$ )                 ' binaer
  A$ = XOR$ ( A$, 55H )                   ' A$ XOR 55H
  PRINT #1, "X:";
  CALL DISP_BIN ( 2, "<55H><55H>" )         ' zeige AND-Maske
  PRINT #1, "1:";                         ' zeige Ergebnis
  CALL DISP_BIN ( 2, A$ )                 ' binaer
END                                        ' Ende Task MAIN

SUB DISP_BIN ( CNT; BYTES$ )              ' zeige CNT Bytes binaer
  BYTE B, I, J                            ' lokale Variable
  FOR I = 1 TO CNT                        ' soviele Bytes
    B = ASC ( MID$ ( BYTES$, I-1, 1 ) )   ' hole ein Byte
    FOR J = 1 TO 8                        ' zeige 8 Bits
      IF BIT ( B, 7 ) = 1 THEN             ' teste ob bit 1 ist
        PRINT #1, "1";                    ' drucke Bit
      ELSE                                 ' falls kein Rest
        PRINT #1, "0";                    ' drucke Bit
      ENDIF
      B = B SHL 1                          ' B eins nach rechts (neues Bit)
    NEXT J
    PRINT #1, " ";                         ' Leerzeichen zwischen Bytes
  NEXT I
END                                        ' Ende Unterprogramm
```

Siehe auch: ADD\$, AND\$, BIT_MIRR\$, OR\$, Operator BIT_XOR

XOUT

XOUT (Phys_Portadr., Var [, Anzahl])

XOUT: Schreibt Byte(s) auf XPort I/O-Erweiterungs-System
(nur in Projektmodell 'PM_FULL')

Funktion: Gibt Byte(s) aus **Var** an dem externen Port mit der Adresse **Phys_Portadr.** aus.

- Phys_Portadr.** ist eine Konstante vom Typ BYTE, WORD oder LONG und gibt die physikalische Portadresse an.
- Var** ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG, ein STRING oder eine Flash-Adresse (DATALABEL).
BYTE, WORD, LONG: es werden die untersten 8 Bit des numerischen Wertes auf den Port geschrieben.
STRING: es werden die Bytes des STRINGS auf den Port geschrieben.
DATALABEL: es werden die Bytes ab der Flash-Adresse auf den Port geschrieben.
- Anzahl** ist eine Variable, eine Konstante oder ein Ausdruck vom Datentypen BYTE, WORD oder LONG und ist nur anzugeben, wenn Var eine Flash-Adresse ist.

3

Diese Funktion liefert keinen Funktionswert zurück.

XOUT ist eine 'low-Level'-Instruktion. Die Ausführungsgeschwindigkeit ist gegenüber der Portausgabe mit der OUT-Instruktion höher, da weniger Verwaltungsaufwand zur Laufzeit betrieben wird. Beachte: XOUT prüft nicht die Zulässigkeit des Portzugriffs.

Die Trennung in Adreßbereiche für Output und Input gilt für XOUT nicht. Es können sich Input-Adressen und Output-Adressen im gesamten Adreßraum von 0 bis 255 befinden und auch beliebig überlagern.

Ist das Argument vom Typ BYTE, WORD oder LONG, werden nur die untersten 8 Bit geschrieben. Die zusätzlichen Bytes in WORD oder LONG bleiben unberücksichtigt.

Ist das Argument ein STRING, werden alle Bytes des Strings nacheinander in aufsteigender Adreßfolge geschrieben.

Ist das Argument eine Flash-Adresse, ist zusätzlich eine Längenangabe erforderlich. Es werden soviel Bytes aus dem User-Flashbereich geschrieben, wie angegeben.

XOUT

Programmbeispiel:

```
-----
' Name: XOUT.TIG
' direkte Ausgabe auf phys. Adressen der erweiterten Ports
' aus Flash, direkt, aus String
-----
TASK MAIN                                ' Beginn Task MAIN
  A$ = "01 02 04 08"%                     ' Bytes fuer Ports
  '                                     '
  '      adr,      src,      offs,len,mode
  R = POKEM_FLASH (0, "AA 55 F0 0F"% , 0,  4, 2)
  '                                     ' 3. Parameter: aus Flash
  XOUT ( 8, 0, 4 )                        ' phys. ports 8-11 aus Flash

  LOOP 10                                ' 10 mal
    XOUT ( 12, 0 )                        ' Erw. I/O Port (phys.) 12 low
    WAIT_DURATION 200                     ' 200ms warten
    XOUT ( 12, 0AAh )                     ' Erw. Port (phys.) 8 Muster
    WAIT_DURATION 200                     ' 200ms warten
    XOUT ( 12, 55h )                       ' Erw. Port (phys.) 8 Muster
    WAIT_DURATION 200                     ' 200ms warten
  ENDLOOP

  XOUT ( 8, A$ )                          ' phys. ports 8-11 aus String
END                                         ' Ende Task MAIN
```

Siehe auch: XBUS_INR\$, XBUS_OUTR, XIN\$, XSETUP

XSETUP

XSETUP (P1, P2, P3, P4, P5)

XSETUP: Setzt Einstellung für XPort I/O-Erweiterungs-System
(nur in Projektmodell 'PM_FULL')

Funktion: Setzt die Portadressen und Bitnummern für das erweiterte I/O-System.

P1...P5 geben die Parameter des erweiterten I/O-Systems wie folgt an:

	Nach Power-On oder Reset	Beschreibung
BUS_PORT	Port 6	Port, der als 8-Bit Daten-Adreßbus verwendet wird: 6 oder 8
CTRL_PORT	Port 3	Port, der als 3-Bit Steuerbus verwendet wird
BIT_ACLK	L33	Bit-Nr: 0...7 für ACLK: Adreß-Clock
BIT_DCLK	L34	Bit-Nr: 0...7 für DCLK: Daten-Clock
BIT_INE	L35	Bit-Nr: 0...7 für -INE: Input-Enable (low active)
CTRL2_PORT	Port 4	Port, der für CE-Signal verwendet wird
BIT_BUS_CE	L40	Bit-Nr: 0...7 für CE: Chip-Enable Pegel wird vorher gesetzt, Pin wird nicht geschützt.

Funktionswert 0 = OK
 1...5 = fehlerhafter Parameter

Alle folgenden XPORT-Funktionen arbeiten mit den hier eingestellten Parametern.

Die XPORT-Funktionen bedienen das erweiterte Portsystem des BASIC-Tiger® und bieten dabei folgende Vorteile:

- ist viel schneller als IN und OUT auf erweiterte Ports.
- Logische Adresse = physikalische Adresse.
- keine Unterteilung des Adreßraums in Adressen für Outputs und Adressen für Inputs.
- maximaler Adreßraum: 256 Input-Ports und 256 Output-Ports, d.h. bis zu 4096 Pins.
- alle Einstellungen mit einem Befehl XSETUP.
- zusätzliche interessante Funktionen:

XSETUP

3

Funktion	Beschreibung
XIN	liest 1 Byte ein.
XIN\$	liest viele Bytes in aufsteigender Adreßfolge ein.
XINR\$	liest viele Bytes ein und erzeugt gleichzeitig ein Adress-Clock-Signal.
XOUT	gibt viele Bytes in aufsteigender Adreßfolge aus, wenn das Argument ein String ist. Sonst: 1 Byte
XOUTR	gibt viele Bytes aus und erzeugt gleichzeitig ein Adress-Clock-Signal.

Die Funktion und Verwendung der Pins ACLK, DCLK und INE ist im Hardware-Handbuch unter ‚Erweitertes I/O-System‘ beschrieben. Neu ist in diesem Zusammenhang der CE-Pin. Dieser Pin wird vor seiner Verwendung als Ausgang gesetzt und auf einen Ruhepegel. Die Funktionen XBUS_INR\$ und XBUS_OUTR toggeln diesen Pin während sie die ACLK-Signale erzeugen. Auf diese Weise erhält man eine Chip-Enable-Funktion, die gezielt nur die Adreß-Clocks zu dem externen Chip durchläßt, die von den XBUS-Funktionen erzeugt wurden. ACLK-Signale von Device-Treibern dringen so jedoch nicht zu dem externen Chip durch.

Programmbeispiel:

```

'-----
' Name: XSETUP.TIG
' Setzt Busport und Steuerleitungen des erweiterten Portsystems
' hier: Standardwerte, damit es auf dem Plug & Play Lab funktioniert
'-----
TASK MAIN                                ' Beginn Task MAIN
  DIR_PIN 4, 0, 0                        ' Pin L40 als Ausgang
  OUT 4,00000001b, 255                   ' Pin L40 high
  '          bus, ctrl, ACLK, DCLK, INE, ctrl2, CE
  FLAG = XSETUP ( 6, 3, 3, 4, 5, 8, 1 )
  IF FLAG <> 0 THEN                       ' wenn Parameterfehler
    GOTO ABORT                           ' dann brich ab
  ENDIF
  loop 9999999999
  for i = 0 to 255
    XOUT ( i, 0 )                        ' Erw. I/O Port (phys.) low
  '   WAIT_DURATION 200                  ' 200ms warten
  next
endloop
ABORT:
END                                     ' Ende Task MAIN

```

3

Siehe auch: XBUS_INR\$, XBUS_OUTR, XIN\$, XOUT

Umschreibung mathematischer Funktionen

Umschreibung mathematischer Funktionen

Einige mathematische Funktionen, die nicht direkt in Tiger-BASIC[®] enthalten sind, können durch eine Umschreibung realisiert werden.

Die folgende Liste enthält weitere nicht integrierten mathematischen Funktionen, die aus den integrierten mathematischen Funktionen abgeleitet werden können:

3

Funktion	Abgeleitete Äquivalenten
X hoch y	$x^y = e^{y \cdot \ln(x)}$
Nte Wurzel aus x	$= e^{(1/n) \cdot \ln(x)}$
Sekans	$\text{Sekans}(X) = 1 / \cos(X)$
Kosekans	$\text{Kosekans}(X) = 1 / \sin(X)$
Kotangens	$\text{Kotangens}(X) = 1 / \tan(X)$
Arkussinus	$\text{Arkussinus}(X) = \text{Atn}(X / \sqrt{-X^2 + 1})$
Arkuskosinus	$\text{Arkuskosinus}(X) = \text{Atn}(-X / \sqrt{-X^2 + 1}) + 2 \cdot \text{Atn}(1)$
Arkussekans	$\text{Arkussekans}(X) = \text{Atn}(X / \sqrt{X^2 - 1}) + \text{Sgn}(X - 1) \cdot (2 \cdot \text{Atn}(1))$
Arkuskosekans	$\text{Arkuskosekans}(X) = \text{Atn}(X / \sqrt{X^2 - 1}) + (\text{Sgn}(X) - 1) \cdot (2 \cdot \text{Atn}(1))$
Arkuskotangens	$\text{Arkuskotangens}(X) = \text{Atn}(X) + 2 \cdot \text{Atn}(1)$
Hyperb. Sinus	$\text{HSin}(X) = (\exp(X) - \exp(-X)) / 2$
Hyperb. Kosinus	$\text{HCos}(X) = (\exp(X) + \exp(-X)) / 2$
Hyperb. Tangens	$\text{HTan}(X) = (\exp(X) - \exp(-X)) / (\exp(X) + \exp(-X))$
Hyperb. Sekans	$\text{HSekans}(X) = 2 / (\exp(X) + \exp(-X))$
Hyperb. Kosekans	$\text{HKosekans}(X) = 2 / (\exp(X) - \exp(-X))$
Hyperb. Kotangens	$\text{HKotangens}(X) = (\exp(X) + \exp(-X)) / (\exp(X) - \exp(-X))$
Hyperb. Arkussinus	$\text{HArkussinus}(X) = \log(X + \sqrt{X^2 + 1})$
Hyperb. Arkuskosinus	$\text{HArkuskosinus}(X) = \log(X + \sqrt{X^2 - 1})$
Hyperb. Arkustangens	$\text{HArkustangens}(X) = \log((1 + X) / (1 - X)) / 2$
Hyperb. Arkussekans	$\text{HArkussekans}(X) = \log((\sqrt{-X^2 + 1} + 1) / X)$
Hyperb. Arkuskosekans	$\text{HArkuskosekans}(X) = \log((\text{Sgn}(X) \cdot \sqrt{X^2 + 1} + 1) / X)$

Umschreibung mathematischer Funktionen

Funktion	Abgeleitete Äquivalenten
Hyperb. Arkuskotangens	$H\text{Arkuskotangens}(X) = \text{Log}((X + 1) / (X - 1)) / 2$
Logarithmus zur Basis N	$\text{LogN}(X) = \text{Log}(X) / \text{Log}(N)$

Umschreibung mathematischer Funktionen

Leere Seite

3

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger[®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

4 Graphik im BASIC-Tiger®

Unter der Überschrift Applikationen im Device-Treiber-Handbuch befinden sich die Beispiele zum BASIC-Tiger® Graphic-Toolkit. Die Beispiele sind eine Weiterführung dieses Kapitels. Das Touch-Panel als Tastatur ist dort ebenfalls beschrieben.

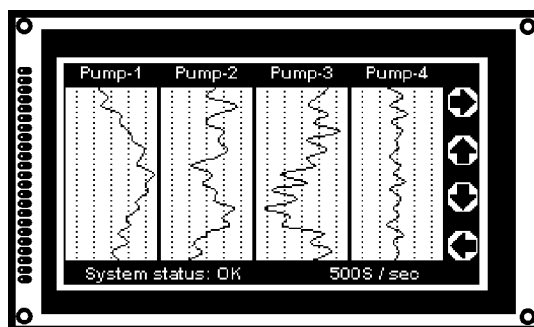
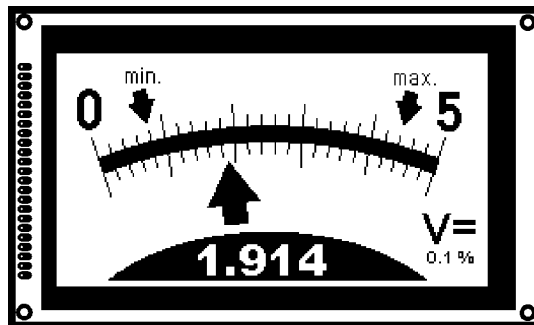
Übersicht über Grafikfunktionen

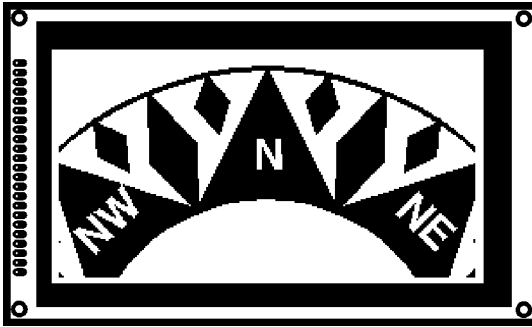
AND2\$	589
AND3\$	589
AND4\$	589
OR2\$	589
OR3\$	589
OR4\$	589
XOR1\$	589
GRAPHIC_MASK_COPY	600
GRAPHIC_MIRROR	604
GRAPHIC_COPY	607
INVERT	627
GRAPHIC_FILL_MASK	629
GRAPHIC_EXP\$	638
DRAW_LINE	645
DRAW_NEXT_LINE	727
CLOSE_LINE	729
SET_BASE	731
SET_SCALE	734
SET_ROTATION	736
SET_GRAREA	739
SET_DOT	741
DISTANCE	743
FILL_AREA	745

Ein Bild sagt mehr als tausend Worte.

Grafikfunktionen verhelfen Anwendungen zu klareren und besser verständlichen Ausgaben. Auf Bildschirmen und Druckern lassen sich Bilder und Graphiken zeigen, Zeichnungen, Schaubilder, Instrumente, Kurven und vieles mehr.

Einige Beispiele:





Grafik im BASIC-Tiger® ist einfach zu programmieren und zeichnet sich aus durch:

- hohe Geschwindigkeit / schnellen Bildaufbau,
- fertige Funktionen für immer wiederkehrende Anwendungen,
- große Flexibilität durch Operationen wie:
 - - Überlagerungen
 - - Maskierungen
 - - Verschiebungen
 - - Skalierungen

Alle Grafik Funktionen laufen komplett im BASIC-Tiger® selbst ab, es sind also keine Grafikkarten oder spezielle Controller für die Erzeugung und Bearbeitung von Grafik erforderlich. Grafikdaten werden in der Regel in Strings oder in FLASH-Bereichen des Tigers aufgebaut.

4

Die Art und der Typ des grafischen Ausgabegerätes ist bei der Erzeugung und Bearbeitung von Grafiken noch nicht von Bedeutung. Erst bei der Ausgabe von Bildern und bewegten Sequenzen werden durch Wahl des geeigneten Device-Treibers oder durch eigene BASIC Routinen die Belange des jeweiligen grafischen Ausgabegerätes berücksichtigt.

Als Hilfsmittel für den Entwickler steht ein spezielles Grafik-Toolkit zur Verfügung, das über die erforderliche Hardware wie Grafikbildschirm, Touch-Screen, Mouse-Input, etc. sowie zahlreiche Beispiel-Applikationen verfügt. Die hier im Handbuch aufgeführten Beispielprogramme stammen aus diesem Toolkit und können dort sofort ausgeführt werden.

Tiger-BASIC® unterscheidet grundsätzlich 2 Arten von Grafik Darstellungen:

- Pixelgrafik
- Vektorgrafik

Pixelgrafik - Vektorgrafik

Bei Pixelgrafik handelt es sich um die 1:1-Repräsentation von Bildpunkten durch einzelne Bits. Ein ,1'-Bit entspricht einem „schwarzen“ Punkt bzw. der Vordergrundfarbe, eine ,0' einem „weißen“ Punkt bzw. der Hintergrundfarbe.

- Bitwert = „0“ → weißes Pixel
- Bitwert = „1“ → schwarzes Pixel

Die 8 Bits eines Bytes bilden eine Pixelgruppe von 8 waagerecht angeordneten Pixeln, wobei das höchstwertige Bit dem am weitesten links stehenden Pixel entspricht:

Bitnummer	7	6	5	4	3	2	1	0
Bitwert	1	0	0	0	1	1	1	1
Pixel								

Die aufeinander folgenden Bytes eines Strings bilden im Bereich der Pixeldaten die sich daran anschließenden Pixelgruppen:

1. Byte				2. Byte				3. Byte			
8FH				33H				18H			
1	0	0	0	1	1	1	1	0	0	1	1

4

Und schließlich wird aus einem langen „Faden“ sequentiell aufgereihter Pixel eine 2-dimensionale Darstellung, indem man jeweils nach Erreichen der Bildbreite von nn Pixeln, wieder am Zeilenanfang beginnt.

Beispiel: 16 x 14 Pixelgrafik, dargestellt als Pixel Abbildung und als String (hex)

	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Zeile 0																
Zeile 1																
Zeile 2																
Zeile 3																
Zeile 4																
Zeile 5																
Zeile 6																
Zeile 7																
Zeile 8																
Zeile 9																
Zeile 10																
Zeile 11																
Zeile 12																
Zeile 13																

Das gleiche als String (hex): „01 00 03 80 07 C0 0F E0 1F F0 10 10 12 50 ...“%

Die **Formatbreite** in obigem Beispiel beträgt 16 Pixel, die Höhe 14 Pixel. Aus der Art der Pixel-Repräsentation in den Bytes eines Strings, wird klar, daß die Formatbreite einer Grafik stets ein ganzes Vielfaches der Zahl 8 beträgt, da immer nur eine ganze Anzahl von Bytes je Pixelzeile verwendet werden kann.

Dies führt jedoch zu keiner Einschränkung der Allgemeinheit, wenn man zuläßt, daß für die Manipulation, das Bewegung und das Erzeugen von Grafiken auch jeder andere, nicht durch 8 teilbare Wert für die **Bildbreite** erlaubt wird:

	Format	Bild
Breite	immer ein Vielfaches von 8	beliebig
Höhe	beliebig	beliebig

4

Dazu das Beispiel eines kleinen grafischen LC Displays mit 20 x 20 Pixeln (FORMAT-Angabe!):

	Bit-Positionen in zugehörigen Bytes																			
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4
0																				
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				
12																				
13																				

Nutzbares Bit = sichtbares Pixel:	
Bit in Grafikstring, wird jedoch nicht angezeigt:	•

Obwohl der Grafikstring für ein formatfüllendes Bild auf diesem Display 60 Byte (3 x 20) bzw. 480 Bits enthält, werden nur 20 x 20 = 400 Pixel dort dargestellt. Die überzähligen Bits, hier als „•“ dargestellt, werden zwar als Speicherplatz belegt, jedoch niemals angezeigt.

Der geringfügigen „Verschwendung“ von Bits in solchen Fällen steht der große Vorteil einer einheitlichen Bit-zu-Pixel Zuordnung gegenüber. Stets beginnt eine Pixelzeile mit Bit-7 eines Bytes.

Regeln in BASIC-Tiger® Pixelgrafik:

- 1 Bit ↔ 1 Pixel
- Bitwert „0“ = „weiß“, Bitwert „1“ = „schwarz“
- 1 Byte = 8 Bits = 8 waagerechte Pixel
- Bit-7 → Pixel steht am weitesten links
- Das 1. Byte eines Strings steht in der linken oberen Ecke einer Grafik
- Das letzte Byte eines (formatfüllenden) Strings steht rechts unten in der Grafik.

4

Bedeutung von Pixelgrafik

Pixelgrafik ist die übliche Form der Grafikdarstellung für die Ausgabe von Grafiken auf grafischen Bildschirmen (LC-Display) und Druckern (Matrix-, Thermo, Laser-Drucker). Je nach Gerätetyp kann es sein, daß die Pixel-Repräsentation für die Ausgabe auf das Ausgabegerät noch reformatiert werden muß.

Reformatierung erfolgt entweder:

- im Device-Treiber,
- durch spezielle Grafikfunktionen, oder
- durch eigene BASIC Routinen

Anwendungen, bei denen das Quellformat von Grafiken typischerweise in Pixel-Darstellung vorliegen sind:

- Fotos, low-resolution Video
- gescannte Strichvorlagen
- Firmen- und Sonderzeichen
- Buchstaben und Ziffern in besonderem Design
- sehr kleine Bilder und Symbole
- bewegte Animationen, besonders bei kleinen Abmessungen
- Stadtpläne ... etc.

Pixelgrafiken belegen stets den gleichen Speicherplatz, unabhängig davon ob und was in einer Pixelgrafik dargestellt wird. Pixelgrafiken mit besonderen Strukturmerkmalen können durch Datenkompression in Ihrem Speicherbedarf reduziert werden. Dabei sind Speicherplatz und Rechenzeit für Kompression und Dekompression gegeneinander abzuwägen.

Ein möglicher Weg, Speicherplatz durch Kompression zu sparen und dennoch kurze Rechenzeiten zu erreichen, könnte darin bestehen, komprimierte Pixelgrafiken im FLASH-Datenbereich zu speichern und sie beim Programmstart einmalig in Variablen (RAM) zu dekomprimieren. Im weiteren Programmablauf wird dann direkt auf die fertigen (dekomprimierten) Pixelgrafiken zugegriffen.

Bearbeitungsmöglichkeiten von Pixelgrafik

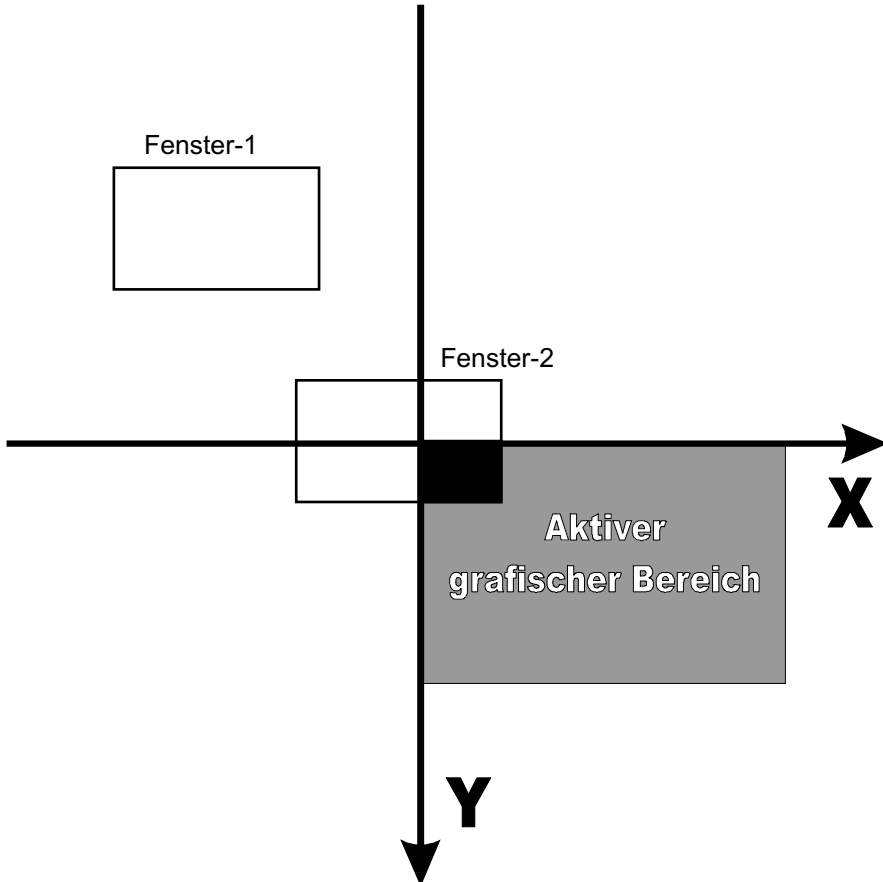
Pixelgrafiken können im BASIC-Tiger® mit diesen Instrumenten bearbeitet werden:

- kopieren
- ausschneiden
- spiegeln
- invertieren
- überlagern
- verschieben
- maskieren

Die Bearbeitung von Pixelgrafik erfordert nur wenig CPU-Zeit im BASIC-Tiger®, so daß Pixelgrafiken schnell bearbeitet werden können und sich für bewegte Animationen gut eignen.

Koordinatensystem für Pixelgrafik

Für Pixelgrafiken wird das kartesische Koordinatensystem mit einer nach unten gerichteten Y-Achse angewendet, was den Gegebenheiten bei typischen Ausgabegeräten wie Grafik-LCD oder Matrix-Drucker entspricht (vergleiche: Vektorgrafik).



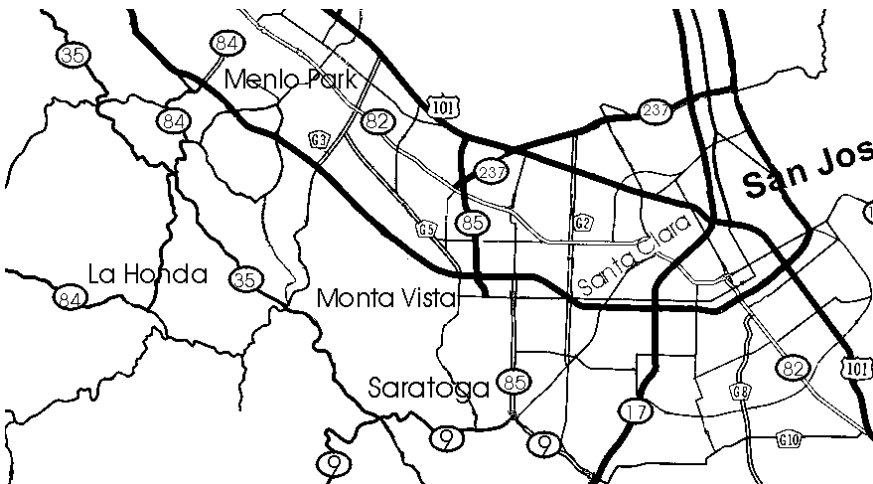
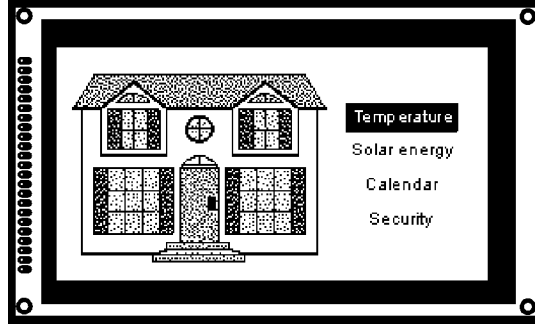
4

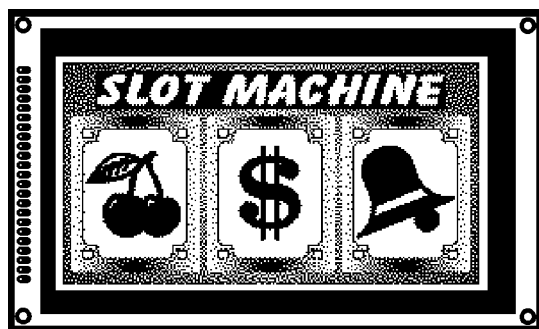
Mit Pixelgrafiken kann man in allen 4 Quadranten arbeiten, sichtbar werden in jedem Fall nur die Bestandteile einer Pixelgrafik, die im „aktiven grafischen Bereich“ liegen. Wird eine grafische Manipulation im „Fenster-1“ vorgenommen, so hat das keine Auswirkung auf die tatsächlich vorhandene Pixel Grafik im „aktiven grafischen Bereich“. Eine Manipulation im „Fenster-2“ hingegen wird zu einer Beeinflussung des schwarz markierten Überlappungsbereichs führen und die Pixelgrafik dort verändern.

Diese Eigenschaft wird u.a. ausgenutzt, um beliebige Ausschnitte aus Pixelgrafiken bilden und diese frei auf einem Bildschirm verschieben zu können. Solche Ausschnit-

te können den ganzen Bildschirm überdecken oder nur ein kleines Fenster im „aktiven grafischen Bereich“ belegen.

Typische Beispiele für Pixelgrafik





4

Vektorgrafik

Bei Vektorgrafik wird die Bildebene mit einem Koordinatennetz überzogen, so daß jedem Punkt der Ebene ein Koordinaten-Wertepaar zugeordnet werden kann.

Zuordnung Pixel → kartesisches Koordinatensystem:

	X-Koordinate 															
Y-Koord.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																
3												■				
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																

Koordinate von des Pixels = (11,3)

Besonderheit in diesem Koordinatensystem ist die nach unten gerichtete Y-Achse. Hierdurch entsteht eine inverse Orientierung gegenüber dem häufig gebrauchten Schema mit nach oben gerichteter Y-Achse.

Grund für diese Orientierung ist die in Grafik-LCDs und Matrix-Druckern übliche Zuordnung von Speicher zu Pixel, bzw. die zeitliche Zuordnung wann welche Pixelzeilen ausgedruckt und übertragen werden.

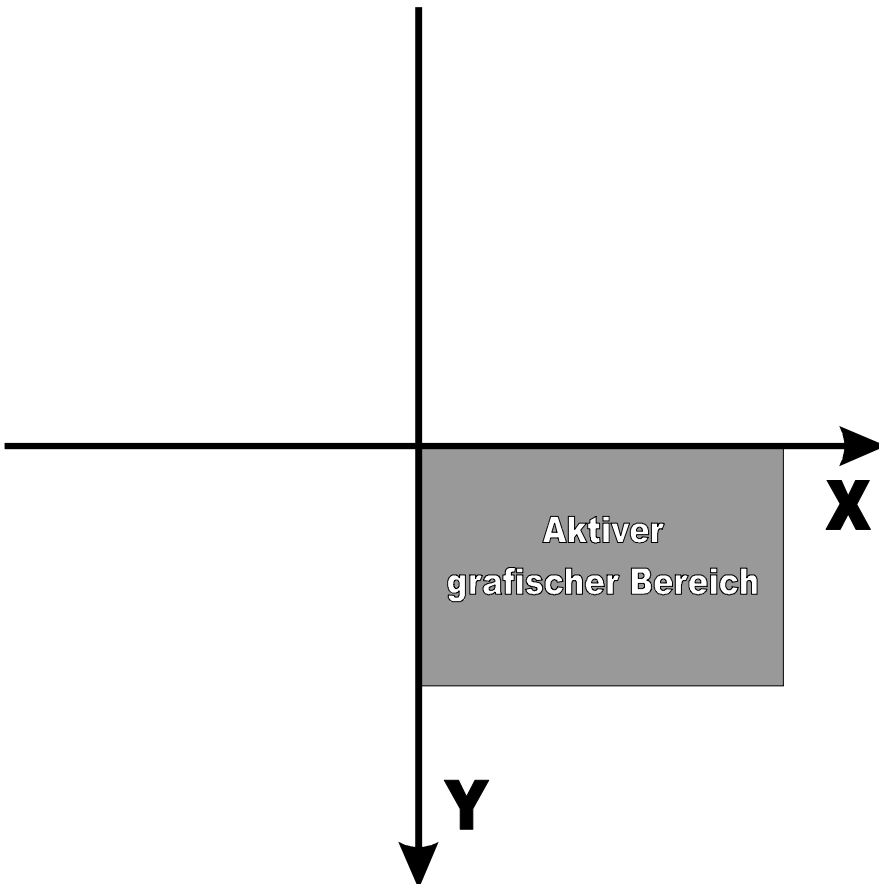
Das kartesische Koordinatensystem gibt die Verhältnisse der meisten grafischen Ein-/Ausgabe-Geräte optimal wieder:

- Bildschirme
- Drucker
- Laseranlagen
- NC-Maschinen
- Scanner ... etc.

Das kartesische Koordinatensystem im BASIC-Tiger® arbeitet mit folgenden Wertebereichen und Auflösungen:

X-/Y-Koordinaten: -8191...0...8191, Step = 1

4



Der im Bild mit „Aktiver grafischer Bereich“ markierte Bereich entspricht dem Bereich, der von einem String mit Pixeldaten abgedeckt wird. Dies kann z.B. auch der sichtbare Bereich eines grafischen LC Displays sein. Bei Vektorgrafik sind ebenfalls negative wie positive Koordinatenwerte zulässig, jedoch werden Vektorgrafik Anweisungen stets auf den aktiven Grafikbereich begrenzt. Zeichnet man über den Rand des aktiven Grafikbereichs hinaus, so wird die Linie automatisch am Rand begrenzt.

Darüber hinaus weit verbreitet sind auch polare Koordinatensysteme, die insbesondere dort eingesetzt werden, wo z.B. physikalische Gegebenheiten dieses System nahelegen, z.B:

- Radaranlagen
- Ultraschall-Diagnostik
- Richtcharakteristiken von Strahlern / Sensoren, etc.

4

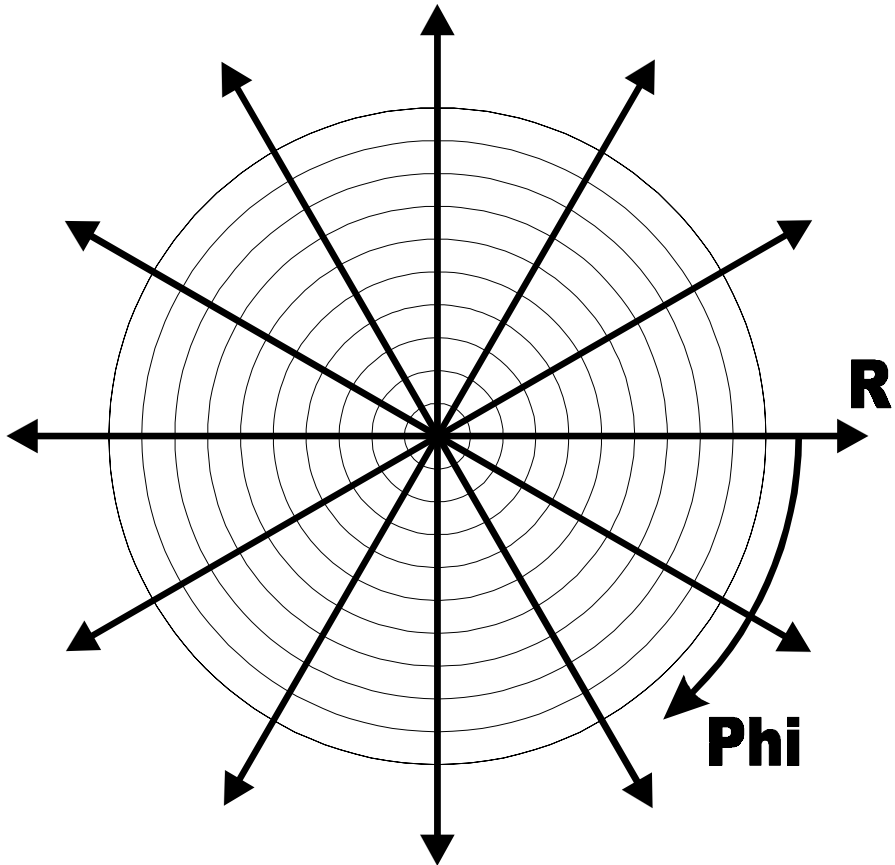
Tiger-BASIC® unterstützt auch Polarkoordinaten, die in entsprechenden Anwendungen Vereinfachungen bieten. Man beachte die damit jedoch grundsätzlich auftretenden Iterationsfehler beim Übergang vom einen zum anderen Koordinatensystem.

Das Polar-Koordinatensystem im BASIC-Tiger® arbeitet mit folgenden Wertebereichen und Auflösungen:

Radius (R): 0...8191, Step = 1

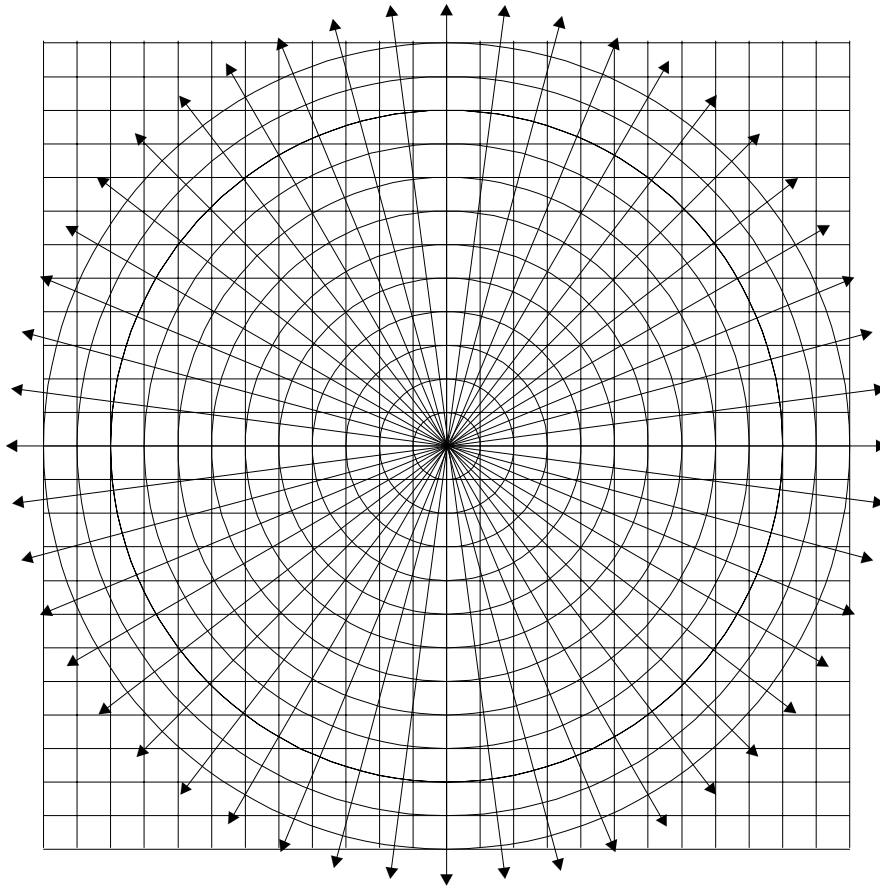
Winkel (PHI): -36000...36000, Step = 1 = 0,01 Grad

Orientierung des Polar-Koordinatensystems:



4

Den Zusammenhang von kartesischen und Polar-Koordinaten zeigt folgendes Bild:



4

Vektorgrafik ermöglicht sehr flexible visuelle Umsetzungen. Vektorgrafikbilder werden durch eine Folge von Koordinatenpaaren und zusätzlichen Parametern beschrieben. Koordinatenpaare können aus unterschiedlichen Quellen stammen, z.B.

- aus tabellarischen Daten,
- aus Berechnungen,
- aus Messungen.

Da sich Koordinatenpaare leicht mathematischen Abbildungen unterziehen lassen, können Darstellungen aus Vektorgrafik vielfältigen Transformation und Manipulationen unterworfen werden.

Einige Transformationen werden für eine einfache Handhabung und schnelles Run-time-Zeitverhalten bereits in den Funktionen von Tiger-BASIC® unterstützt:

- Rotation
- Skalierung
- Verschiebung
- Koordinaten-Transformation
- Flächen füllen

Bei nicht zu hoher Komplexität erlaubt Vektorgrafik die Speicherung von Abbildungen mit nur geringem Speicherplatzbedarf.

Grafik-LCD Funktionen

Die Funktionen, die speziell für die Programmierung des Grafik-LCD entwickelt wurden, sind in der folgenden Tabelle zusammen aufgeführt und kurz erläutert. Im Programmierhandbuch finden Sie ein Kapitel, welches nur der Grafik gewidmet ist..

4

Funktionsname	Pixel-grafik	Vek-tor-grafik	
OR2	•		2 Grafiken werden mit unbedingtem OR überlagert
OR3	•		3 Grafiken werden mit unbedingtem OR überlagert
OR4	•		4 Grafiken werden mit unbedingtem OR überlagert
AND2	•		2 Grafiken werden mit unbedingtem AND überlagert
AND3	•		3 Grafiken werden mit unbedingtem AND überlagert
AND4	•		4 Grafiken werden mit unbedingtem AND überlagert
XOR1	•		2 Grafiken werden mit unbedingtem XOR verknüpft
GRAPHIC_MASK_COPY	•		2 Grafiken werden über Maske verknüpft
GRAPHIC_MIRROR	•		Spiegeln einer Grafik an: X-, Y- oder X+Y Achsen
GRAPHIC_EXP\$	•		Expandiert Grafik-Bytes
GRAPHIC_COPY	•		Kopiert Grafikfenster

Funktionsname	Pixel- grafik	Vek- tor- grafik	
GRAPHIC_FILL_MASK	•		Füllt Rechteck-Maske in Grafik
INVERT	•		Invertiere Pixelwerte
DRAW_LINE		•	Beginnt Linie in Grafikbereich
DRAW_NEXT_LINE		•	Setzt Linie in Grafikbereich fort
CLOSE_LINE		•	Schließt Linie
SET_ROTATION		•	Setzt Rotationswinkel 0,01-Grad
SET_SCALE		•	Setzt Maßstab (100% = 1000)
SET_BASE		•	Setzt Basispunkt X/Y
SET_GRAREA		•	Setzt Grafikbereich: Destin + B x H
SET_DOT		•	Setzt DOT zu 0 or 1
FILL_AREA		•	Füllt Grafikbereich
Zubehör-Funktionen:			
DISTANCE			Berechnet Entfernung zweier Koordinaten
QUICK_WORD_SIN			Berechnet schnellen Sinus in WORD
QUICK_WORD_COS			Berechnet schnellen Cosinus in WORD

Arbeiten mit Pixelgrafik

Unter Pixelgrafik wollen wir die Darstellung „schwarzer“ und „weißer“ Punkte (Pixel) durch einen zugeordneten Bitwert von „1“ bzw. „0“ verstehen. Diese Pixeldaten werden im BASIC-Tiger[®] üblicherweise in Strings oder im Flash-Datenbereich gespeichert.

Pixelgrafiken können auf verschiedene Weise erzeugt werden, sei es durch die Vektorgrafik Funktionen von Tiger-BASIC[®], durch Einlesen von Grafiken über Schnittstellen oder durch eigene BASIC-Programme. Eine weitere Möglichkeit besteht im Import von Grafiken aus anderen Programmen (Grafikprogramme, CAD, DTP, etc.). Das Format „Bitmap“ (*.BMP) kann direkt in ein Tiger-BASIC[®] Programm importiert werden.

Grafik im Flash

Um solche Daten in das interne Format des Tigers zu übertragen wird das Filter GRAPHFLT verwendet:

DATA FILTER "MOZART1.BMP", "GRAPHFLT", 0 ' mozart

Diese Instruktion bewirkt, daß die Datei "MOZART1.BMP", in dem sich eine Bitmap-Grafik befindet, in den Datenbereich des FLASH geladen wird und dabei in das beschriebene, interne Pixelgrafik Format übertragen wird.

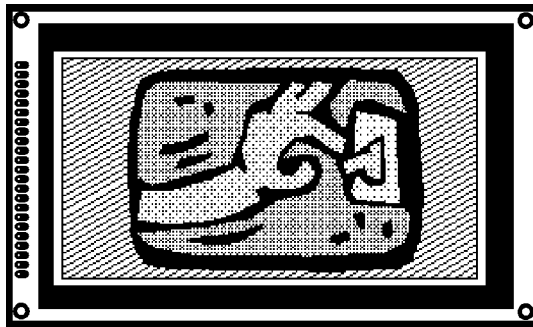
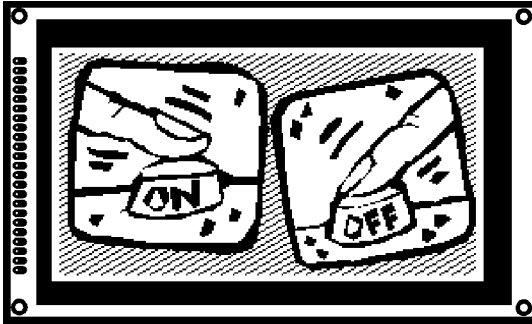


Zur Erzeugung von Bitmap-Grafiken, die im Tiger verwendet werden sollen, hier einige Hinweise:

- Wählen Sie stets eine Formatbreite im ganzen Vielfachen von 8. Dies entspricht den Konventionen im BASIC-Tiger® und bedeutet keine Einschränkung der Allgemeinheit für die weitere Bearbeitung solcher Grafiken.
- Die Höhe der Grafik kann beliebig gewählt werden.
- Wählen Sie die Art „Schwarzweiss-Grafik“. D.h. es wird eine Bitmap-Grafik mit nur 2 Farbwerten erzeugt: schwarzen und weißen Pixeln (nicht: Grauwert- oder Farbgrafik).
- Wenn Sie Bilder mit Farbe oder Grauwerten verwenden wollen, so können diese durch Rastern in ein darstellungsfähiges Format gebracht werden. Es gibt verschiedene Rastertypen und Methoden. Punktraster und Dithering sind gebräuchlich.
- Wenn Sie Animations-Sequenzen gestalten, berücksichtigen Sie ggf. die optische Trägheit von LC-Displays. Sie führt dazu, daß schnelle Wechsel und feine Linien nicht wie erwartet wiedergegeben werden.

Hier 3 Bild-Beispiele für:

- Bild als Strichzeichnung,
- Bild mit technischem Raster,
- und Bild mit Dithering.



In den Beispielpogrammen dieses Kapitels werden richtige und fehlerhafte Verwendung von importierten Pixelgrafiken demonstriert.

Grafik & Geschwindigkeit

Mit den hier vorgestellten Grafikfunktionen für Vektor- und Pixelgrafik, bekommt der Entwickler beachtliche Möglichkeiten in die Hand für optisch ansprechende und informative Ausgaben auf Displays und Druckern. Für die Grafikverarbeitung und die Ausgabe sind keine speziellen Grafik-Boards oder Controller erforderlich. Alle Operationen für Bildaufbau und -bearbeitung sowie die Ausgabe auf einem Display laufen komplett im BASIC-Tiger® ab.

Dies sollte man berücksichtigen, wenn es um die Planung von Anwendungen geht, bei denen sowohl:

- sehr intensiver Grafik-Gebrauch, als auch
- sehr schnelle Ausgabe-Sequenzen (Filme, Animationen ...) gewünscht werden.

4

Gut geplant und mit ein paar Tips läßt sich die ohnehin schnelle Grafik des BASIC-Tigers® noch besser nutzen und zusätzliche CPU Zeit für andere Tasks gewinnen.

Wenn Sie grafik-intensive Anwendungen entwickeln, hier einige Tips:

Bild-Wiederholffrequenz nur so hoch auslegen wie sinnvoll. Also berücksichtigen: muß man eine Grafik 5-mal oder 10-mal pro Sekunde auffrischen, oder reichen schon 2-, 3-, 4-mal pro Sekunde für ein gutes Ergebnis?

TIP

Bereiche mit schnellen Veränderungen (Meßinstrumente, Animation, Low-Res Video) und Bereiche mit statischen, oder nur gelegentlich aufgefrischten Daten sollten getrennt werden. Schnell veränderliche Pixelzeilen können öfter aufgefrischt werden, ohne daß gleich das gesamte Bild neu berechnet und komplett an den LC-Bildschirm übertragen werden muß.

TIP

Gibt es konstante Bildelemente oder Bildhintergründe? Diese können als Pixelgrafik importiert werden (Instruktion DATA) oder direkt nach dem Power-ON in Strings aufgebaut werden. Dort stehen sie dann während des gesamten Programmablaufs für den sofortigen Zugriff bereit.

TIP

Wählen Sie möglichst einfache Kopierparameter bei GRAPHIC_COPY. Diese Funktion arbeitet dann besonders schnell, wenn Daten-Bits vom Quellgebiet zum Zielgebiet nicht in ihrer Bitposition verschoben werden müssen. Also, besonders schnell gehen zum Beispiel:

TIP

Quelle	Ziel
schneller:	
X-Pos = 50	...34 oder 42 oder 50 oder 58 oder 66 oder 74 ...
X-Pos = 77	.. 61 oder 69 oder 77 oder 85 oder 93 oder 101 ...
X-Pos = 18	... 2 oder 10 oder 18 oder 26 oder 34 oder 42 ...
Langsamer arbeiten z.B:	
X-Pos = 49	... 34 oder 42 oder 50 oder 58 oder 66 oder 74 ...
X-Pos = 76	... 61 oder 69 oder 77 oder 85 oder 93 oder 101 ...
X-Pos = 17	... 2 oder 10 oder 18 oder 26 oder 34 oder 42 ...

TIP Wählen Sie eine hohe Übertragungsrate zum LC-Display, wodurch die CPU-Last pro Byte reduziert wird.

Grafik mit PEEK_FLASH

Das folgende Beispiel zeigt den Import und die Bildschirmausgabe der Grafik „MO-ZART1.BMP“ auf einem 240 x 128-Pixelgrafikdisplay (T6963C). In Tiger-BASIC® Programmen können Daten aus dem Dateisystem des PCs durch die Instruktion DATA mit dem Zusatz „FILE“ bzw. „FILTER“ in den Flash-Datenbereich des Tigers geladen werden.

Die Angabe des Filternamens „GRAHFLT“ sowie des Parameters: 0 veranlaßt den Compiler das „*.BMP“ Datenformat in das interne Pixelformat des Tigers umzusetzen.

Gelesen wird hier die Grafik mit der Instruktion PEEK_FLASH. „SCREEN\$“ nimmt die Grafik auf und übergibt sie zum Ausdruck an den Device-Treiber (PUT #12, ...).

Beachten Sie bei der Compilation dieses Beispiels bitte die korrekte Angabe des Pfadnamens für die zu importierende Datei, entweder zusammen mit dem Dateinamen, oder allgemein gültig unter „OPTIONEN - VERZEICHNISSE“ in der Menüleiste der Oberfläche.

4

Programmbeispiel:

```

-----
' Name: GRAPHIC_PEEK.TIG - PIXEL graphic: import, display pixelgraphic
' initializes the LCD and outputs a graphic peeked out of flash memory
-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
-----

USER_VAR_STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K)             ' Global Strings for Pixel-data
DATALABEL PIX_DATA

TASK MAIN
PIX_DATA:
    DATA FILTER "MOZART1.BMP", "GRAPHFLT", 0

    DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
    DIR_PORT 7,11111101b        ' L71 output: LCD reset
    OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
    OUT P_LCD_RESET,M_LCD_RESET,255 ' "
    OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
    OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
    OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
    INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

    PEEK FLASH PIX_DATA, SCREEN$, 3820
    PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
END

```

4



Grafik direkt aus dem Flash

Dieses Beispiel erzeugt den gleichen Ausdruck wie das vorige Beispielprogramm. Im Unterschied dazu wird hier lediglich auf die Instruktion PEEK_FLASH verzichtet, da man Pixelgrafiken auch direkt aus dem Flash-Datenbereich ausgeben kann. Dies vereinfacht die Ausgabe von Standardgrafiken, die als Bibliotheken im Flash vorliegen und nicht erst in eine RAM-Variable eingelesen werden müssen.

Programmbeispiel:

```

'-----
' Name: GRAPHIC_DIRECT.TIG PIXEL graphic: import,display pixel graphic
' initializes the LCD and ouputs a graphic direct from flash memory
'-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
'-----

USER_VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

DATALABEL PIX_DATA

TASK MAIN
PIX_DATA:
    DATA FILTER "MOZART1.BMP", "GRAPHFLT", 0

    DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
    DIR_PORT 7,11111101b        ' L71 output: LCD reset
    OUT P_LCD RESET,M_LCD RESET,0 ' RESET LCD
    OUT P_LCD_RESET,M_LCD_RESET,255 ' "
    OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
    OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
    OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
    INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

    PUT #12, #1, PIX_DATA, 0,0,3840 ' directly to LCD
END

```

4



Bitmap mit zu kurzen Zeilen

Das folgende Beispiel zeigt den Import eines „*.BMP“ Bildes, das mit einer Breite von 239 Pixeln - statt der korrekten Breite von 240 - erzeugt worden ist. Bei der Ausgabe auf einem 240 Pixel breiten Grafikbildschirm erscheint ein unkenntliches Bild mit typischen Verzerrungen.

4

Programmbeispiel:

```

-----
' Name: TOO_SHORT.TIG - PIXEL-Graphic: import + display pixel-graphic
' shows effect of an error:  graphic 239 instead of 240 pixel wide
-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
-----

USER_VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

DATALABEL PIX_DATA

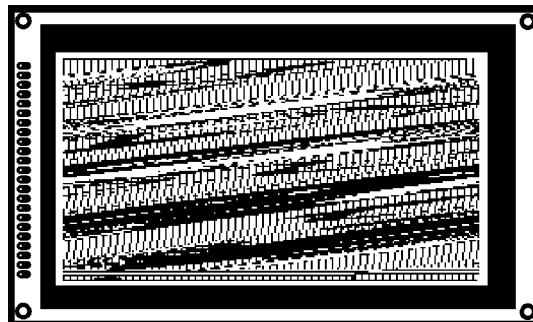
TASK MAIN
PIX_DATA::
    DATA FILTER "MOZARTS.BMP", "GRAPHFLT", 0      ' <-- 239 pixel wide

    DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
    DIR_PORT 7,11111101b        ' L71 output: LCD reset
    OUT P_LCD RESET,M_LCD RESET,0 ' RESET LCD
    OUT P_LCD_RESET,M_LCD_RESET,255 ' "
    OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
    OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
    OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
    INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150 KB/s

    PUT #12, #1, PIX_DATA, 0,0,3840 ' directly to LCD
END

```

4



Bitmap mit zu kleiner Höhe

In diesem Beispiel wurde das Bild im Grafikprogramm mit einer korrekten Breite von 240 Pixeln und einer zu kleinen Höhenangabe von 80 Pixeln exportiert. Als Folge hiervon erscheint das Bild auf dem LCD-Ausdruck zusammengedrückt und der untere Bildschirmbereich zeigt schwarze Pixel, da hier keine Bilddaten mehr vorliegen.

Programmbeispiel:

```

'-----
' Name: TOO_LOW.TIG - PIXEL graphic: import + display pixel graphic
' shows effect of an error: source graphic set to only 80 pixel high
'-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
'-----

USER_VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

DATALABEL PIX_DATA

TASK MAIN
PIX_DATA:
  DATA FILTER "MOZARTL.BMP", "GRAPHFLT", 0 ' <-- height not OK

  DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD RESET,M_LCD RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

  PUT #12, #1, PIX_DATA, 0,0,3840 ' directly to LCD
END

```

4



Bitmap mit zu großer Höhe

Das nachfolgende Programmbeispiel gibt eine Pixelgrafik aus, die im Grafikprogramm mit einer zu großen Höhe - 190 Pixel statt 128 - exportiert worden ist. Als Folge hiervon erscheint das Bild in den Höhe gedehnt und man sieht nur einen Teil des ganzen Bildes. Auch belegt diese Grafik mehr Byte Speicherplatz im Datenbereich des FLASH als erforderlich:

Bildschirmgröße: **$240 \times 128 \text{ Pixel} = 30720 \text{ Pixel} = 3840 \text{ Bytes}$**

Belegter FLASH-Speicher: **$240 \times 190 / 8 = 5700 \text{ Bytes}$**

Programmbeispiel:

```

'-----
' Name: TOO_HIGH.TIG - PIXEL-Graphic: import + display pixel-graphic
' shows effect of an error: source graphic set to 190 pixel high
'-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
'-----

USER_VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

DATALABEL PIX_DATA

TASK MAIN
PIX_DATA:
  DATA FILTER "MOZARTH.BMP", "GRAPHFLT", 0      ' <-- hight not OK

  DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0  ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

  PUT #12, #1, PIX_DATA, 0,0,3840 ' directly to LCD
END

```

4



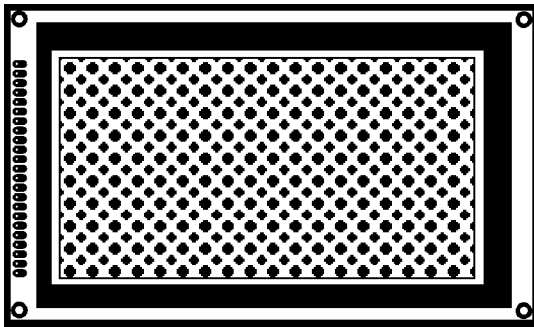
Beispiel-Bitmaps

Nachfolgend sind die Pixelgrafiken wiedergegeben, die sich als „*.BMP“ Files bei den Beispielen befinden. Sie sind auf eine Größe von 240 x 128 Pixeln skaliert, was dem LC-Bildschirm des Grafik-Toolkits entspricht.

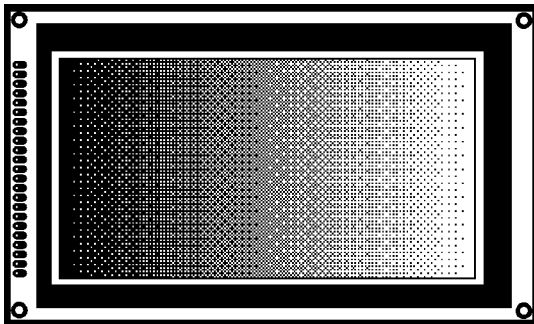
Alle Beispielprogramme zur Bearbeitung und Ausgabe von Pixelgrafik und Vektorgrafik laufen auf dem „BASIC Tiger Grafik-Toolkit“ sofort ab. Dort enthalten sind auch weitere Grafiken und Programme für die Grafikbearbeitung, sowie die Eingabe mit Zeigegegeräten (Touch-Screen, Maus, Encoder, etc.) und typische Applikationen.

Es folgen einige Seiten mit Pixelgrafiken (BMP) in alphabetischer Reihenfolge sortiert nach Namen.

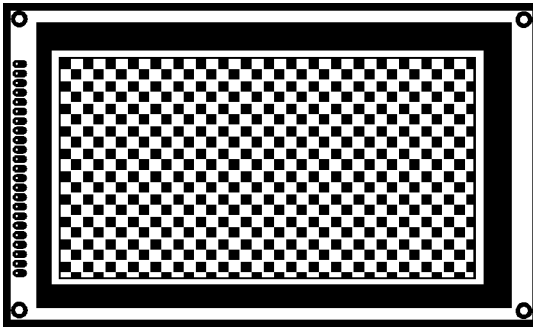
4



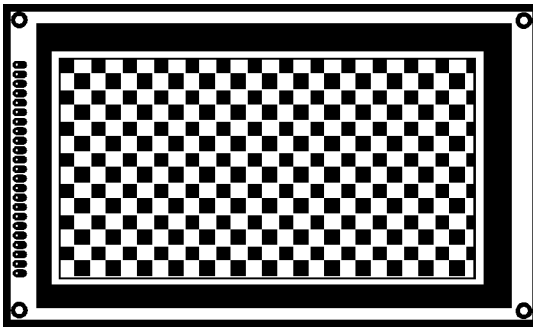
BIGDOTS.BMP



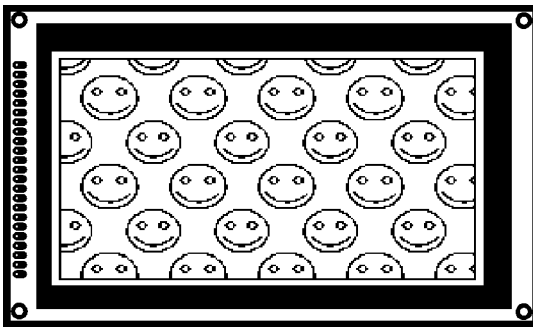
GREYRAMP.BMP



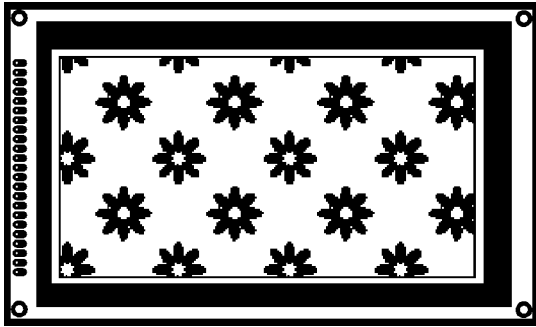
RECT_S.BMP



RECT_M.BMP

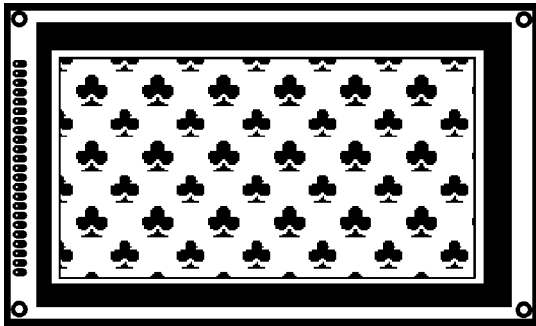


SMILIES.BMP



FLOWERS.BMP

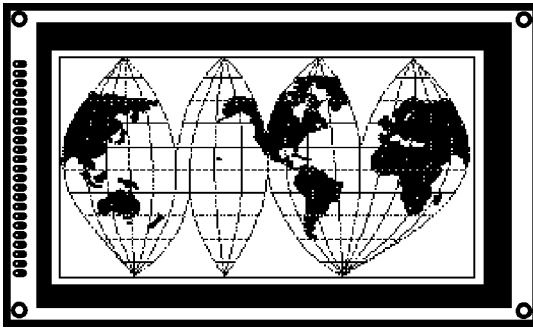
4



CLUBS.BMP



EINSTEIN.BMP



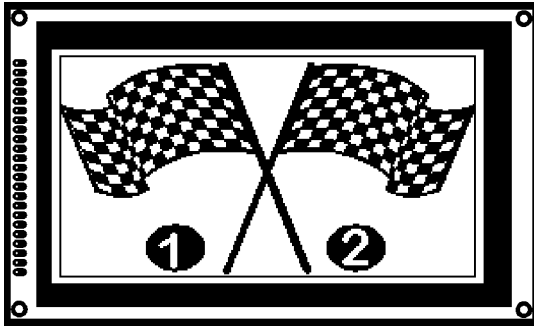
WORLD.BMP



MOZART1.BMP

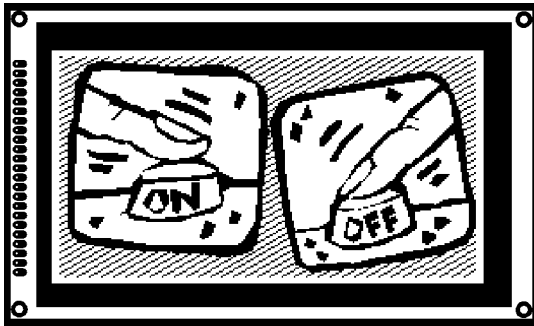


WOMAN.BMP

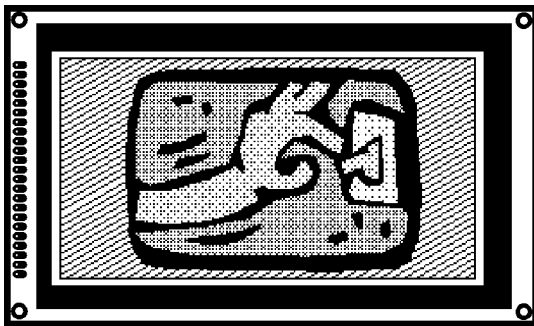


RACEFLGS.BMP

4



ON-OFF.BMP



SWITCH.BMP



MASK_AB.BMP



MASKABC1.BMP



MASKABC2.BMP



MASKABC3.BMP

4



HELLO.BMP

AND\$, OR\$, XOR\$**AND2\$ (A\$, B\$)****AND3\$ (A\$, B\$, C\$)****AND4\$ (A\$, B\$, C\$, D\$)****OR2\$ (A\$, B\$)****OR3\$ (A\$, B\$, C\$)****OR4\$ (A\$, B\$, C\$, D\$)****XOR1\$ (A\$)**

AND2\$, AND3\$, AND4\$: Vereint 2, 3 oder 4 Strings byteweise per AND

OR2\$, OR3\$, OR4\$: Vereint 2, 3 oder 4 Strings byteweise per OR

XOR1\$: Überlagert 1 String byteweise per XOR
(alle nicht in Projektmodell 'pm_min')**4**

Diese Funktionen vereinen Strings byteweise mit der AND, OR, bzw. XOR Verknüpfung.

Damit diese Verknüpfungen bei Pixelgrafiken zu sinnvollen Ergebnissen führen, müssen alle beteiligten Strings die gleiche Formatbreite besitzen. Unterschiedliche Formathöhe führt dazu, daß die kleinste vorkommende Höhe für die Überlagerung verwendet wird.

Die Beispiele zeigen die Wirkung dieser Funktionen.

Diese logischen Überlagerungen von Strings sind einfach in der Anwendung und laufen im Tiger schnell ab. Man verwendet sie daher z.B. gerne in Anwendungen, wo Grafiken auf mehreren Ebenen aufgebaut werden, die dann gemeinsam zur Anzeige kommen sollen. So können konstante und veränderliche Bildkomponenten separat gehalten werden, was organisatorische Vorteile und schneller ablaufende Ergebnisse bringt.

Bispiel: Oszilloskop Darstellung. In einer solchen Anwendung werden z.B. folgende Bestandteile auf getrennten Grafikebenen geführt, um sie dann zur Anzeige mit einer OR-Verknüpfung zu vereinigen:

- Ebene-1: Oszilloskop Layout mit Knöpfen, leeren Anzeige-Feldern und Rasterlinien (konstante Ebene)
- Ebene-2: Meßwert-Kurven, 1- oder mehrkanalig - laufend auffrischen.
- Ebene-3: Cursor / Fadenkreuze ... etc. - werden nur bei Bedarf aufgefrischt
- Ebene-4: Knöpfe, Lämpchen, Sonderanzeigen, etc - werden nur bei Bedarf aufgefrischt

Alle 4 Ebenen werden dann vor der Ausgabe mit einer OR4\$ Funktion zu einem Pixel-String vereinigt.

Die **OR Verknüpfung** „addiert“ die schwarzen Bildelemente mehrerer Bilder. Dies gleicht dem Übereinanderlegen von Overheadfolien auf dem Projektor. OR wird in der Regel also eingesetzt um mehrere Zeichnungsebenen zu vereinigen.

Die **AND Verknüpfung** „addiert“ die weißen Bildelemente mehrerer Bilder, oder anders ausgedrückt, die AND Verknüpfung maskiert Bildelemente aus. Nur die Bildbestandteile, die in allen Bildern = schwarz sind, bleiben erhalten. AND wird häufig eingesetzt um Bildbestandteile zu entfernen, oder um z.B. schwarze Flächen mit Rastern / Muster zu hinterlegen (wie ein weiches Ein- und Ausblenden von Bildern).

Die **XOR Verknüpfung** invertiert dort alle Pixel in der Zielgrafik, wo in der Quellgrafik eine „1“ vorliegt, und sie läßt alle Pixel in der Zielgrafik unverändert, wo in der Quellgrafik eine „0“ steht. XOR Masken können z.B. eingesetzt werden um Bereiche in einer Grafik blinken zu lassen.

Es folgen 3 Beispiele, die Wirkung und mögliche Anwendungen der logischen Stringverknüpfungen demonstrieren.

Programmbeispiel:

```

-----
' Name: COMBINE1.TIG - PIXEL graphic
' combining several pixel graphic areas
' shows combinations of pixel graphics using AND2$, OR2$, XOR2$
-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
-----

USER_VAR_STRICT
#include DEFINE A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC            ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K), A$(4K), B$(4K) ' Global Strings for pixels
DATALABEL PICT1, PICT2

TASK MAIN
PICT1::
  DATA FILTER "WOMAN.BMP", "GRAPHFLT", 0 ' Picture of Woman: 240 x 128
PICT2::
  DATA FILTER "GREYRAMP.BMP", "GRAPHFLT", 0 ' grey ramp: 240 x 128

DIR PORT 8,1000000b          ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b        ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL DEVICE #i2,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

PEEK_FLASH PICT1, A$, 3840    ' read picture from FLASH
PEEK_FLASH PICT2, B$, 3840    ' read picture from FLASH

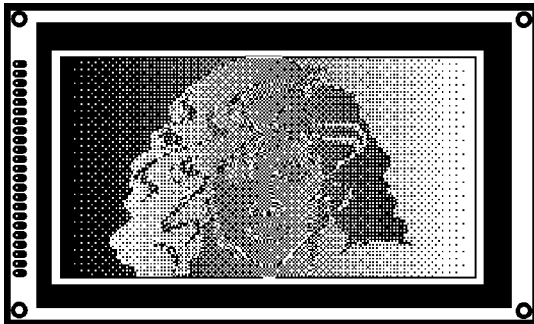
LOOP 999999999              ' many loops
  SCREEN$ = OR2$ (A$, B$)    ' combine 2 graphic areas by OR
  PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
  WAIT DURATION 1000

  SCREEN$ = AND2$ (A$, B$)    ' combine 2 graphic areas by AND
  PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
  WAIT DURATION 1000

  SCREEN$=A$
  SCREEN$ = XOR1$ (B$)        ' combine 2 graphic areas by XOR
  PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
  WAIT DURATION 1000
ENDLOOP
END

```

4



Programmbeispiel:

```

'-----
' Name: PATTERN.TIG - PIXEL-Graphic
' combining several pixel graphic areas
' combines the graphic WOMAN.BMP with different patterns,
' resulting in remarkable picture transitions
'-----
' Note: set correct path for flash data (OPTIONS)
'       connect L71 to reset LCD
'-----

USER VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K),A$(4K),B$(4K),GREY$(4K) ' Global String Pixel-data
DATALABEL PICT1, PICT2
LONG ADR, GLEN, N

TASK MAIN
DIR_PORT 8,1000000b           ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b         ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

PEEK_FLASH PICT1, A$, 3840     ' read picture from FLASH
GLEN=3840                     ' length of 1 picture
GREY$ = FILL$("<0>",4K)       ' generate a "white" Pixel-string

LOOP 999999999               ' many loops
  LOOP 2
    PUT #12, #1, GREY$, 0,0,3840 ' show a white screen now
    WAIT_DURATION 500            '
    FOR ADR=PICT2 TO PICT2+GLEN*9 STEP GLEN'
      PEEK_FLASH ADR, B$, 3840    ' read grey screen from FLASH
      SCREEN$ = AND2$ (A$, B$)   ' combine 2 graphic areas by AND
      PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
      WAIT_DURATION 70          '
    NEXT
    PUT #12, #1, A$, 0,0,3840     ' show the original picture
    WAIT_DURATION 600            ' for 1 second
    FOR ADR=PICT2+GLEN*9 TO PICT2 STEP -GLEN'
      PEEK_FLASH ADR, B$, 3840    ' read grey screen from FLASH
      SCREEN$ = AND2$ (A$, B$)   ' combine 2 graphic areas by AND
      PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
      WAIT_DURATION 70          '
    NEXT
  ENDDLOOP

  LOOP 2
    FOR N=0 TO 8
      B$ = FILL$ (CHR$ (EXP(2,N)-1), 4K) ' build grey screen
      SCREEN$ = AND2$ (A$, B$)   ' combine 2 graphic areas by AND
      PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD

```

```

    WAIT_DURATION 70
NEXT
WAIT_DURATION 600
FOR N=8 TO 0 STEP -1
    B$ = FILL$ (CHR$ (EXP(2,N)-1), 4K) ' build grey screen
    SCREEN$ = AND2$ (A$, B$) ' combine 2 graphic areas by AND
    PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
    WAIT_DURATION 70
NEXT
WAIT_DURATION 600
ENDLOOP
ENDLOOP

PICT1::
    DATA FILTER "WOMAN.BMP", "GRAPHFLT", 0 ' Picture of Woman: 240 x 128
PICT2::
    DATA FILTER "PATTRN10.BMP", "GRAPHFLT", 0 ' grey screen 10%: 240 x 128
    DATA FILTER "PATTRN20.BMP", "GRAPHFLT", 0 ' grey screen 20%: 240 x 128
    DATA FILTER "PATTRN30.BMP", "GRAPHFLT", 0 ' grey screen 30%: 240 x 128
    DATA FILTER "PATTRN40.BMP", "GRAPHFLT", 0 ' grey screen 40%: 240 x 128
    DATA FILTER "PATTRN50.BMP", "GRAPHFLT", 0 ' grey screen 50%: 240 x 128
    DATA FILTER "PATTRN60.BMP", "GRAPHFLT", 0 ' grey screen 60%: 240 x 128
    DATA FILTER "PATTRN70.BMP", "GRAPHFLT", 0 ' grey screen 70%: 240 x 128
    DATA FILTER "PATTRN80.BMP", "GRAPHFLT", 0 ' grey screen 80%: 240 x 128
    DATA FILTER "PATTRN90.BMP", "GRAPHFLT", 0 ' grey screen 90%: 240 x 128
END

```

4

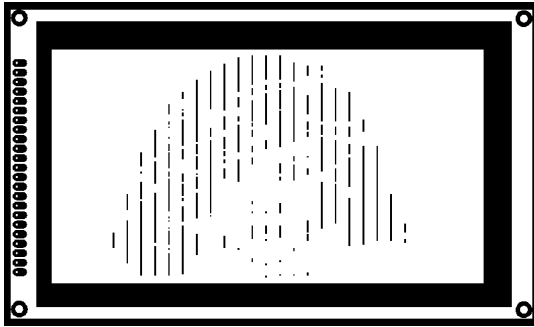


AND\$, OR\$, XOR\$

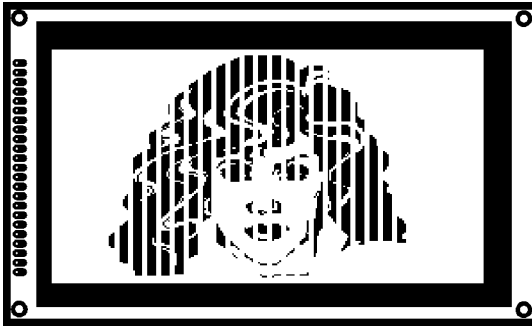


4

4



AND\$, OR\$, XOR\$



4

Programmbeispiel:

4

```

'-----
' Name: OSCILLO1.TIG - PIXEL-Graphic
' combining several pixel graphic areas
' shows a static picture of an oscilloscope screen
'-----
' Note:  set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING SCREEN$(4K),A$(4K),B$(4K)    ' Global String Pixel-data
DATALABEL PICT1, PICT2
LONG ADR, GLEN, N

TASK MAIN
PICT1::
  DATA FILTER "OSCILLO1.BMP", "GRAPHFLT", 0' Oscilloscope: 240 x 128
PICT2::
  DATA FILTER "OSCILLO2.BMP", "GRAPHFLT", 0' Oscilloscope: 240 x 128

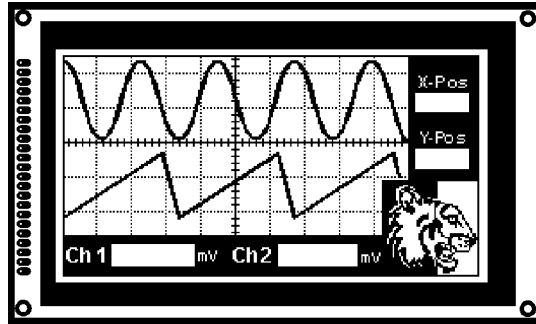
DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0   ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255      ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0      ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255     ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

PEEK_FLASH PICT1, A$, 3840       ' read picture from FLASH
PEEK_FLASH PICT2, B$, 3840       ' read picture from FLASH

SCREEN$ = OR$(A$, B$)            ' combine 2 graphic areas by OR
PUT #12, #1, SCREEN$, 0,0,3840   ' show resulting graphic to LCD
END

```


AND\$, OR\$, XOR\$



4

GRAPHIC_MASK_COPY

GRAPHIC_MASK_COPY (*Str1*, *Str2*, *Mask*)

GRAPHIC_MASK_COPY: Vereinigt 2 Grafiken lt. Grafik-Make
(nur in Projektmodell 'PM_FULL')

Pixelgrafik. Kopiert die mit einer Maske ausgewählten Bildbestandteile einer Pixelgrafik in eine andere Pixelgrafik. Diese Funktion bearbeitet den Argumentstring **Str1**. Es findet keine Zuweisung statt.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Str1 ist gleichzeitig der Zielstring.
Str2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten, die (teilweise) in Str1 eingekopiert werden.
Mask	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und die Maskierungsdaten für den Kopiervorgang. Alle Bildelemente des Masken-Strings, die mit „1“ Bits markiert sind werden von B\$ nach A\$ kopiert.
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in Str1 werden direkt verändert.

Damit diese Funktion zu sinnvollen Ergebnissen führt ist es erforderlich, daß Größe und Format (Breite x Höhe) aller 3 Strings übereinstimmen.

Das Beispiel für GRAPHIC_MASK_COPY vereint 3 Pixelgrafiken - „2 Bilder“ und 1 Maske - auf zwei unterschiedliche Arten miteinander und zeigt sie wechselweise alle 600 ms auf dem Grafik-LCD an.

Programmbeispiel:

```

-----
' Name: GRAPHIC_MASK_COPY.TIG - PIXEL-Graphic: mask-copy
' demonstrates how different ways of masking appear on the LCD
-----
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
-----

USER VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING A$(4K), B$(4K), M$(4K) ' global strings: pixel data
DATALABEL STRI_A, STRI_B, STRI_M

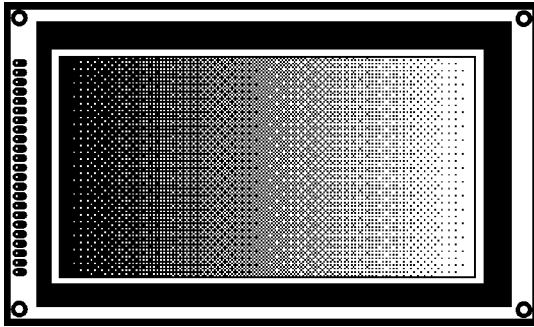
TASK MAIN
STRI_A::
  DATA FILTER "EINSTEIN.BMP", "GRAPHFLT", 0 ' Einstein
STRI_B::
  DATA FILTER "GREYRAMP.BMP", "GRAPHFLT", 0 ' grey ramp
STRI_M::
  DATA FILTER "MASKABC3.BMP", "GRAPHFLT", 0 ' mask

DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b         ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
PEEK_FLASH STRI_M, M$, 3820    ' <- get pixel graphic "mask"
LOOP 999999999                ' many loops
  LOOP 3
    PEEK_FLASH STRI_A, A$, 3820 ' <- get pixel graphic "A"
    PEEK_FLASH STRI_B, B$, 3820 ' <- get pixel graphic "B"
    PUT #12, #1, B$, 0,0,3840   ' show resulting graphic on LCD
    WAIT DURATION 600
    GRAPHIC_MASK_COPY (B$,A$,M$) ' do the masking process
    PUT #12, #1, B$, 0,0,3840   ' show resulting graphic on LCD
    WAIT DURATION 600
  ENDDLOOP
ENDLOOP

LOOP 3
  PEEK_FLASH STRI_A, A$, 3820 ' <- get pixel graphic "A"
  PEEK_FLASH STRI_B, B$, 3820 ' <- get pixel graphic "B"
  PUT #12, #1, A$, 0,0,3840   ' show resulting graphic on LCD
  WAIT DURATION 600
  GRAPHIC_MASK_COPY (A$,B$,M$) ' do the masking process
  PUT #12, #1, A$, 0,0,3840   ' show resulting graphic on LCD
  WAIT DURATION 600
ENDLOOP
ENDLOOP
END

```

4

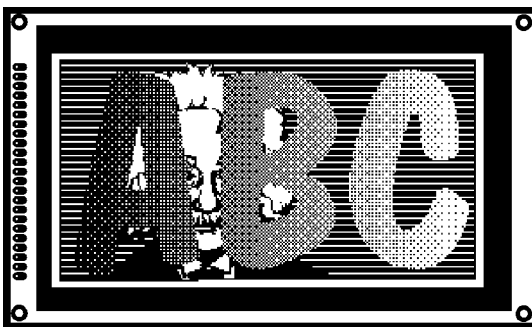
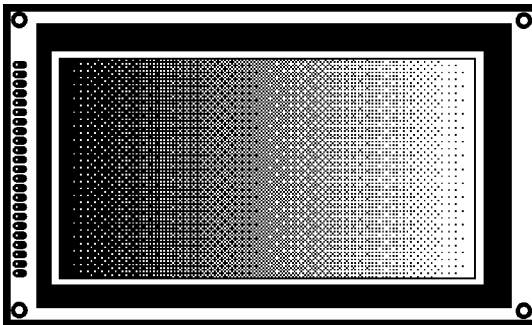


Im ersten Fall wird in einen „Graukeil“ (GREYRAMP.BMP) das mit der Maske (XMASK4.BMP) ausgestanzte Bild „Einstein“ (EINSTEIN.BMP) einkopiert:





Im zweiten Fall sind die beiden Bilder vertauscht: der ausgestanzte Graukeil wird in das Einstein-Bild einkopiert.



GRAPHIC_MIRROR

GRAPHIC_MIRROR (Src_Dest, FB, FH, Flag)

GRAPHIC_MIRROR: Spiegelt Pixel-Grafik an X- und/oder Y-Achse
(nur in Projektmodell 'PM_FULL')

Pixelgrafik: Spiegelt eine Grafik an X-Achse, Y-Achse oder X- und Y-Achse. Diese Funktion bearbeitet den Argumentstring String Str1. Es findet keine Zuweisung statt.

Src_Dest	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Str1 ist gleichzeitig der Zielstring.
FB	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formatbreite in Pixeln.
FH	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formathöhe in Pixeln:
Flag	Flag wählt gewünschte Spiegelungs-Operation aus: 1: X-Achse 2: Y-Achse 3: X+Y Achsen
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in Src_Dest werden direkt verändert.

Anwendungen:

- Ein Bildschirm wird durch einen Spiegel betrachtet. GRAPHIC_MIRROR erzeugt ein Spiegelbild, so daß der Betrachter wieder ein seitenrichtiges Bild sieht.
- Die Bildausgabe soll um 180-Grad gedreht werden können, z.B. für die Bedienung von 2 verschiedenen Positionen.

Das folgende Beispiel zeigt eine Pixelgrafik im Original und in den 3 gespiegelten Varianten. Dazu wird eine Pixelgrafik vom Dateisystem des PCs in den Data-Bereich des FLASH Speichers geladen.

Das Filter **GRAPHFLT** im DATA-Befehl wandelt das „BMP“-Format um in das interne Pixel-Format im Tiger. Beachten Sie, die Pfadangabe für dieses File in Ihrem System richtig zu setzen - entweder direkt hier im DATA Statement, oder in der Auswahl „OPTIONEN“ + „VERZEICHNISSE“.

Programmbeispiel:

```

-----
' Name: GRAPHIC_MIRROR.TIG - PIXEL-Graphic: Mirror graphic
' mirrors a pixel graphic - horizontal and vertical
-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
-----

USER VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING SCREEN$(4K), SCREEN2$(4K) ' global strings: pixel data
LONG N,K

DATALABEL PIX_DATA

TASK MAIN
PIX_DATA:
  DATA FILTER "HELLO.BMP", "GRAPHFLT", 0

  DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

  PEEK_FLASH PIX_DATA, SCREEN$, 3840 ' read Pixel graphic from FLASH
  LOOP 999999999
    FOR N=1 TO 3
      LOOP 2
        GRAPHIC_MIRROR (SCREEN$,240,128,N)' SRC_DST, wide, high, flg
        PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
        WAIT DURATION 500
      ENDOLOOP
    NEXT
  ENDOLOOP
END

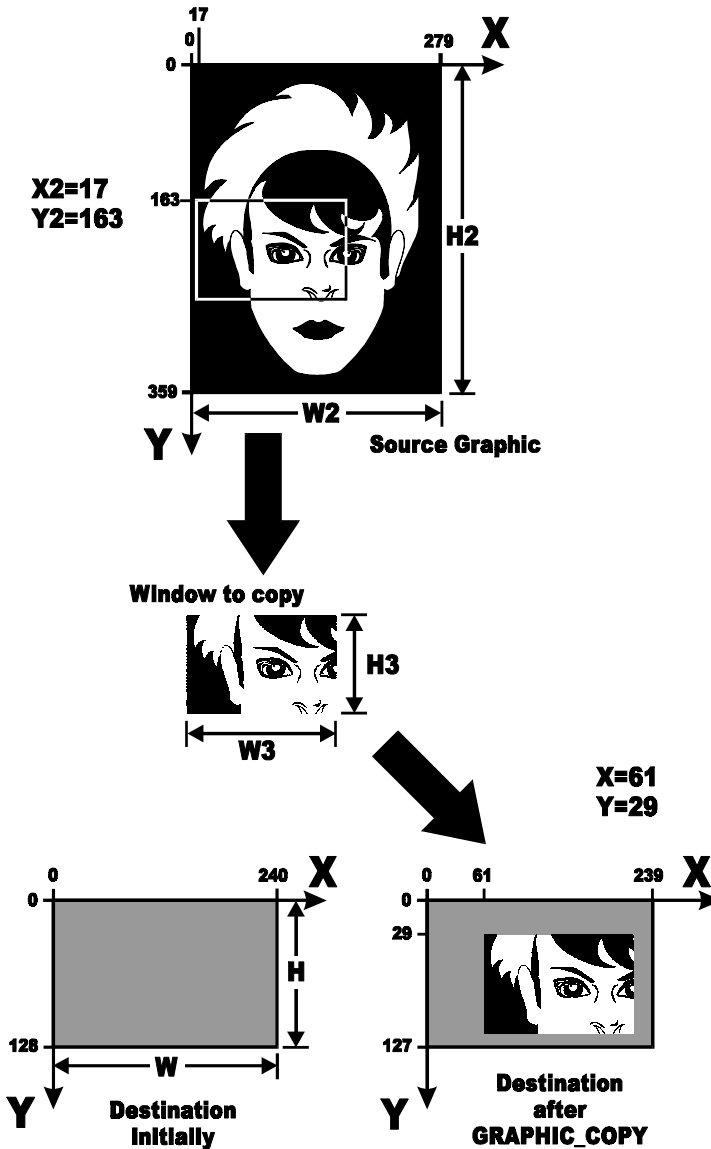
```

4



GRAPHIC_COPY

GRAPHIC_COPY (Str1, Str2, W, H, X, Y, W2, H2, X2, Y2, W3, H3, Mode)



GRAPHIC_COPY: Kopiert Fenster aus einer Grafik in andere Grafik
(nur in Projektmodell 'PM_FULL')

Kopiert ein Fenster mit Pixelgrafik von einem grafischen Gebiet in ein grafisches Gebiet. Diese Funktion bearbeitet den Argumentstring String **Str1**. Es findet keine Zuweisung statt.

- Str1** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Str1 ist gleichzeitig der Zielstring. Str1 enthält bereits vor Ausführung dieser Funktion ein grafisches Gebiet von der Ausdehnung B x H Pixeln. Durch die GRAPHIC_COPY Funktion wird ggf. ein Teil dieses grafischen Gebietes überschrieben.
- Str2** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und ist die Quellvariable. Str2 enthält ein grafischen Gebiet von der Ausdehnung: B2 x H2 Pixeln.
- W, H** sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die Formatabmessungen des Zielles: Breite, Höhe in Pixeln. Die Formatbreite ist stets ein ganzes Vielfaches von 8.
- X, Y** sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die Zielposition für das einkopierte Fenster, kartesische Koordinaten. Die Zielposition darf in allen 4 Quadranten liegen, also auch außerhalb des grafischen Zielgebietes. Ggf. wird nur der Teil des Fensters kopiert, der sich mit dem Zielgebiet überlappt.
- W2, H2** sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die Formatabmessungen des Quellgebietes: Breite, Höhe in Pixeln. Die Formatbreite ist stets ein ganzes Vielfaches von 8.
- X2, Y2** sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die Quellposition für das zu kopierende Fenster, kartesische Koordinaten. Die Quellposition darf in allen 4 Quadranten liegen, also auch außerhalb des grafischen Quellgebietes. Ggf. wird nur der Teil des Fensters kopiert, der sich mit dem Quellgebiet überlappt.

W3, H3

sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die Formatabmessungen des zu kopierenden Fensters. Diese Angaben dürfen frei gewählt, insbesondere dürfen:

- die Fensterbreite darf frei gewählt werden: 0,1,2, ... nnnn.
- die Fensterhöhe darf frei gewählt werden: 0,1,2, ... nnnn.
- die Fenstergröße darf auch größer sein als das Ziel- und/oder Quellgebiet.

Mode ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und definiert den Kopiermodus:

- 0: COPY
- 1: OR Überlagerung. „Schwarze“ Pixel werden addiert.
- 2: AND Überlagerung: „Weiße“ Pixel werden addiert.
- 3: XOR Überlagerung: „1“ Pixel invertieren Farbwert im Zielgebiet

Funktionswert gibt es bei dieser Funktion nicht. Die Daten in **Str1** werden direkt verändert.

Beachte: Ziel- und Quellgrafiken müssen bereits vor Ausführung dieser Funktion (mindestens) mit der entsprechenden Anzahl von Bytes gefüllt sein, die sich aus den Formatangaben ergeben: $B \times H / 8$, bzw. $B2 \times H2 / 8$.

GRAPHIC_COPY ist eine sehr nützliche und in Pixelgrafik häufig eingesetzte Funktion. Sie arbeitet schnell, ist einfach zu handhaben und sehr flexibel. Die nachfolgenden Beispiele zeigen einige Möglichkeiten. Insbesondere kann man mit GRAPHIC_COPY:

- stehende oder bewegte Bilder in eine Grafik einkopieren,
- Ausschnitte aus großen Grafiken darstellen und diese verschiebbar machen,
- große Ziffern und Buchstaben, große Texte einblenden
- Firmen-Logos und Sonderzeichen einkopieren
- Symbole frei über den Bildschirm wandern lassen (Fadenkreuz, Cursor und andere Marker)
- Animationen und Prozeßvisualisierungen realisieren.

Bei der Benutzung von GRAPHIC_COPY ist man vollkommen frei bei der Wahl der Verhältnisse, insbesondere darf man:

- Fenstergröße beliebig wählen, auch wenn diese größer ist als das Quell- oder Zielgebiet,
- Die Fensterposition im Quellgebiet darf im Quellgebiet oder auch teilweise oder ganz außerhalb des Quellgebietes liegen,
- Die Fensterposition im Zielgebiet darf im Zielgebiet oder auch teilweise oder ganz außerhalb des Zielgebietes liegen
- Quell- und Zielgebiet können verschiedene Abmessungen besitzen.

In allen Fällen ermittelt GRAPHIC_COPY die richtigen Positionen und Fenstergrößen, so daß ein plausibles Ergebnis zu stande kommt. Die Beispiele zeigen diese Eigenschaften exemplarisch.

Durchrollende Ziffer

In diesem Beispiel wird eine endlos durchrollende Ziffernstelle in ein Bild eingebildet. Die Ziffern: 0...9 sowie noch eine 0 liegen hier in einem Bitmap-File vor:



4

Durch diese Art der Anordnung kann mit GRAPHIC_COPY sehr einfach auf die gewünschte Ziffer zugegriffen werden. Die Y-Quellposition einer Ziffer mit dem Wert N bestimmt sich dann aus dem Zusammenhang:

$$\mathbf{Y-POS = N * (Y-Rasterweite)}$$

Es ist sehr einfach einen solchen Bitmap-File mit Ziffern, Buchstaben oder anderen Symbolen zu erstellen. Grafik- und DTP-Programme bieten meist sofort eine Export-Funktion nach „*.BMP“ an. Man wähle dann die Einstellung: „schwarz-weiß“, achte auf die Breite von ganzen Vielfachen von 8 und entscheide sich ggf. für eine „Strichspeicherung“ oder „Rasterspeicherung“.

Große Zahlenanzeigen und Großtexte können dann in beliebigen Schriftarten, Größen und Layouts eingesetzt werden.

```

Name: GRAPHIC_COPY1.TIG      -      PIXEL-Graphic:  graphic-window-copy
-----
shows a "rolling" number display
Notes: set correct path for flash data (OPTIONS)
       connect L71 to reset LCD
-----

USER VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING SCREEN$(4K)             ' global strings: pixel data
LONG Y2

DATALABEL PICT1, PICT2

TASK MAIN
PICT1::
  DATA FILTER "NUMB_01.BMP", "GRAPHFLT", 0 ' Numbers - 80 pixel wide
PICT2::
  DATA FILTER "EINSTEIN.BMP", "GRAPHFLT", 0 ' Einstein

DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0  ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0     ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255    ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

PEEK_FLASH PICT2, SCREEN$, 3820 ' <- get pixel graphic
Y2=0                             ' access-position for numbers

LOOP 999999999                 ' many loops
  Y2 = MODULO_INC (Y2, 0, 1048, 2)

'                               Format      Format      Window
'                               <->        <--->        <--->
'                               DESTIN, SRC,  B,H,      X, Y,  B2,H2, X2,Y2,B3,H3,MODE
'                               GRAPHIC_COPY (SCREEN$, PICT1,240,128,150,20,80,1500,0,Y2, 70,80, 0)
'                               <---DEST---> <-----SOURCE----->

  PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic on LCD
ENDLOOP
END

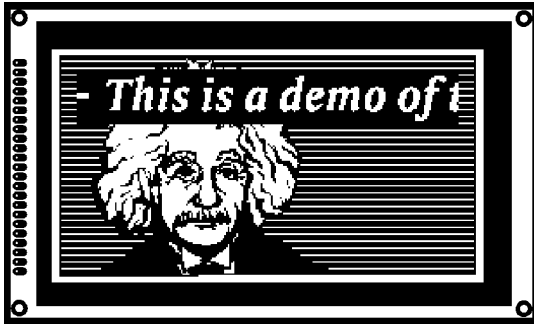
```



Spruchband

In diesem Beispiel wird eine waagerechtes, laufendes Spruchband in ein Standbild eingeblendet:

4



Programmbeispiel:

```

-----
' Name: BANNER.TIG - PIXEL-Graphic: graphic-window-copy
' shows a sliding message banner
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
-----

USER_VAR STRICT
#include DEFINE_A.INC          ' general defines
#include UFUNC3.INC           ' defines user function codes
#include LCD_4.INC            ' defines LCD Typ 4
#include GR_TK1.INC           ' defines Graphic Toolkit

STRING SCREEN$(4K)            ' Global Strings for Pixel-data
LONG X2
DATALABEL PICT1, PICT2

TASK MAIN
PICT1::
  DATA FILTER "BANNER_1.BMP", "GRAPHFLT", 0 ' Text Banner: 1200x32(wxh)
PICT2::
  DATA FILTER "EINSTEIN.BMP", "GRAPHFLT", 0 ' Einstein

  DIR PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
  INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

  PEEK FLASH PICT2, SCREEN$, 3820 ' <- get pixel graphic
  PUT #12, #1, SCREEN$, 0,0,3840  ' show resulting graphic to LCD
  X2=0                            ' access-position for numbers

  LOOP 999999999               ' many loops
    X2 = MODULO_INC (X2, 0, 852, 1)

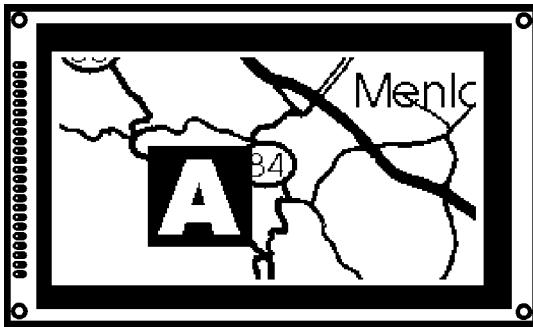
'                               Format      Format      Window
'                               <->        <---->      <---->
'                               B,H,      X,Y,      B2,H2,X2,Y2,B3,H3,MODE
  GRAPHIC_COPY (SCREEN$, PICT1,240,128,10,8,1200,32,X2,0, 220,32, 0)
'                               <---DEST---> <-----SOURCE----->
  PUT #12,#1,SCREEN$,300,300,960 ' SRC, LCD_offs, SRC_offs, SRC_len
  WAIT DURATION 30
ENDLOOP
END

```

Bewegte Straßenkarte und bewegte Box

In diesem Beispiel wird aus einer größeren Grafik (Straßenkarte) ein Ausschnitt in der Größe des Bildschirms entnommen und angezeigt. Das Quellfenster wird in einem Loop schrittweise verschoben, so daß sich der Bildschirm über der Karte hinwegzubewegen scheint. Zusätzlich wird eine rechteckige, bewegte Box mit wechselndem Inhalt einkopiert.

4



Programmbeispiel:

```

Name: GRAPHIC_COPY3.TIG - PIXEL-Graphic: graphic-window-copy
-----
the LCD serves as a moving window over a street map
+ a character box moving inside
Notes: set correct path for flash data (OPTIONS)
connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K)            ' global strings: pixel data
LONG X2,Y2,YYA,XA,YA
DATALABEL PICT1, PICT2

TASK MAIN
PICT1::
DATA FILTER "A2Z.BMP", "GRAPHFLT", 0' Letters: A...Z 64 x 1526 (wxh)
PICT2::
DATA FILTER "CITYMAP.BMP", "GRAPHFLT", 0' City map: 880x480 (w x h)

DIR_PORT 8,1000000b           ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b         ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

X2=MODULO_UPDO (50, 0, 880-240, 1, 0) ' Value, Min, Max, Step, ID
Y2=MODULO_UPDO (50, 0, 480-128, 1, 1) ' Value, Min, Max, Step, ID
XA=MODULO_UPDO (50, 0, 240-60, 1,2)   ' Value, Min, Max, Step, ID
YA=MODULO_UPDO (50, 0, 128-58, 1,3)   ' Value, Min, Max, Step, ID
YYA=300                                '
SCREEN$ = FILL$("<0>",4K)              ' graphic string (dummy bytes)

LOOP 999999999                ' many loops
LOOP 15

      Format               Format       Window
      <->                 <--->        <--->
      DESTIN, SRC,        B,H,    X,Y,    B2,H2, X2,Y2, B3,H3, MODE
GRAPHIC_COPY &
(SCREEN$, PICT2, 240,128, 0,0, 880,480, X2,Y2, 240,128,0)
GRAPHIC_COPY &
(SCREEN$, PICT1, 240,128,XA,YA, 64,1526, 0,YYA/100,60,58, 0)
<----DEST----> <-----SOURCE----->
PUT #12, #1, SCREEN$, 0,0,3840      ' resulting graphic on LCD
X2 = MODULO_UPDO (X2,0)              ' advance Y-Position
Y2 = MODULO_UPDO (Y2,1)              '
XA = MODULO_UPDO (XA,2)              ' advance Positions
YA = MODULO_UPDO (YA,3)              '
WAIT DURATION 20                     ' just a moment
ENDLOOP
YYA= MODULO_INC (YYA,300-5804,25*5804+300, 5804)

```

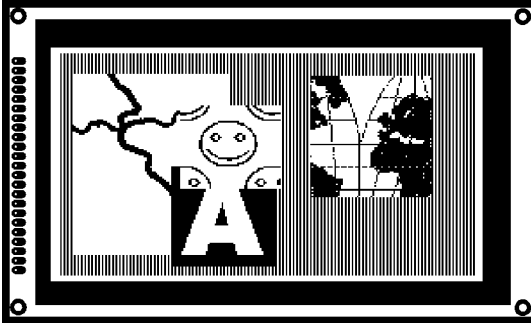
```
ENDLOOP          ' advance YYA-Position  
END
```

4

Statt des eingeblendeten ‚A‘ in dem vorigen Beispiel ist ein Fadenkreuz denkbar, welches die Position anzeigt, an der man sich befindet. Die dazu notwendigen Daten kommen von einem GPS-Empfänger. Die Begrenzungskordinaten der Karte in geografischer Breite und Länge werden von den absoluten Positionsdaten des GPS abgezogen, so daß die relative Position auf der Karte bekannt ist. Als letzter Rechenschritt wird die relative Position von Graden in Pixel umgerechnet. Da die Daten der Karte im Flash gespeichert sind, bestehen gewisse Einschränkungen bezüglich der Auflösung und/oder der Größe des Kartenausschnitts. Auch das hier verwendete Display setzt mit 240x128 Pixeln nur in schwarz-weiß enge Grenzen. Als low-cost-Navigations-Anzeige ist es jedoch durchaus sinnvoll zu realisieren.

Mehrere Fenster auf einem Bildschirm

GRAPHIC_COPY wird auch eingesetzt um Fenster mit verschiedenem Inhalt auf dem Bildschirm zu bringen. Die Fenster können bei Bedarf natürlich auch verschoben werden, in Breite und Höhe verändert werden und Fenster dürfen sich auch überlappen.



4

Programmbeispiel:

4

```

'-----
' Name: GRAPHIC_COPY4.TIG - PIXEL-Graphic: graphic-window-copy
'-----
' shows on graphic screen several windows, some of them moving
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data
LONG X2,Y2,YA,XA,YA
DATALABEL PICT1, PICT2, PICT3, PICT4

TASK MAIN
PICT1::
DATA FILTER "A2Z.BMP", "GRAPHFLT", 0 ' A...Z, 64x1526) =win-1
PICT2::
DATA FILTER "CITYMAP.BMP", "GRAPHFLT", 0 ' City map: 880x480 =win-2
PICT3::
DATA FILTER "SMILIES.BMP", "GRAPHFLT", 0 ' smilie 240x128 =win-3
PICT4::
DATA FILTER "WORLD.BMP", "GRAPHFLT", 0 ' world map 240x128 =win-4

DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

X2=MODULO_UPDO (50, 0, 880-240, 1, 0) ' Value, Min, Max, Step, ID
Y2=MODULO_UPDO (50, 0, 480-128, 1, 1) ' Value, Min, Max, Step, ID
XA=MODULO_UPDO (50, 0, 240-60, 1, 2) ' Value, Min, Max, Step, ID
YA=MODULO_UPDO (50, 0, 128-58, 1, 3) ' Value, Min, Max, Step, ID
YYA=300

LOOP 999999999 ' many loops
LOOP 15
SCREEN$ = FILL$("<0FH>",4K) ' fill graphic string background

'
' format Format Window
' <--> <---> <--->
' DESTIN, SRC, B,H, X,Y, B2,H2, X2,Y2, B3,H3,MOD
GRAPHIC_COPY &
(SCREEN$,PICT2,240,128, 8,12, 880,480, X2,Y2, 90,104,0)
GRAPHIC_COPY &
(SCREEN$,PICT1,240,128, XA,YA, 64,1526, 0,YYA/100,60,58,0)
GRAPHIC_COPY &
(SCREEN$,PICT3,240,128, 70,30, 240,128, 1, 1,58,48,0)
GRAPHIC_COPY &
(SCREEN$,PICT4,240,128, 210-XA,78-YA,240,128,210-XA,78-YA,70,50,0)
' <-----DEST-----> <-----SOURCE----->

```

```
PUT #12, #1, SCREEN$, 0,0,3840      ' show resulting graphic to LCD
X2 = MODULO_UPDO (X2,0)              ' advance Y-Position
Y2 = MODULO_UPDO (Y2,1)              '
XA = MODULO_UPDO (XA,2)              ' advance Positions
YA = MODULO_UPDO (YA,3)              '
WAIT DURATION 50                     ' just a moment
ENDLOOP                              '
YYA= MODULO_INC(YYA,300-5804,25*5804+300,5804) ' advance YYA-Pos
ENDLOOP                              ' (next letter)
END
```

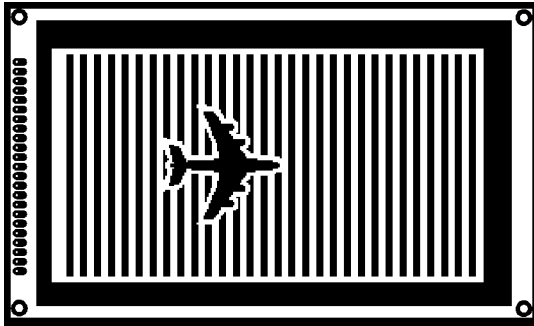
Einkopieren eines Cursors mit beliebiger Kontur

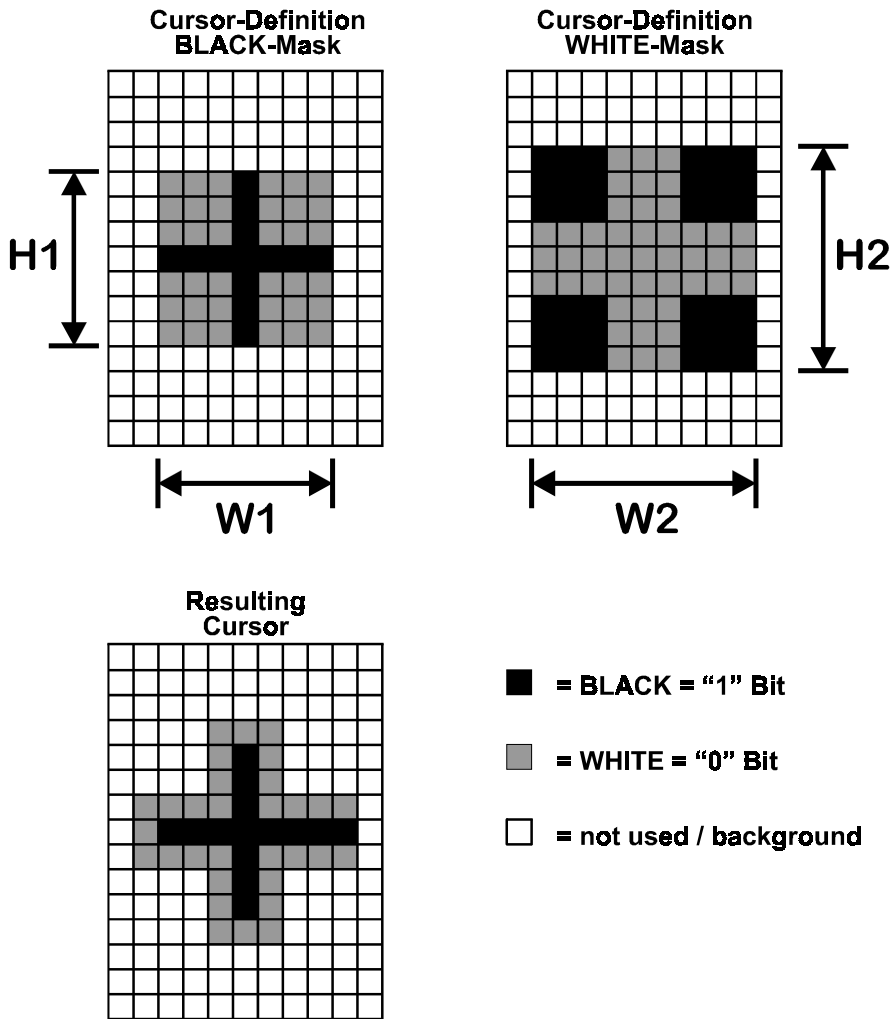
Für Anwendungen wie das Darstellen eines Cursors möchte man frei sein in der Gestaltung von Form und Aussehen. Auch soll das Cursor-Symbol sich gut vom Hintergrund abheben.

GRAPHIC_COPY kopiert jedoch wie gezeigt nur Rechtecke - nicht jedoch anders geformte Konturen. Dennoch lassen sich beliebig geformte Konturen, mit oder ohne Umrandung, frei verschieblich in eine Grafik einkopieren. Dies geschieht mit Hilfe von Masken und der Verwendung von AND und OR Überlagerungen.

Nachfolgende Grafik zeigt das grundsätzliche Vorgehen an Hand eines Kreuzes, das mit einem weißen Rand umgeben in eine Zielgrafik kopiert wird. Das ausgeführte Beispiel verwendet als Cursor-Symbol ein Flugzeug (auch mit weißem Rand).

4

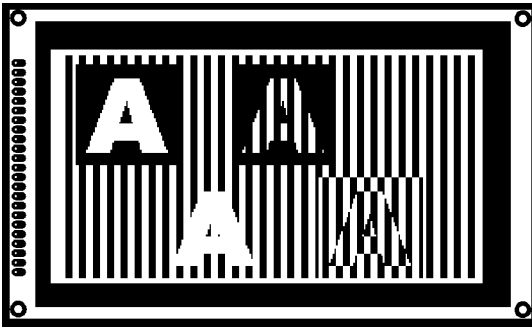




Grafische Fenster kopieren mit verschiedenen Überlagerungsarten

Dieses Beispiel zeigt die verschiedenen Überlagerungs-Möglichkeiten bei der GRAPHIC_COPY Funktion, die durch den „MODE“-Parameter gewählt werden:

- 0 = PUNCH
- 1 = OR
- 2 = AND
- 3 = XOR



```
' Name: GRAPHIC_COPY6.TIG - PIXEL-Graphic: graphic-window-copy
-----
' shows 4 copy methods: punch, or, and, xor
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
-----

USER_VAR STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data
LONG YYA '
DATALABEL PICT1 '

TASK MAIN '
PICT1::
DATA FILTER "A2Z.BMP", "GRAPHFLT", 0 ' A...Z, 64x1526

DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

YYA=300 '
LOOP 999999999 ' many loops
SCREEN$ = FILL$(" <0FH>",4K) ' fill graphic string background

' Format Window
' <-> <---> <--->
' DESTIN,SRC, B,H, X,Y, B2,H2, X2,Y2, B3,H3,M
GRAPHIC_COPY(SCREEN$,PICT1,240,128, 10, 5,64,1526,0,YYA/100,60,58,0)
GRAPHIC_COPY(SCREEN$,PICT1,240,128,100, 5,64,1526,0,YYA/100,60,58,1)
GRAPHIC_COPY(SCREEN$,PICT1,240,128, 60,70,64,1526,0,YYA/100,60,58,2)
GRAPHIC_COPY(SCREEN$,PICT1,240,128,150,70,64,1526,0,YYA/100,60,58,3)
' <----DEST----> <-----SOURCE----->

PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
WAIT DURATION 600 '
YYA= MODULO_INC(YYA,300-5804,25*5804+300,5804)' advance YYA-Position
ENDLOOP
END
```

INVERT

INVERT (*Str1*, *Offset*, *Len*)

INVERT: Invertiert Anzahl Bytes in einem String
(nur in Projektmodell 'PM_FULL')

Invertiert alle Bytes in **Str1**. Diese Funktion bearbeitet den Argumentstring String **Str1**. Es findet keine Zuweisung statt.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Str1 ist gleichzeitig der Zielstring.
Offset	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Start-Offset im String an. Angabe in Bytes. Bei Offset = 0 wird Source ab dem 1. Zeichen genommen.
Len	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Länge in Bytes, die ab OFFSET für die Inversion genommen werden soll
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in Str1 werden direkt verändert.

Die INVERT Funktion wird z.B. eingesetzt um Pixelgrafiken in ihrem Farbwert umzukehren. Das folgende Beispiel zeigt die Wirkung.

4

Invertiertes Bild:



4

Wahres Bild:



GRAPHIC_FILL_MASK

GRAPHIC_FILL_MASK (Str1, FB, FH, X, Y, B2, H2, Colour)

GRAPHIC_FILL_MASK: Baut rechteckige Maske in Grafik-String auf (nur in Projektmodell 'PM_FULL')

Pixelgrafik: Füllt eine rechteckige-Maske in ein grafisches Gebiet. Diese Funktion bearbeitet den Argumentstring String **Str1**. Es findet keine Zuweisung statt.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Str1 ist gleichzeitig der Zielstring.
FB	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formatbreite in Pixeln.
FH	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formathöhe in Pixeln:
FB2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formatbreite in Pixeln.
FH2	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formathöhe in Pixeln:
X, Y	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und Position der Maske in der Zielgrafik (kartesisch): -8191...0...8191. Angegeben wird die linke obere Ecke der Maske
B2, H2	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und geben die Breite und Höhe der Maske an, jeweils: 0...nnnn Pixel
Colour	Legt fest, ob weiße Punkte (0) oder schwarze Punkte (X) in die Maske gefüllt werden.
Funktionswert	gibt es bei dieser Funktion nicht. Die Daten in Str1 werden direkt verändert.

Das Beispielprogramm zeigt die Benutzung von GRAPHIC_FILL_MASK indem unterschiedliche Masken vor einen gestreiften Hintergrund geschrieben werden.

```
Name: GRAPHIC_FILL_MASK.TIG PIXEL-Graphic: fill mask pixel-graphic displays square fill areas with random size at random position
```

```
GRAPHIC_FILL_MASK    (DEST, W,H, X,Y,   W2, H2, FARBE)
<---Destin----> <-----Mask----->
```



```
<-W--(bytes)->
-X->
! : <0,0>.....
! : .....
! Y .....
! : .....,X,Y>....
! : .....MMMMMMMMMM.....      : 
! .....MMMMMMMMMM.....        : H2
H .....MMMMMMMMMM.....          : 
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
! .....
!
```

```
' Notes: set correct path for flash data (OPTIONS)
connect L71 to reset LCD'
```

```
USER_VAR_STRICT
#include DEFINE_A.INC           ' general definitions
#include UFUNC3.INC             ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_Tk1.INC            ' definitions for Graphic Toolkit
```



```
STRING SCREEN$(K)               ' Global Strings for Pixel-data
LONG N, X, Y, W, H, COL
```



```
TASK MAIN
DIR PORT 8,1000000b                ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b              ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0     ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255   ' "
OUT P_LCD_CTRL,M_INVERT,255       ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0       ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255      ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
```



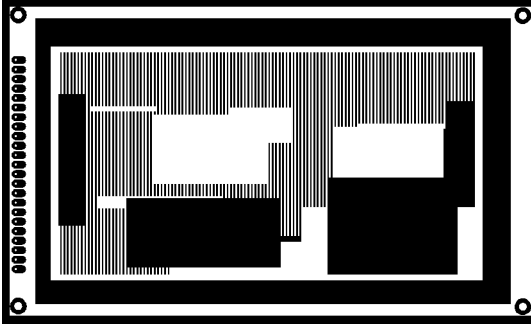
```
LOOP 999999999                   ' many loops
SCREEN$=FILL$("<55H">",3840)      ' Fill 00-Bytes into graphic
FOR N=1 TO 55                     ' 
  X =(MOD(RND(N*TICKS()),240))-10 ' X-Pos
  Y =(MOD(RND(N*TICKS()+1),128))-10 ' Y-POS
  W =(MOD(RND(N*TICKS()+2),130))    ' wide
  H =(MOD(RND(N*TICKS()+3),80))     ' high
  COL=(MOD(RND(N),2))              ' colors: 0/1
```



```
'                                DESTIN   wide,high         colour
GRAPHIC FILL MASK (SCREEN$, 240,128, X.Y, W,H, COL) '
```


GRAPHIC_FILL_MASK

```
PUT #12, #1, SCREEN$, 0,0,3840      ' show graphic on LCD
WAIT DURATION 100
NEXT
ENDLOOP
END
```



4

Das 2. Beispiel zeigt die Anwendung von GRAPHIC_FILL_MASK für die Erzeugung von Rahmen um Fenster oder Bilder.

Programmbeispiel:

4

```

'-----
' Name: GRAPHIC_FILLM2.TIG - PIXEL-Graphic: fill mask in pixel-graphic
' shows how to use GRAPHIC_MASK_COPY to create frames around pictures
'-----
' Notes: set correct path for flash data (OPTIONS)
'        connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING SCREEN$(4K)             ' Global Strings for Pixel-data
DATALABEL PICT1

TASK MAIN
PICT1::
  DATA FILTER "MOZART1.BMP", "GRAPHFLT", 0 ' Mozart

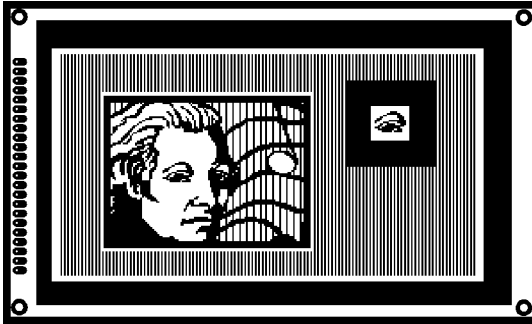
  DIR_PORT 8,1000000b           ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b         ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
  SCREEN$=FILL$("<55H">",4K)    ' fill background stripes

' DESTIN wide,high  X,      Y      W,H  colour
GRAPHIC_FILL_MASK &
(SCREEN$, 240,128, 30-6,28 -6,111+12,80+12, 0) ' white (1)
GRAPHIC_FILL_MASK &
(SCREEN$, 240,128, 30-4,28 -4,111 +8,80 +8, 1) ' black (1)
GRAPHIC_FILL_MASK &
(SCREEN$, 240,128,180-15,30-15, 22+30,20+30, 1) ' black (2)

'
'          Format          Format          Window
'          <->           <--->          <--->
' DESTIN, SRC,  B,H,      X,Y,  B2,H2,  X2,Y2,  B3,H3, MODE
GRAPHIC_COPY &
(SCREEN$,PICT1, 240,128, 30,28, 240,128, 70, 5, 111,80, 0)
GRAPHIC_COPY &
(SCREEN$,PICT1, 240,128, 180,30, 240,128, 98,36, 22,20, 0)
'          <-----DEST----> <-----SOURCE----->

PUT #12, #1, SCREEN$, 0,0,3840 ' show resulting graphic to LCD
END

```



Dieses Beispiel zeigt eine kurze Low-Resolution Video-Sequenz. Die Bilder der Sequenz liegen einzeln als BMP-Files auf Disk vor. Das Video-Signal wurde digitalisiert und per Dithering an die rein schwarz/weiße Pixelgrafik angepaßt.

Programmbeispiel:

```

'-----
' Name: MOVIE1.TIG - PIXEL-Graphic: shows a short low-res movie
' shows a short movie
'-----
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K),A$(31K) ' Global Strings for Pixel-data
LONG ADR, PLEN
DATALABEL PICT1, PICT2

TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
PLEN=PICT2-PICT1 ' Picture-Len

LOOP 999999999 ' many loops
FOR ADR=PICT1 TO PLEN*32 STEP PLEN '
PUT #12, #1, ADR, 0,0,3840 ' show resulting graphic on LCD
WAIT_DURATION 70 '
NEXT
FOR ADR=PLEN*33 TO PICT1+PLEN STEP -PLEN'
PUT #12, #1, ADR, 0,0,3840 ' show resulting graphic to LCD
WAIT_DURATION 70 '
NEXT
ENDLOOP

PICT1::
DATA FILTER "HELLO1.BMP", "GRAPHFLT", 0' Picture of the movie
PICT2::
DATA FILTER "HELLO2.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO3.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO4.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO5.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO6.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO7.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO8.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO9.BMP", "GRAPHFLT", 0' Picture of the movie

DATA FILTER "HELLO10.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO11.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO12.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO13.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO14.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO15.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO16.BMP", "GRAPHFLT", 0' Picture of the movie

```

4

```
DATA FILTER "HELLO17.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO18.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO19.BMP", "GRAPHFLT", 0' Picture of the movie

DATA FILTER "HELLO20.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO21.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO22.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO23.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO24.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO25.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO26.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO27.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO28.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO29.BMP", "GRAPHFLT", 0' Picture of the movie

DATA FILTER "HELLO30.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO31.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO32.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO33.BMP", "GRAPHFLT", 0' Picture of the movie
DATA FILTER "HELLO34.BMP", "GRAPHFLT", 0' Picture of the movie

END
```





4

GRAPHIC_EXP\$

GRAPHIC_EXP\$ (*Str1*, *Breit1*, *Len2*, *Breit2*, *Pix2*, *SHL2*)

GRAPHIC_EXP\$: Bit-Expansion zur Anpassung an Ausgabe-Geräte
(nur in Projektmodell 'PM_FULL')

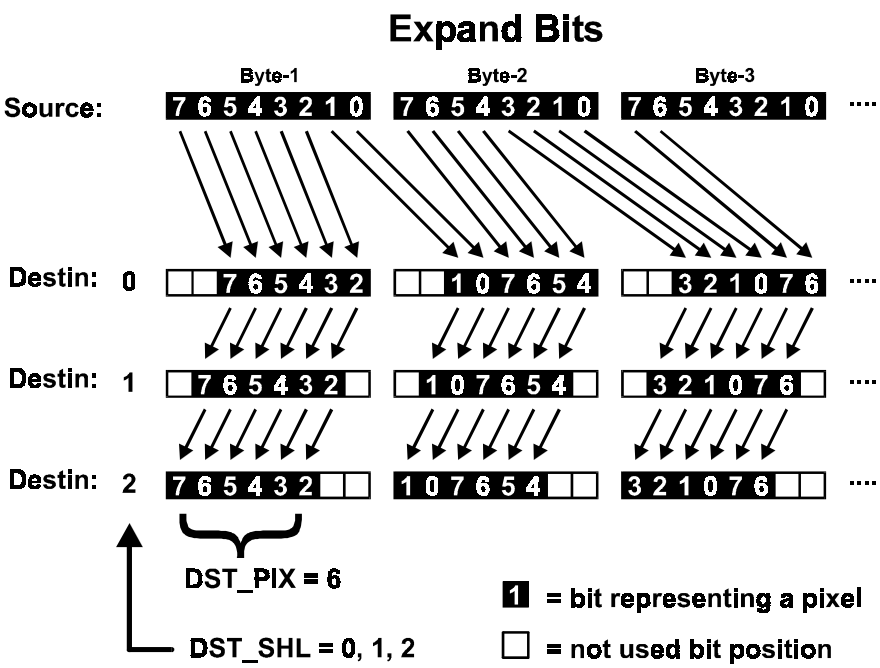
Verteilt die Pixel einer Pixelgrafik auf eine größere Zahl von Bytes.

Str1	ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Grafikdaten. Es befinden sich 8 Pixel pro Byte in dieser Darstellung.
Breit1	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Quellbreite in Anzahl Bytes .
Len2	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Ziellänge insgesamt in Bytes.
Breit2	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Zielbreite in Anzahl Bytes .
Pix2	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Anzahl der Pixel pro Byte im Zielstring: 1...8.
SHL2	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Anzahl der Bit-Schiebe-Operationen in jedem Byte (shift-left): 0...7.
Funktionswert	ist vom Typ STRING und enthält den konvertierten String.

Normalerweise befinden sich jedem Byte genau 8 Pixel einer Pixelgrafik. Es gibt jedoch Ausgabegeräte, zum Beispiel LC-Displays, die nur 7, 6 oder weniger Pixel pro Byte darstellen. Um Pixelgrafiken an solche Ausgabegeräte anzupassen existiert die GRAPHIC_EXP\$ Funktion.

Eine in SRC\$ vorliegende Pixelgrafik wird entsprechend der vorgegebenen Parameter in senkrechte „Streifen“ geschnitten, die dem geforderten Format des Ausgabegerätes entsprechen. GRAPHIC_EXP\$ wird daher erst als letzte Grafik-Operation - unmittelbar vor der Ausgabe - angewendet.

Die folgende Abbildung verdeutlicht den Vorgang.



4

Das Beispielprogramm zeigt, wie GRAPHIC_EXP\$ eine Grafik in immer dünnere Streifen zerlegt. Es werden alle möglichen Werte für Anzahl der Pixel pro Byte im Zielstring (DST_PIX) = 1...8 durchlaufen.

Programmbeispiel:

4

```

-----
' Name: GRAPHIC_EXP$.TIG - PIXEL-Graphic: Expand Graphic
-----
' shows how to expand from 8 pix per byte to 1...7 pix per byte
' Notes: set correct path for flash data (OPTIONS)
' connect L71 to reset LCD
-----

'
'   ERG$ <- GRAPHIC_EXP$ (SRC$, SRC_BREIT,
'                         DST_LEN, DST_BREIT, DST_PIX, DST_SHL)
'
'
' expand Graphic Pixel in a String to 1...8 Pixel per Byte.
' supports the 6-Bit mode of a LCD:
' expand from 8 pixel bits to 6 pixel bits per byte
'
'
'   SRC$ = source string / source data in FLASH
' SRC_BREIT = number of B y t e s width in source
' DST_LEN = number of B y t e s total length in destination
' DST_WIDTH = width of destination strings in B y t e s
' DST_PIX = number of pixel per byte: 1...8
' DST_SHL = shift-lefts inside the destination bytes
' RESULT$ = result / destination string
'
' example:
'   Source width = 2 bytes
'   Destination width = 5 bytes --> destination = 5 bytes wide
'   Destin-PIX = 5 --> 5 pixels in destination bytes
'   DST_SHL = 1 --> shift 1 bit LEFT
'
'
'   76543210 76543210 76543210 76543210 76543210
' SRC$: abcdefgh ijklmnop
' RLT$: ..abcde. ..fghij. ..klmno. ..p----. ++++++++
'
'   . = 0
'   - = 0
'   + = unchanged
'
-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K),A$(31K) ' global strings: pixel data
LONG BITS
DATALABEL PICT1

TASK MAIN
PICT1::
    DATA FILTER "EINSTEIN.BMP", "GRAPHFLT", 0 ' Einstein

    DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
    DIR_PORT 7,11111101b ' L71 output: LCD reset
    OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD

```

```

OUT P_LCD_RESET,M_LCD_RESET,255      '      "
OUT P_LCD_CTRL,M_INVERT,255           ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0           ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255          ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

LOOP 999999999                        ' many loops
FOR BITS=8 TO 1 STEP -1               '
  PEEK_FLASH PICT1, SCREEN$, 3820     ' <- get pixel graphic

'                                     SRC, SRC_WIDE,          DST_LEN,
A$=GRAPHIC_EXP$ (SCREEN$,              30, (2400/BITS+9)/10*128,&
(2400/BITS+9)/10,                      BITS,          0)
'                                     DST_WIDE, DST_PIX, DST_SHL)

'                                     Format          Format      Window
'                                     <->           <---->      <--->
'       DESTIN, SRC, W,H,  X,Y,          B2,H2,X2,Y2, B3,H3,MOD
GRAPHIC_COPY &
(SCREEN$,A$,240,128,0,0,8*(2400/BITS+9)/10,128, 0,0, 240,128,0)
'       <---DEST--> <-----SOURCE----->

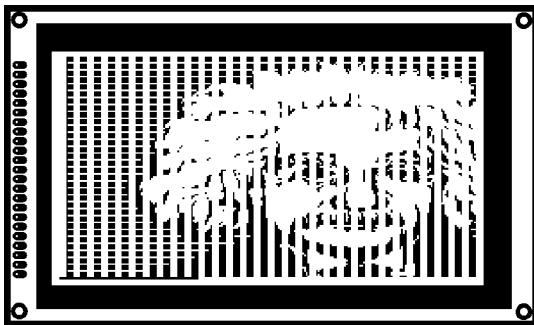
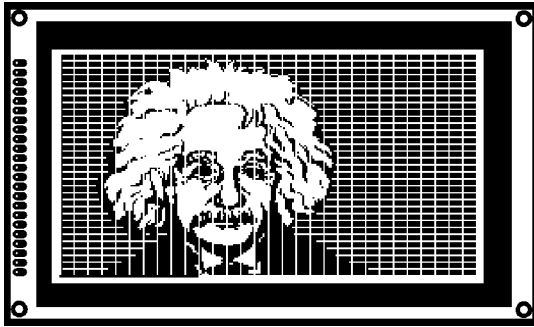
PUT #12, #1, SCREEN$, 0,0,3840        ' show resulting graphic to LCD
WAIT DURATION 600
NEXT
ENDLOOP
END

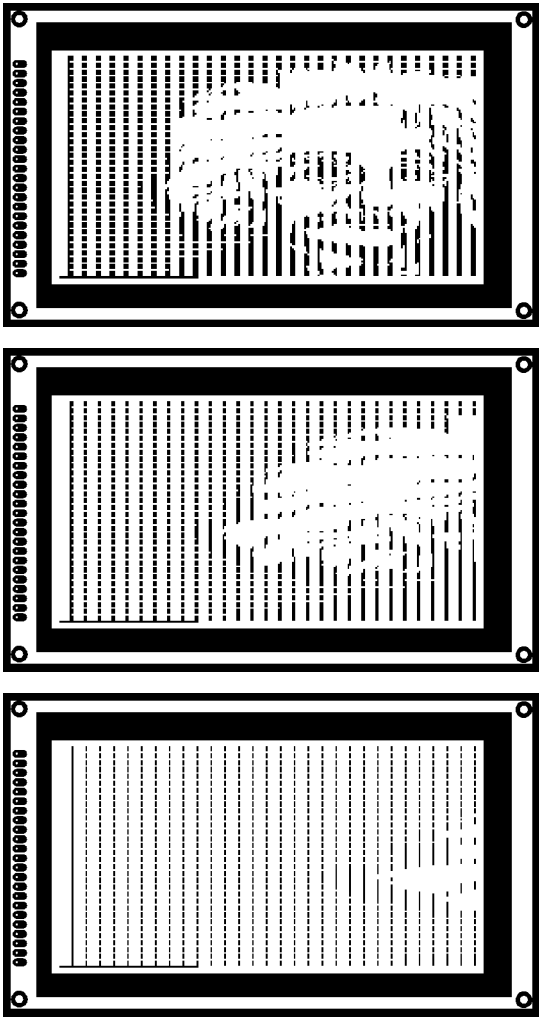
```

4



4





4

Arbeiten mit Vektorgrafik

Die Grundlagen zur Vektorgrafik finden Sie weiter vorn im Handbuch bei der Gegenüberstellung zur Pixelgrafik: Seiten ab 560.

Im nachfolgenden Abschnitt werden die Vektorgrafikfunktionen im Einzelnen vorgestellt.

Übersicht über die Vektorgrafikfunktionen:

Funktionsname	
DRAW_LINE	Beginnt Linie in Grafikbereich
DRAW_NEXT_LINE	Setzt Linie in Grafikbereich fort
CLOSE_LINE	Schließt Linie
SET_ROTATION	Setzt Rotationswinkel 0,01-Grad
SET_SCALE	Setzt Maßstab (100% = 1000)
SET_BASE	Setzt Basispunkt X/Y
SET_GRAREA	Setzt Grafikbereich: Destin + B x H
SET_DOT	Setzt DOT zu 0 or 1
FILL_AREA	Füllt Grafikbereich

DRAW_LINE

DRAW_LINE (Dest, FB, FH, X0, Y0, X1, Y1, X2, Y2, XScale, YScale, Rotation, Mode, Pen)

DRAW_LINE: Zeichnet Linie incl. Linienanfang
(nur in Projektmodell 'PM_FULL')

Vektorgrafik: DRAW_LINE beginnt eine Linie im Pixelfeld und legt einen Parametersatz unter einer ID-Nummer ab (Pen-Nr=Stift-Nr.). Andere Funktionen können unter Verwendung dieser Pen-Nummer aufgerufen werden, ohne daß in jedem Funktion der gesamte Parametersatz mit angegeben werden muß. Diese Funktion bearbeitet den Argumentstring mit den Pixeldaten direkt. Es findet keine Zuweisung statt. Die 14 Parameter haben folgende Bedeutung:

- Dest** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Pixelgrafikdaten. Der hier angegebene String ist gleichzeitig der Zielstring. Die DRAW_LINE Funktion erwartet, daß dieser String bereits genügend Bytes enthält, um die Grafik mit der angegebenen Formatgröße (FB x FH / 8 Bytes) vollständig aufzunehmen. Üblicherweise wird dieser Zielstring vor der ersten Bearbeitung mit einer Pixelgrafik gefüllt bzw. mit der FILL-Funktion, z.B: SCREEN\$ = FILL\$ ("<00>", 4K) einheitlich mit „weißen“ bzw. „schwarzen“ Hintergrund-Pixeln gefüllt.
- FB** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formatbreite in Pixeln. FB ist stets ein Vielfaches von 8: 0....nnnn. FB gibt die Breite des Grafikbereiches an, in dem man arbeiten möchte. Dies kann die Breite des Grafikbildschirms oder Druckers sein, auf dem die Grafik ausgegeben werden soll. FB kann jedoch auch größer oder kleiner gewählt werden entsprechend der gewünschten Größe der Grafikebene in der man arbeiten möchte.
- FH** ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formathöhe in Pixeln: 0...nnnn, in 1 Pixel-Schritten.
- Durch die Angabe von Formatbreite und -Höhe legt man den aktiven Grafikbereich fest. Nur innerhalb dieses Bereiches kann man Linien ziehen und Pixel sichtbar machen.
- X0, Y0** sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und ist der Basispunkt in kartesischen Koordinaten: -nnnn...0...nnnn.
- Die Koordinaten aller nachfolgenden Linienpunkte (2 Punkte in dieser Funktion, je 1 weiterer mit DRAW_NEXT_LINE) beziehen sich auf diesen Basispunkt, der beliebig in allen 4

Quadranten positioniert werden darf. Diese Angabe erfolgt stets in kartesischen Koordinaten, auch wenn das polare Koordinatensystem für die nachfolgenden Linienpunkte gewählt wird (siehe: MODE → POLAR).

Alle Punkte der Vektorgrafik werden automatisch auf den aktiven Grafikbereich begrenzt, kein Linienpunkt kann außerhalb dieses Bereichs liegen. Wird also z.B. der Basispunkt außerhalb des aktiven Grafikbereichs platziert, so benötigen die Linienpunkte entsprechende relative Koordinaten, so daß sie, zum Wert dieses Basispunktes addiert, tatsächlich in den aktiven Grafikbereich fallen.

4

X1,Y1

sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und ist der 1. Punkt der Linie relativ zum „Basispunkt“ (X0,Y0) in
a) kartesischen Koordinaten im Wertebereich: -8191...0...8191, kleinste Einheit: 1 Pixel.
oder
b) Polar-Koordinaten: Winkel (PHI), Radius (R). Wertebereich Winkel: -36.000...0...36.000, kleinste Einheit: 1=0,01-Grad,
Wertebereich Radius: 0...8191, kleinste Einheit = 1 Pixel

X2,Y2

sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und ist der 2. Punkt der Linie relativ zum „Basispunkt“ (X0,Y0) in
a) kartesischen Koordinaten im Wertebereich: -8191...0...8191, kleinste Einheit: 1 Pixel.
oder
b) Polarkoordinaten: Winkel (PHI), Radius (R). Wertebereich Winkel: -36.000...0...36.000, kleinste Einheit: 1=0,01-Grad,
Wertebereich Radius: 0...8191, kleinste Einheit = 1 Pixel

XScale

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Skalierungsfaktor in X-Richtung in zehntel Prozent an. Wertebereich: 10 ... 30.000 (1%...3000%). Bei Polarkoordinaten bezieht sich die Skalierung auf den Winkel.

YScale

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Skalierungsfaktor in Y-Richtung in zehntel Prozent an. Wertebereich: 10 ... 30.000 (1%...3000%). Bei Polarkoordinaten bezieht sich die Skalierung auf den Radius.

Rot

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Rotationswinkel in 0,01-Grad Schritten gegen den Uhrzeigersinn.

Wertebereich: -36.000...0...36.000, 1=0,01 Grad,
100=1-Grad.

Mode

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt die Linienart an.

Bit-0 und Bit-1 legen die Art der Linie fest:

0: „schwarze“ Linie schreiben: zeichnen auf hellem Grund

1: „weiße“ Linie schreiben: zeichnen auf dunklem Grund

2: „unsichtbare“ Linie schreiben - nur Endpunkte sichtbar.

Wird in der Regel nur für Tests verwendet, z.B. bei der Optimierung von Grafiken für möglichst geringen Speicherbedarf oder bei der Untersuchung von Kurven um möglichst „glatt“ wirkende Ergebnisse zu erhalten.

3: „Inversions“-Linie schreiben. Diese Auswahl wird verwendet um anschließend die umschlossenen Flächen zu füllen (siehe: FILL_AREA(...)). Tatsächlich werden in den Grafikbereich bei dieser Einstellungen nur dort „1“-Pixel geschrieben, wo es bei einem waagerechten Scan später zu einer Inversion der aktuellen Farbe kommen soll (schwarz → weiß bzw. weiß → schwarz). Das Schreiben von Inversions-Linien ist komplett zu trennen vom Schreiben mit den anderen 3 Einstellungen. Der Ablauf beim Zeichnen von Inversions-Linien folgt stets diesem Schema:

1. Gesamten Grafikbereich mit 00-Bytes (=weißen Pixeln) füllen.
2. Alle Inversionslinien wie gewünscht schreiben.
3. Mit Funktion „FILL_AREA (...)“ die umschlossenen Gebiete mit Pixeln füllen
4. Ggf. nun weitere Operationen ausführen wie: schwarze/weiße Linien zeichnen, andere Grafikelemente in das Bild einkopieren, spiegeln, invertieren, maskieren etc.

Beim Schreiben mit „Inversions“-Linien gibt es keine Beschränkungen. Insbesondere dürfen Gebiete, die anschließend gefüllt werden sollen sowohl im Uhrzeigersinn als auch entgegen dem Uhrzeigersinn umschlossen werden. Ebenso dürfen Inversions-Linien auch mehrmals über den gleichen Punkt laufen oder mehrmals über den gleichen Linienzug. Und schließlich können sich mehrere geschlossene Inversions-Kurvenzüge nach Belieben berühren und überlagern. Es gilt stets die Regel, daß sich die Farbe entlang des Linienzuges ändert. Die Beispiele der folgenden Seiten verdeutlichen diesen Zusammenhang.

Bit-4 legt die Reihenfolge der Transformationen fest:

0: erst skalieren, danach rotieren

1: erst rotieren, danach skalieren. Im Falle von Polarkoordinaten führen beide Einstellungen zum gleichen Ergebnis.

Bit-5 legt das Koordinatensystem für die Punkte der Linie fest:

0: kartesische Koordinaten

1: Polarkoordinaten

Beachte: Basispunkt (X0,Y0) immer kartesisch!

Pen

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Es können bis zu 16 Stifte mit unterschiedlichen Einstellungen gleichzeitig verwendet werden. Alle zuvor in dieser Funktion getroffenen Einstellungen können bei jedem der max. 16 Stifte übereinstimmen oder sich voneinander unterscheiden.

Funktionswert

gibt es bei dieser Funktion nicht.

4

Es können mehrere Stifte in die gleiche Grafik schreiben, etwa um mehrere laufend hereinkommende Meßwerte als kontinuierliche Kurvenzüge darzustellen (Oszilloskop, Mehrkanalschreiber, etc.).

Beachte: DEST, der Zielstring für Pixelgrafik, muß vor Anwendung von DRAW_LINE mit genügend vielen Pixeln gefüllt sein damit man mit DRAW_LINE und andere Grafikfunktionen Pixel schreiben und verändern kann. Eine darüber hinausgehende Größe ist auch zulässig und hat keinen Einfluß auf die Größe der aktiven grafischen Fläche

Beachte: Dest ist sollte immer als STATISCHE Variable definiert werden, d.h. es sollte sich um eine GLOBALE Variable oder um eine TASK-LOKALE Variable handeln. Dies ist ein Erfordernis wenn mit verschiedenen Tasks bzw. System-Tasks (z.B. Device-Treiber) gearbeitet wird, und dies bringt in der Regel auch Geschwindigkeitsvorteile zur Laufzeit, da Grafikbereiche nicht stets neu initialisiert werden müssen wie bei Verwendung lokaler Variablen.

Mit DRAW_LINE wird ein neuer Linienzug begonnen und gleichzeitig werden alle relevanten Parameter spezifiziert. Zusätzlich können auch die weiter hinten beschriebenen Kurzformen zur Veränderung der Parameter-Einstellungen verwendet werden siehe: SET_ROTATION, SET_SCALE, etc.)

Auf dem Zeichnen von Linien basieren alle Funktionen der Vektorgrafik. Linien werden sowohl benutzt um schlanke Gebilde zu zeichnen als auch flächige Objekte, die sich aus vielen Teilflächen zusammensetzen können. In den nachfolgenden 25 Beispielprogrammen werden die Möglichkeiten der DRAW_LINE Funktion im Einzelnen erläutert. Die Beispielprogramme laufen direkt auf dem Grafik-Toolkit des BASIC Tigers ab. Die eingesetzte Hardware besteht u.a. aus:

- BASIC-Tiger® Modul: AXI-4/4
- grafisches LC-Display T6963C, 240 x 128 Dots

Die Beispielprogramme sind zusammen mit der erzeugten Bildschirmausgabe hier abgedruckt und liegen auch als fertige „*.TIG“ Files im Entwicklungssystem bzw. im Grafik-Toolkit. vor. Bitte beachten Sie die korrekte Einstellung der Pfadangaben in

Auswahl: OPTIONEN, VERZEICHNISSE, die für die richtige Compilation der Beispiele erforderlich ist.

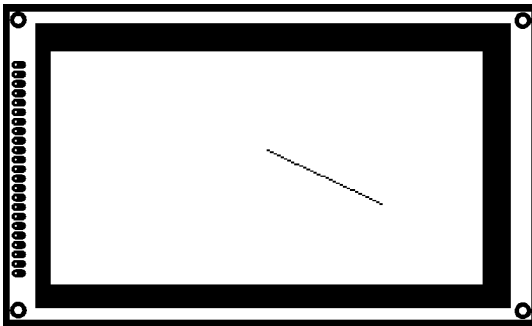
Wenn Sie mit einem T6963C Display mit anderer Bildschirmgröße arbeiten, ändern sich die Formatangaben und die Größe des aktiven grafischen Bereichs entsprechend. In Beispielprogrammen mit bewegten Grafiken ist hier jeweils nur eine Phase der Bildschirmausgabe dargestellt.

Zeichen einer Verbindungslinie zwischen 2 Punkten

Im folgenden Beispiel wird gezeigt welche Instruktionen bzw. Funktionen in diesem einfachsten Fall von Vektorgrafik erforderlich sind:

- Variable zur Aufnahme der Grafik deklarieren: SCREEN\$
- Device-Treiber für grafisches LC Display installieren: LCD-6963.TDD
- Grafikbereich mit Hintergrundfarbe füllen: SCREEN\$ = FILL\$ (....)
- Linien zeichnen und alle Draw-Parameter spezifizieren: DRAW_LINE
- Ergebnisgrafik auf LCD anzeigen: PUT #12, #1, SCREEN\$, ...
- In diesem Fall wurde der Basispunkt in die Mitte des Bildschirms plaziert und hierauf werden die Koordinaten der beiden Punkte bezogen.

4



Programmbeispiel:

```

'-----
' Name: DRAW_1LINE.TIG          -      Vector graphic:  How to draw line
' draws one line (connection between 2 points)
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR STRICT
#include DEFINE_A.INC           ' general definitions
#include UFUNC3.INC             ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC             ' definitions for Graphic Toolkit

STRING SCREEN$(4K)              ' Global Strings for Pixel-data

TASK MAIN                       '
DIR PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0  ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0     ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255    ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)     ' initiate screen string: white dots
'-----
'          <-Format-> Base Pt-1 Pt-2  scaling Rot Draw  Pen
'          Destin,wide high X  Y X  Y X Y   X    Y ang Mode No
DRAW_LINE (SCREEN$,240,128,120,80,0,-27,66,4,1000,1000, 0,  0,  0)
'-----
PUT #12, #1, SCREEN$, 0,0,3840' <-- print graphic on LCD
END

```

Der Unterschied einer auf diese Weise gezogenen Linie besteht unter anderem darin, daß für die Instruktion `DRAW_LINE` wie für andere Vektorgrafik-Instruktionen Parameter eingestellt werden können:

- Skalierung
- Rotation
- Modus

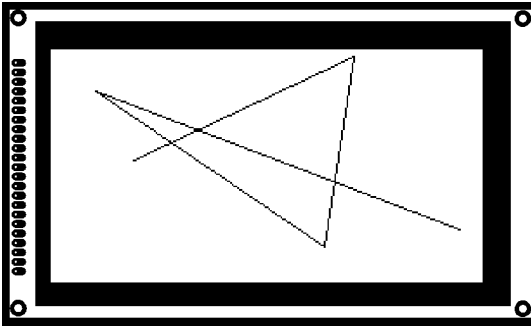
Ein einmal geschriebenes Unterprogramm kann mit verschiedenen Parametern immer wieder verwendet werden, um das gleiche Objekt in verschiedenen Größen und eventuell gedreht erneut zu Zeichnen. Statt der festen Zahlen als Parameter müssen dazu lediglich Variable verwendet werden.

Ein weiterer Vorteil der Vektorgrafik ist, daß der Programmcode in diesem Falle kompakter ist als mehrere hinterlegte Bitmap-Grafiken, die bei diesem LCD jedesmal ca. 4kByte belegen.

Zeichen einer Linie mit mehreren Punkten

Zum Ablauf des vorigen Beispiels kommt nun noch die „DRAW_NEXT_LINE“ Funktion hinzu, mit der ein mit „DRAW_LINE“ begonnener Linienzug beliebig fortgesetzt werden kann. Es wird jeweils der nächste Punkt im DRAW_NEXT_LINE spezifiziert (X-/Y-Koordinaten), der zu der angegebenen PEN-Nr. gehört (3. Parameter, hier = 0).

Alle Draw-Parameter werden übernommen wie sie in DRAW_LINE spezifiziert worden sind.



4

Programmbeispiel:

```

'-----
' Name: DRAW_NEXT_LINE1.TIG - Vector graphic
' DRAW_NEXT_LINE demo
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data
LONG RX1, RY1, N

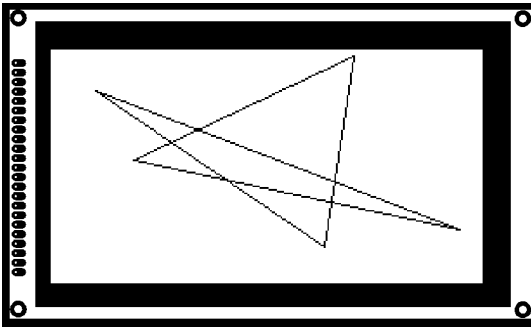
TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
LOOP 9999999000 ' many loops
SCREEN$ = FILL$("<00>",4K) ' init screen to "white" dots
RX1 =(MOD(RND(N*N),240))-120 ' rel X-pos
RY1 =(MOD(RND(N*N),120))-60 ' rel X-pos
DRAW_LINE (SCREEN$,240,128, 120,64, 0,0, RX1,RY1, 1000,1000, 0,0,0)
FOR N=0 TO 70
RX1 =(MOD(RND(N*N),240))-120 ' rel X-pos
RY1 =(MOD(RND(N*N),120))-60 ' rel X-pos
DRAW_NEXT_LINE ( RX1,RY1, 0) '
PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
NEXT
ENDLOOP
END

```


Zeichen einer geschlossenen Linie mit mehreren Punkten

Hier kommt zusätzlich die Funktion „CLOSE_LINE“ zum Einsatz, die für die angegebene PEN-Nr. den letzten Linienzug mit dem 1. Punkt der Linie verbindet.

CLOSE_LINE ist die übliche Methode um einen Kurvenzug zu schließen. Im Falle der „Inversions“-Linien muß jeder Kurvenzug durch „CLOSE_LINE“ geschlossen werden. Das einfache Zeichen zurück bis zum Startpunkt einer Linie reicht hierbei nicht aus.



Programmbeispiel:

4

```

'-----
' Name: CLOSE_LINE.TIG - Vector graphic
' shows how to close a polygon line
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)   ' initiate screen string: white dots
'-----
'          <-Format-> Base Pt-1 Pt-2 scaling Rot Draw Pen
'          Destin,wide high X Y X Y X Y X Y ang Mode No
DRAW_LINE(SCREEN$, 240,128, 120,60,-77,0,50,-60,1000,1000, 0, 0, 0)

DRAW_NEXT_LINE ( 33, 50,0)   ' X= 33, Y= 50 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (-99,-40,0)   ' X=-99, Y=-40 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (111, 40,0)   ' X=111, Y= 40 rel to BASE pt.(120,60)
CLOSE_LINE (0)               ' connect end- and start-point

PUT #12, #1, SCREEN$, 0,0,3840' print graphic on LCD
END

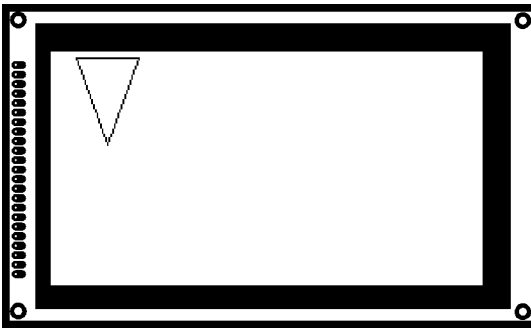
```

Verschieben eines Linienzuges

In diesem Beispiel wird ein Dreieck gezeichnet und ein Stück weit über den Bildschirm bewegt. Die Verschiebung der Grafik wird durch Verändern des Basispunktes vorgenommen. Um den Eindruck einer bewegten Grafik zu erhalten sind diese Voraussetzungen zu erfüllen:

1. Das Bild wird vor jeder neuen Phase mit verschobenen Linien erneut aufgebaut (jedesmal ein FILL\$ zum Löschen der alten Grafik!).
2. Zwischen den Phasen vergeht eine kurze Zeitspanne, damit eine Bewegung in der jeweiligen Geschwindigkeit wahrgenommen werden kann. (WAIT_DURATION 100).

Die zweite Ausführung dieses Beispiels ist ohne 1.) und 2.) ausgeführt und ergibt ein entsprechend anderes Ergebnis.



Programmbeispiel:

4

```

'-----
' Name: MOVE_LINE1.TIG          -          Vector graphic
' shows moving a polygone
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC           ' general definitions
#include UFUNC3.INC             ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC             ' definitions for Graphic Toolkit

STRING SCREEN$(4K),SCREEN2$(4K) ' Global Strings for Pixel-data

TASK MAIN
LONG X0                         ' X/Y coordinates for base point
DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0  ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0     ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255    ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

LOOP 9999999999                ' many loops
FOR X0=30 TO 180 STEP 3         ' move
  SCREEN$ = FILL$ ("<00>",4K)  ' init screen-string with "white" dots
'-----
'          <-Format-> Base Point-1 Pt-2 <scaling> Rot Draw Pen
'          Destin,wide high X Y  X Y  X Y  X Y  ang Mode No
DRAW_LINE(SCREEN$,240,128,X0,77, -20,0, 16,0,1000,1000, 0, 3, 0)
'          -----> X0,Y0 move base point
'
DRAW_NEXT_LINE ( -2, 50,0)      ' X=-2, Y=50 relative to BASE-Point
CLOSE_LINE (0)                  ' connect end- and start-point
FILL_AREA (SCREEN$)             ' fill graphic area with "1" Bits
SCREEN2$=SCREEN$
PUT #12,#1,SCREEN2$,0,0,3840    ' <-- print graphic on LCD
NEXT
ENDLOOP
END

```

Programmbeispiel:

```

-----
' Name: MOVE_LINE2.TIG      -   Vector graphic
' shows extrusion moving a polygone without erasing the previous one
-----
' Note: connect L71 to reset LCD
-----

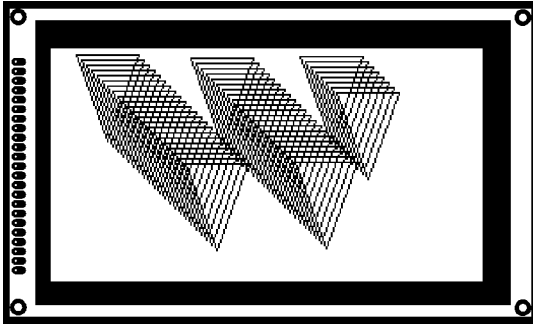
USER_VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
LONG X0,Y0                ' X/Y coordinates for base point
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD RESET,M_LCD RESET,0 ' RESET LCD
OUT P_LCD RESET,M_LCD RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,100 KB/s
SCREEN$ = FILL$ ("<00>",4K)    ' init screen string with white dots

FOR X0=30 TO 180 STEP 3      ' move
  Y0=(X0-30) BITAND 63
  -----
  '          <-Format-> Base Point-1 Pt-2 <scaling> Rot Draw Pen
  '          Destin,wide high X Y   X Y   X Y   X   Y ang Mode No
  DRAW_LINE(SCREEN$,240,128, X0,Y0, -20,0, 16,0,1000,1000, 0, 0, 0)
  '          -----> X0,Y0 move base point
  '
  DRAW_NEXT_LINE ( -2, 50,0) ' X=-2, Y= 50 rel to BASE-Coordinates
  CLOSE_LINE (0)             ' connect end- and start-point
  PUT #12,#1,SCREEN$,0,0,3840 ' print graphic on LCD
NEXT
END

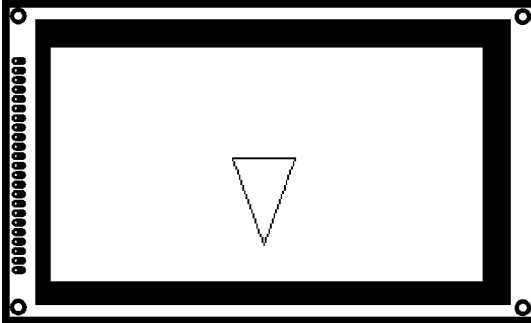
```



4

Rotieren eines Linienzuges

Hier wird ein Dreieck in mehreren aufeinander folgenden Phasen um den Basispunkt rotiert. Der Rotationswinkel wird jeweils um 3-Grad (300) von Bild zu Bild inkrementiert. Hierzu wird im FOR-Loop der Parameter „ROT“ der DRAW_LINE Funktion schrittweise inkrementiert.

**4**

Programmbeispiel:

4

```

'-----
' Name: DRAW_ROTATE.TIG - Vector graphic
' shows how to rotate a polygon line
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K),SCREEN2$(4K) ' Global Strings for Pixel-data

TASK MAIN
LONG ANGLE                  ' rotation angle
DIR_PORT 8,1000000b         ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b       ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

FOR ANGLE=0 TO 36000*2 STEP 300 ' rotate in Steps of +1.00 degrees
  SCREEN$ = FILL$("<00>",4K) ' init screen-string with "white" dots
  '-----
  '          <-Format> Base Point-1 Pt-2 <scaling> Rot Draw Pen
  '          Destin,wide high X Y X Y X Y X Y ang Mode No
  DRAW_LINE(SCREEN$,240,128,120,60,-20,0, 16,0,1000,1000, ANGLE, 0, 0)

  DRAW_NEXT_LINE ( -2, 50,0) ' X=-2, Y= 50 relative to BASE coord.
  CLOSE_LINE (0)             ' connect end- and start-point
  SCREEN2$=SCREEN$
  PUT #12,#1, SCREEN2$,0,0,3840 ' <-- print graphic on LCD
  WAIT DURATION 100
NEXT
END

```


Skalieren eines Polygons

Das Dreieck aus den vorigen Beispielen wird hier in seiner Größe verändert durch die Skalierungsfaktoren XScale bzw. YScale in der DRAW_LINE Funktion.

Im ersten FOR-Loop wird nur die Breite verändert (XScale), im 2. Loop Höhe und Breite gleichzeitig.

Programmbeispiel:

```

-----
' Name: DRAW_SCALE1.TIG - Vector graphic
' shows how to re-scale size of a polygon
-----
' Note: connect L71 to reset LCD
-----

USER_VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K),SCREEN2$(4K) ' Global Strings for Pixel-data

TASK MAIN
LONG SCX,SCY              ' X-scaling/Y-scaling factor:1000=100%
DIR_PORT 8,1000000b       ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

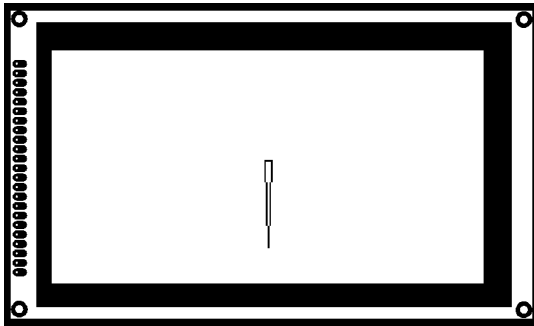
FOR SCX=3000 TO 100 STEP -25 ' re-scale x-coordinate
  SCREEN$ = FILL$("<00>",4K) ' init screen-string with "white" dots
  -----
  '          <Format>   Base Point-1 Pt-2 <scaling> Rot Draw Pen
  '          Destin,wide high X Y X Y X Y X Y ang Mode No
  DRAW_LINE(SCREEN$,240,128, 120,60,-20,0, 16,0, SCX,1000, 0, 0, 0)
  '                                     X-Scaling factor ----> SCX
  -----
  DRAW_NEXT_LINE ( -2, 50,0) ' X=-2, Y=50 relative to BASE-Point
  CLOSE_LINE (0)             ' connect end- and start-point
  SCREEN2$=SCREEN$
  PUT #12,#1,SCREEN2$,0,0,3840 ' print graphic on LCD
  WAIT DURATION 100
NEXT

FOR SCX=1500 TO 100 STEP -25 ' re-scale x- AND y-coordinates
  SCREEN$ = FILL$("<00>",4K) ' init screen-string with "white" dots
  -----
  '          <Format>   Base Point-1 Pt-2<scaling> Rot Draw Pen
  '          Destin,wide high X Y X Y X Y X Y ang Mode No
  DRAW_LINE(SCREEN$,240,128, 120,20,-20,0, 16,0,SCX,SCX, 0, 0, 0)
  '                                     Both factors X-Scaling, Y-Scaling ----> SCX, SCX
  -----

```

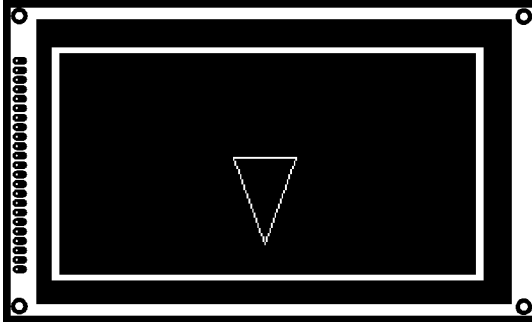
```
DRAW_NEXT_LINE ( -2, 50,0)   ' X=-2, Y= 50 relative to BASE-Point
CLOSE_LINE (0)               ' connect end- and start-point
SCREEN2$=SCREEN$
PUT #12,#1,SCREEN2$,0,0,3840 ' print graphic to LCD
WAIT DURATION 100
NEXT
END
```

4



„Weiße“ Linien zeichnen

Der Grafikbereich in diesem Beispiel wird zunächst mit „schwarzen“ Pixeln gefüllt und dann mit weißen Linien beschrieben (Mode-Parameter = 1 in DRAW_LINE).



4

Programmbeispiel:

4

```

'-----
' Name: DRAW_WHITE.TIG          -          Vector graphic
' show how write WHITE lines
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC           ' general definitions
#include UFUNC3.INC             ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC             ' definitions for Graphic Toolkit

STRING SCREEN$(4K),SCREEN2$(4K) ' Global Strings for Pixel-data

TASK MAIN
LONG ANGLE
DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

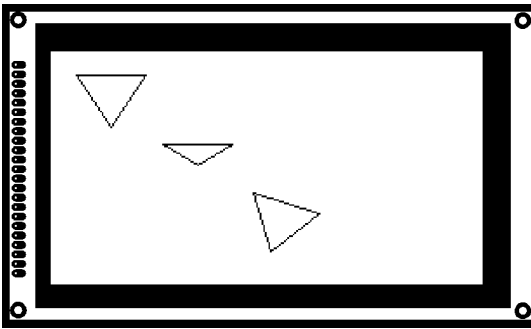
FOR ANGLE=0 TO 36000*2 STEP 300 ' rotate in Steps of +1.00 degrees
  SCREEN$ = FILL$("<OFFH>",4K) ' init screen-string with "BLACK" dots
'-----
'          <Format> Base Point-1 Pt-2<scaling> Rot Draw Pen
'          Destin,wide high X Y X Y X Y X Y ang Mode No
DRAW_LINE(SCREEN$,240,128,120,60,-20,0,16,0,1000,1000,ANGLE, 1, 0)
'          -----> White Line with Draw-Mode =1,
'
DRAW_NEXT_LINE ( -2, 50,0)      ' X=-2, Y= 50 relative to BASE point
CLOSE_LINE (0)                  ' connect end- and start-point
SCREEN2$=SCREEN$
PUT #12, #1, SCREEN2$,0,0,3840' print graphic on LCD
WAIT DURATION 100
NEXT
END

```

Zeichnen mit mehreren Stiften

Dieses Beispiel zeigt den Einsatz mehrerer Stifte in der DRAW_LINE Funktion. Jedem Stift ist eine „PEN-Nr“ zugeordnet. Alle DRAW_LINE Parametereinstellungen für eine bestimmte Stift-Nr. gehören jeweils nur zu diesem Stift.

In diesem Beispiel werden 3 Stifte verwendet, die mit unterschiedlichen Werten für die Skalierungen bzw. für die Rotation parametrisiert werden. Obwohl die Koordinaten der 3 Linienzüge übereinstimmen, erscheinen auf dem Display unterschiedliche Dreiecke, entsprechend den unterschiedlich eingestellten Parameter.



4

Programmbeispiel:

4

```

'-----
' Name: DRAW_PENS.TIG      -      Vector-graphic
' shows how to write with SEVERAL PENS
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

SCREEN$ = FILL$ ("<0>",4K)   ' init screen-string: white dots
'-----
'          <Format>   Base   Pt1 Pt2 <scaling>   Rot Draw Pen
'          Destin,wide high X Y   X Y X Y       X   Y   ang Mode No
DRAW_LINE (SCREEN$,240,128, 50,10,-40,0,0,0, 1000,1000, 0, 0, 0)
DRAW_LINE (SCREEN$,240,128,100,50,-40,0,0,0, 1000, 400, 0, 0, 1)
DRAW_LINE (SCREEN$,240,128,150,90,-40,0,0,0, 1000,1000, 1800, 0, 2)

DRAW_NEXT_LINE (-20,30,0)    ' next line of PEN-0
DRAW_NEXT_LINE (-20,30,1)    ' next line of PEN-1
DRAW_NEXT_LINE (-20,30,2)    ' next line of PEN-2

CLOSE_LINE (2)               ' close line of PEN-2
CLOSE_LINE (1)               ' close line of PEN-1
CLOSE_LINE (0)               ' close line of PEN-0

PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

```

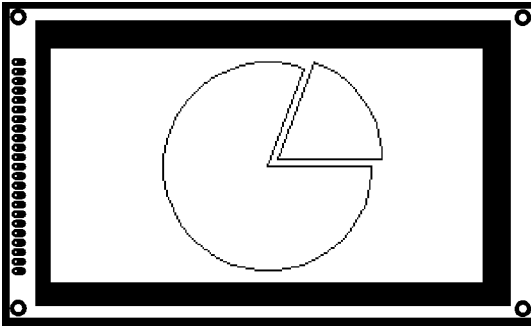
Zeichnen mit Polarkoordinaten

Charts und Zeichnungen, die Rotationssymmetrie aufweisen, lassen sich besonders einfach in Polarkoordinaten beschreiben. Das Beispiel hier zeigt die Darstellung von Pie-Charts mit der DRAW_LINE Funktion und Polarkoordinaten.

Dazu wird das Mode-Bit-5 = 1 gesetzt, die Subroutine „PIE“ übernimmt die Parameter:

- Radius
- X-/Y-Position (kartesisch)
- Anfangs- und Endwinkel
- Winkelschrittweite
- sowie Draw-Mode

4



Programmbeispiel:

4

```

'-----
' Name: DRAW_POLAR.TIG - Vector graphic
' show how to draw in polar coordinates
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data
LONG ANG

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H '240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)    ' init graphic-string to "white"

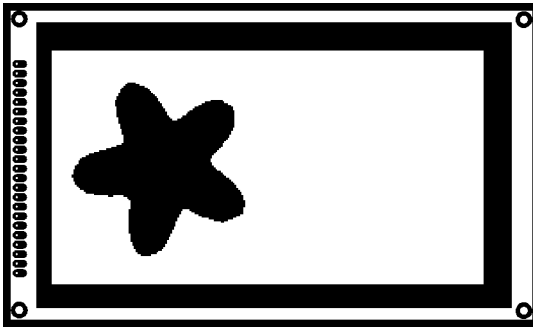
CALL PIE (60,120,64,      0,29000,500,0)'RADIUS, X,Y,ANG0,ANG1,Step,MOD
CALL PIE (60,126,60,29000,36000,500,0)'RADIUS, X,Y,ANG0,ANG1,Step,MOD
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

'-----
' Draw a PIE in a polar system
'-----
SUB PIE (LONG RADIUS, X, Y, ANG0, ANG1, AST, MODE)' print a pie chart
LONG ANG
DRAW_LINE(SCREEN$,240,128,X,Y,0,0,ANG0,RADIUS,1000,1000,0,32+MODE,0)
FOR ANG=ANG0 TO ANG1 STEP AST
DRAW_NEXT_LINE ( ANG, RADIUS, 0)
NEXT
CLOSE_LINE (0)
END

```


Zeichnen und füllen mit Polarkoordinaten

In diesem Beispiel wird eine Form mit Polarkoordinaten gezeichnet und anschließend mit der „FILL_AREA“ Funktion gefüllt. Eine Sinus-Welle ist auf dem Umfang eines Kreises aufgebracht.

**4**

Programmbeispiel:

4

```

'-----
' Name: DRAW_POLAR_FILL.TIG Vector graphic
' draw in polar coordinates + fill area
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

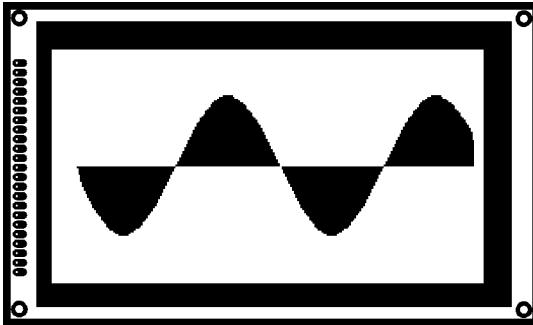
TASK MAIN
  DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b    ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
  SCREEN$ = FILL$ ("<00>",4K)    ' init screen-string with "white" dots
  CALL SIN_POLAR(20,1000,1000,60,63,1200,3)'RAD,SCAL,SC-Y,X,Y, ANGL,MOD
  FILL AREA (SCREEN$)           ' fill screen-area now with "1" Bits
  PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic to LCD
END

'-----
' Draw a sinus waves in a polar system
'-----
SUB SIN_POLAR (LONG RADIUS, SC, SCY, X, Y, ANG, ART) ' show a Sinus
LONG ANGLE, ASTEP, RADIUS2
  ASTEP = 36000/60
  RADIUS2 = RADIUS*2
  DRAW_LINE (SCREEN$,240,128, X,Y, 0,RADIUS2,
             ASTEP,RADIUS2+QUICK_WORD_SIN(ASTEP*5,RADIUS2/3), &
             SC, SCY, ANG,ART+32,0)
  FOR ANGLE =2*ASTEP TO 36000-ASTEP STEP ASTEP
    DRAW_NEXT_LINE (ANGLE, RADIUS2+QUICK_WORD_SIN (ANGLE*5,RADIUS2/3), 0)
  NEXT
  CLOSE_LINE (0)
END

```

Zeichnen und füllen mit kartesischen Koordinaten

Dieses Beispiel zeigt eine Sinuswelle in kartesischen Koordinaten. Anschließend wird der Kurvenzug mit der „FILL_AREA“ Funktion gefüllt. Damit „FILL_AREA“ funktioniert, wird im DRAW_LINE der Modus „Inversions-Line“ (Mode=3) gewählt.



4

Programmbeispiel:

4

```

'-----
' Name: DRAW_CARTESIAN.TIG Vector graphic
' draw in cartes. coord. + fill area
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

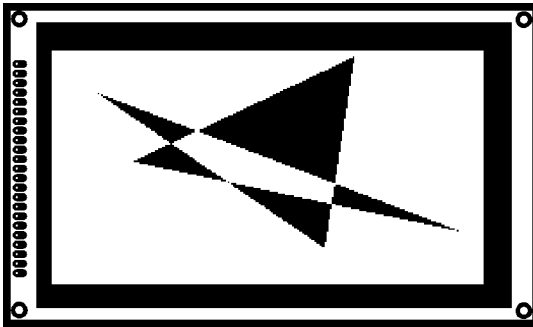
TASK MAIN
DIR PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)    ' init screen-string with "white" dots
CALL SIN_CURVE(20,3000,2000,10,63,0,3)'RADIUS,SCALE,SC-Y,X,Y,ANGL,ART
FILL AREA (SCREEN$)          ' fill screen-area now with "1" Bits
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

'-----
' draw a sinus curve cartesian
'-----
SUB SIN_CURVE (LONG RADIUS,SC,SCY,X,Y,ANG,MODE) '
LONG ANGLE, ASTEP, RADIUS2, XX
  XX=1
  ASTEP = 36000/120
  RADIUS2 = RADIUS*2
  DRAW LINE (SCREEN$,240,128, X,Y, 0,0, 0,0, SC, SCY, ANG,MODE,0)
  FOR ANGLE =2*ASTEP TO 36000-ASTEP STEP ASTEP
    DRAW_NEXT_LINE ( XX, QUICK_WORD_SIN (ANGLE*3,RADIUS2/2), 0)
    XX=XX+1
  NEXT
  DRAW_NEXT_LINE ( XX, 0, 0)
  CLOSE_LINE (0)
END

```

Polygon zeichnen und füllen

Hier wird ein Polygon aus beliebig angeordneten Punkten gebildet und mit „FILL_AREA“ gefüllt. Inversions-Linien dürfen sich beliebig überschneiden, benachbarte Flächen besitzen jeweils inverse Farben.



Programmbeispiel:

4

```

'-----
' Name: DRAW_FILL1.TIG - Vector graphic
' draws a polygon and fills it
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150 KB/s
SCREEN$ = FILL$ ("<00>",4K)   ' init screen-string with "white" dots

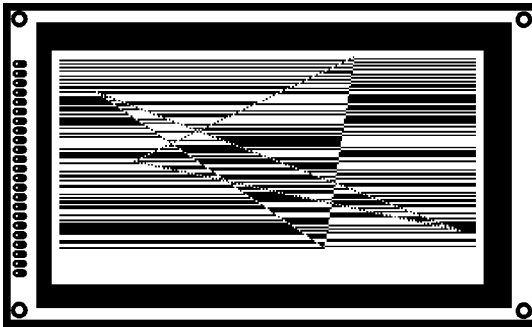
'-----
'          <Format>   Base  Point-1 Pt-2<scaling> Rot Draw Pen
'          Destin,wide high  X Y  X Y  X Y  X  Y ang Mode No
DRAW_LINE(SCREEN$,240,128, 120,60,-77,0,50,-60,1000,1000, 0, 3, 0)

DRAW_NEXT_LINE ( 33, 50,0)    ' X= 33, Y= 50 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (-99,-40,0)    ' X=-99, Y=-40 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (111, 40,0)    ' X=111, Y= 40 rel to BASE pt.(120,60)
CLOSE_LINE (0)                ' connect end- and start-point
FILL_AREA (SCREEN$)           ' fill screen now with "1" Bits
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic to LCD
END

```

Polygon zeichnen und füllen - jedoch fehlerhaft

Hier wird eine typische Darstellung gezeigt, die dabei entsteht, wenn man Gebiete füllen möchte, jedoch fälschlicherweise mit „schwarzen“ Stiften gezeichnet (Mode = 0) hat an Stelle „Inversions-Linien“ zu verwenden (Mode = 3).



4

Programmbeispiel:

4

```

'-----
' Name: DRAW_FILL2.TIG - Vector graphic
'-----
' shows what happens when we you try to FILL a graphic area using an
' improper mode (=0). See correct use of fill area (...) in previous
' examples (e.g.DRAWFILL.TIG). Mode-Bits 0 and 1 must both be set: '1'
'-----
' Note: connect L71 to reset LCD
'-----
USER VAR STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

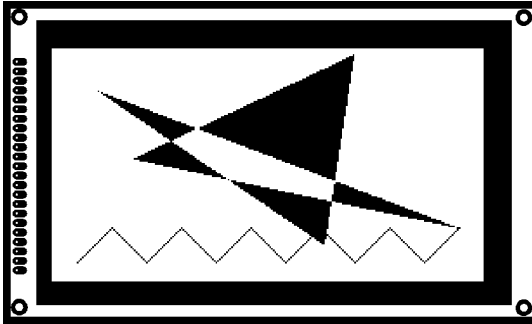
TASK MAIN
DIR PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string: white dots

'-----
' <Format> Base Point-1 Pt-2<scaling> Rot Draw Pen
' Destin,wide high X Y X Y X Y X Y ang Mode No
DRAW_LINE(SCREEN$,240,128, 120,60,-77,0,50,-60,1000,1000, 0, 0, 0)
DRAW_NEXT_LINE ( 33, 50,0) ' X= 33, Y= 50 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (-99,-40,0) ' X=-99, Y=-40 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (111, 40,0) ' X=111, Y= 40 rel to BASE pt.(120,60)
CLOSE_LINE (0) ' connect end- and start-point
'-----
FILL AREA (SCREEN$) ' fill screen-area now with "1" Bits
PUT #12, #1, SCREEN$, 0,0,3840' print graphic to LCD
END

```


Polygon zeichnen und füllen, sowie schwarze Linien zeichnen

Hier wird das gleiche Polygon gezeichnet und gefüllt, zusätzlich werden noch schwarze Linien hinzugefügt. Dies geschieht jedoch erst nach der „FILL_AREA“ Funktion.

**4**

Programmbeispiel:

4

```

'-----
' Name: DRAW_FILL3.TIG - Vector graphic
' polygon filling and lines in the same picture
'-----
' first draw inversion lines + fill_area
' then draw additional line (black or white)
'-----
' Note: connect L71 to reset LCD
'-----
USER VAR STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string with "white" dots

'-----
' <Format> Base Point-1 Pt-2 <scaling> Rot Draw Pen
' Destin,wide high X Y X Y X Y X Y ang Mode No
DRAW_LINE(SCREEN$,240,128, 120,60,-77,0,50,-60,1000,1000, 0, 3, 0)
DRAW_NEXT_LINE ( 33, 50,0) ' X= 33, Y= 50 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (-99,-40,0) ' X=-99, Y=-40 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (111, 40,0) ' X=111, Y= 40 rel to BASE pt.(120,60)
CLOSE_LINE (0) ' connect end- and start-point
'-----
FILL_AREA (SCREEN$) ' fill screen-area now with "1" Bits

'----- now draw black lines (no area filling any more) -----
DRAW_LINE (SCREEN$,240,128, 0,0, 10,120, 30,100, 1000,1000, 0, 0,0)
DRAW_NEXT_LINE ( 50, 120,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 70, 100,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 90, 120,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 110, 100,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 130, 120,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 150, 100,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 170, 120,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 190, 100,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 210, 120,0) ' draw relative to BASE-Point
DRAW_NEXT_LINE ( 230, 100,0) ' draw relative to BASE-Point
PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic on LCD
END

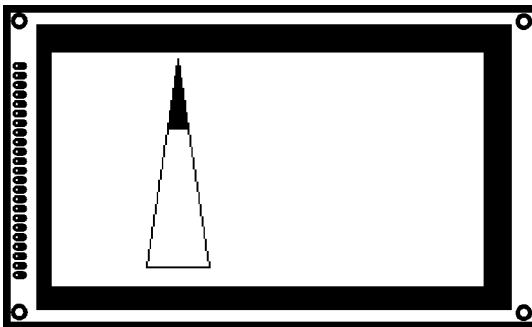
```

Timing- / Phasenproblem bei bewegten Grafiken

Dieses Beispiel zeigt ein Darstellungsproblem auf, das bei bewegten Grafiken entstehen kann, wenn man die zeitlichen Zusammenhänge von Entstehung und Ausgabe eines Grafikbildschirmes nicht beachtet. Je nach Geschwindigkeit des eingesetzten Tiger Moduls und seiner Betriebsart (RUN- oder PC-Mode) wird mit diesem Beispielprogramm ein bewegtes, jedoch unvollständig dargestelltes Dreieck ausgegeben.

Der Grund hierfür liegt in dem Umstand, daß der Grafikstring „SCREEN\$“ bereits kurz nachdem er zur Ausgabe der Pixel an den Device-Treiber (PUT #13, #1, ...) übergeben wird, bereits wieder verändert wird. Der Device-Treiber „LCD2.TDD“ übernimmt den Grafikdaten String im „direct-access-mode“. Das bedeutet, der Treiber kopiert die Daten nicht selbst in einen lokalen Puffer, sondern holt die Grafikbytes einzeln zu bestimmten Zeitpunkten aus dem angegebenen Quellstring heraus. Dies erfolgt mit einer Übertragungsrate von ca. 10 KByte/sec ... 150 KByte / Sec je nach Treiber Einstellung, so daß die gesamte Übertragung ca. 30 ... 500 ms dauert. Während dieser Zeit sollte der Ausgabestring nicht verändert werden.

Im vorliegenden Beispiel wird oben im FOR-Loop mit „FILL\$“ sowie dem anschließenden DFRAW_LINE die Grafik in „SCREEN\$“ schon verändert, noch bevor alles Pixel an den LCD übertragen sind, so daß eine verstümmelte Darstellung entsteht.



Programmbeispiel:

4

```

'-----
' Name: TIME_PHASE1.TIG Vector gr.: moving graphics
' demonstrates phase + timing problem with moving graphics
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

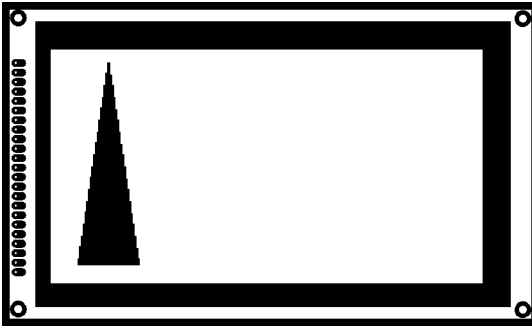
STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
LONG X0                    ' X/Y coordinates for base point
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
X0=30
LOOP 999999999
WAIT_DURATION 15           ' just a moment
X0=MODULO_INC (X0,30,200,1)
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string with "white" dots
'-----
'
'          <-Format-> Base Point-1 Pt-2 <scaling> Rot Draw
Pen
'          Destin,wide high X Y      X Y      X Y      X      Y ang Mode No
DRAW_LINE(SCREEN$,240,128, X0,120, -20,0, 16,0, 1000,1000, 0, 3, 0)
'          -----> X0,Y0 move base point
'
DRAW_NEXT_LINE ( -2,-120,0) ' relative to BASE-Coordinates
CLOSE_LINE (0)              ' connect end- and start-point
FILL_AREA (SCREEN$)         ' fill graphic area now with 1-bits
PUT #12,#1,SCREEN$,0,0,3840 ' print graphic on LCD
ENDLOOP
END

```

Timing- / Phasenproblem behoben

Das Timing- / Phasenproblem des vorigen Beispiels ist hier behoben indem ein Hilfsstring „SCREEN2\$“ verwendet wird. „SCREEN2\$“ übernimmt das fertige Bild und übergibt es an den Device-Treiber, während das darauffolgende Bild schon wieder im Originalstring „SCREEN\$“ aufgebaut wird. Somit findet keine Störung des Ausgabevorganges mehr statt.

**4**

Programmbeispiel:

4

```

'-----
' Name: TIME_PHASE2.TIG      Moving vector-graphics
' shows how the timing + phase problem of TIMPHAS1.TIG is solved
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K), SCREEN2$(4K) ' Global Strings for Pixel-data

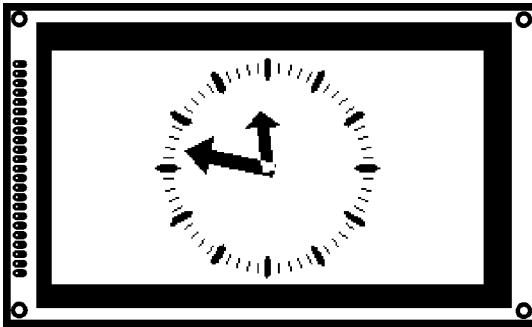
TASK MAIN
LONG X0                    ' X/Y coordinates for base point
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
X0=30
LOOP 999999999
  WAIT_DURATION 15          ' just a moment please
  X0=MODULO_INC (X0,30,200,1)
  SCREEN$ = FILL$ ("<00>",4K) ' init screen-string with "white" dots
'-----
'          <-Format-> Base   Pt-1 Pt-2  <scaling> Rot Draw Pen
'          Destin, wide high X Y   X Y X Y      X   Y ang Mode No
DRAW_LINE(SCREEN$, 240,128,X0,120,-20,0,16,0, 1000,1000, 0, 3, 0)
'          -----> X0,Y0 move base point
'
DRAW_NEXT_LINE ( -2,-120,0) ' relative to BASE-Coordinates
CLOSE_LINE (0)              ' connect end- and start-point
FILL_AREA (SCREEN$)         ' fill graphic area now with "1" Bits
SCREEN2$=SCREEN$
PUT #12,#1,SCREEN2$,0,0,3840 ' print graphic on LCD
ENDLOOP
END

```

Anwendungsbeispiel „Uhr“

Hier wird eine mögliche Anwendung von Vektorgrafik zum Aufbau eines Zifferblattes mit Zeigern gezeigt. Hier dargestellt ist eine Uhr mit Minuten- und Stundenzeiger, man kann sich leicht auch eine andere Teilung vorstellen für Meßinstrumente wie Höhenmesser, Durchflußmesser, etc. Ggf. können auch digitale Werte in die Uhr eingeblendet werden.

Vektorgrafik kann sehr vorteilhaft für solche Applikationen eingesetzt werden, da die Instrumente individuell gestaltet werden können und sich leicht verschieben, rotieren und skalieren lassen. Im Grafik-Toolkit ist eine Kollektion solcher Instrumente im Source-Code enthalten.



Programmbeispiel:

4

```

'-----
' Name: CLOCK1.TIG - Vector graphic
' displays a clock
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general defines
#include UFUNC3.INC ' defines user function codes
#include LCD_4.INC ' defines LCD Typ 4
#include GR_TK1.INC ' defines Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

TASK MAIN
LONG X0,Y0, DOT

DIR PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR PORT 7,1111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string with "white" dots

'-----
' Draw the clocks face / Dial
'-----
X0= 120 ' Base Point
Y0= 64 '
FOR DOT=0 TO 36000-3000 STEP 3000 ' Dot-Angle
DRAW_LINE (SCREEN$,240,128, X0,Y0, 0,-27, 1,-26, 2400,2400,DOT,3,0)
DRAW_NEXT_LINE ( 1,-22, 0) '
DRAW_NEXT_LINE ( 0,-21, 0) '
DRAW_NEXT_LINE (-1,-22, 0) '
DRAW_NEXT_LINE (-1,-26, 0) '
CLOSE_LINE (0) '
NEXT
CALL HAND (10200, 3800, 3800, X0,Y0,3) ' Angle, Scale-X/Y, X/Y, Mode=3
CALL HAND (17350, 3800, 2600, X0,Y0,3) ' Angle, Scale-X/Y, X/Y, Mode=3
FILL_AREA (SCREEN$) ' fill screen area with 1-Bits
FOR DOT=0 TO 36000-600 STEP 600 ' Dot-Angle
DRAW_LINE (SCREEN$,240,128, X0,Y0, 0,-25, 0,-23, 2400,2400, DOT,0,0)
NEXT
PUT #12, #1,SCREEN$,0,0,3840 ' print graphic on LCD
END

'-----
' Draw a clocks hand
'-----

SUB HAND (LONG ANGLE, XSCALING, YSCALING, X, Y, ART)
DRAW_LINE (SCREEN$, 240,128,X,Y,0,13,3,9,XSCALING,YSCALING,ANGLE,ART,0)
DRAW_NEXT_LINE ( 1, 9, 0)
DRAW_NEXT_LINE ( 1,-1, 0)
DRAW_NEXT_LINE (-1,-1, 0)
DRAW_NEXT_LINE (-1, 9, 0)

```


DRAW_LINE

```
DRAW_NEXT_LINE (-3, 9, 0)  
CLOSE_LINE (0)  
END
```

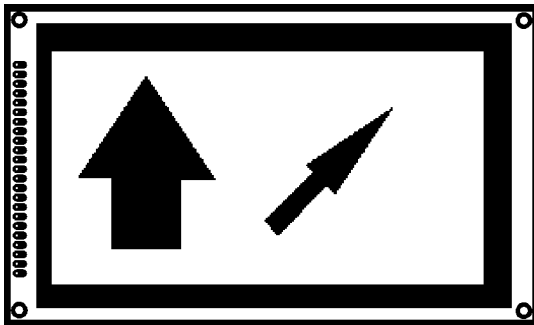
4

Transformationen: erst skalieren, dann rotieren und umgekehrt

Alle mit DRAW-Funktionen gezeichneten Linien und Flächen lassen sich in kleinen Schritten sowohl skalieren (0,1%-Schritte) als auch drehen (0,01-Grad Schritte). Diese Transformationsmöglichkeiten vereinfachen die Entwicklung und Ausgabe von Grafiken und sparen Rechenzeit und Speicherplatz zur Laufzeit. Skalen, Zeiger, Symbole und andere grafischen Elemente brauchen jeweils nur einmal definiert zu werden und können dann jeweils in Größe, Breite, Höhe und Drehung entsprechend der Anwendung eingesetzt werden.

Bei Grafiken, die im kartesischen Koordinatensystem definiert werden, gibt es diese zwei Möglichkeiten:

- erst skalieren, dann rotieren, bzw.
- erst rotieren, dann skalieren.



Entsprechend unterschiedliche Ergebnisse werden erzielt wie die Beispiele hier zeigen.

Programmbeispiel:

```

'-----
' Name: DRAW_CART_SCALE_ROT.TIG - Vector graphic
' first scale, then rotate, cartesian
'-----
' Note: connect L71 to reset LCD
'-----

USER VAR STRICT
#include DEFINE A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD 4.INC            ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K)            ' Global Strings for Pixel-data

TASK MAIN
  DIR PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
  SCREEN$ = FILL$("<00>",4K)    ' init screen string white dots
  CALL DRAW_ARROW( 50,70,10000,10000,0,3) ' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW(150,70, 3000,10000,4500,3)' X0,Y0,SCX,SCY,ROT,MODE
  FILL_AREA (SCREEN$)           ' fill screen-area with "1" Bits
  PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

SUB DRAW_ARROW (LONG X0,Y0,SCX,SCY,ROT,MODE) ' <-- draw an Arrow
  DRAW_LINE (SCREEN$,240,128, X0,Y0, -2,4, -2,0, SCX,SCY, ROT, MODE,0)
  DRAW_NEXT_LINE (-4, 0, 0)
  DRAW_NEXT_LINE ( 0, -6, 0)
  DRAW_NEXT_LINE ( 4, 0, 0)
  DRAW_NEXT_LINE ( 2, 0, 0)
  DRAW_NEXT_LINE ( 2, 4, 0)
  CLOSE_LINE (0)
END

```

Programmbeispiel:

4

```

'-----
' Name: DRAW_CART_ROT_SCALE.TIG - Vector graphic
' first rotate, then scale, cartesian
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

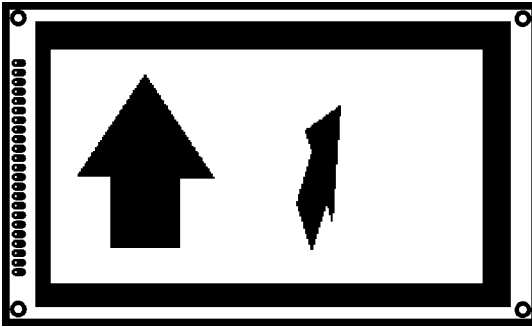
STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
LONG X0,Y0, DOT
DIR PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)    ' init screen-string to white
CALL DRAW_ARROW ( 50,70,10000,10000,0,3) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (150,70, 3000,10000,4500,19)' X0,Y0,SCX,SCY,ROT,MODE
FILL_AREA (SCREEN$)          ' fill screen now with 1-bits
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic to LCD
END

SUB DRAW_ARROW (LONG X0,Y0,SCX,SCY,ROT,MODE)' draw an Arrow
DRAW_LINE (SCREEN$,240,128, X0,Y0, -2,4, -2,0, SCX,SCY, ROT, MODE,0)
DRAW_NEXT_LINE (-4, 0, 0)
DRAW_NEXT_LINE ( 0, -6, 0)
DRAW_NEXT_LINE ( 4, 0, 0)
DRAW_NEXT_LINE ( 2, 0, 0)
DRAW_NEXT_LINE ( 2, 4, 0)
CLOSE_LINE (0)
END

```

DRAW_LINE



4

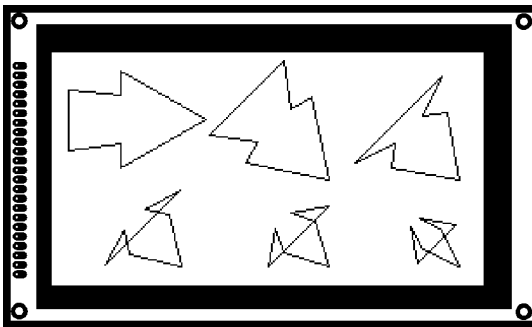
Polarkoordinaten - PHI-Skalierung

Hier ist ein Pfeilsymbol in Polarkoordinaten definiert worden. Es wird in seiner ursprünglichen Form und in transformierten Variationen dargestellt:

1. Originalform
2. 45-Grad rotiert (4500), PHI ist skaliert auf 80% (800)
3. 45-Grad rotiert (4500), PHI ist skaliert auf 60% (600)
4. 45-Grad rotiert (4500), PHI ist skaliert auf 40% (400)
5. 45-Grad rotiert (4500), PHI ist skaliert auf 30% (300)
6. 45-Grad rotiert (4500), PHI ist skaliert auf 18% (180)

Betrachtet man die PHI-Skalierung von Grafiken in Polarkoordinaten im Detail, wird klar, daß die Veränderung des Aussehens auch stark abhängt von der ursprünglichen Definition der Grafik.

Zwei Beispielprogramme mit ähnlichen Pfeilen, jeweils unterschiedlich definiert, zeigt die Auswirkung bei der PHI-Skalierung.



Programmbeispiel:

```

'-----
' Name: DRAW_POLAR_ROT_SCALE1.TIG      -      Vector graphic
' Polar: rotate and scale PHI
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings Pixel-Data

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$("<00>",4K)    ' init screen-string
CALL DRAW_ARROW ( 35,35,1000,7000,0,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (120,35, 800,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (195,35, 600,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW ( 35,85, 400,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (120,85, 300,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (195,85, 180,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
PUT #12, #1, SCREEN$, 0,0,3840          ' print graphic on LCD
END

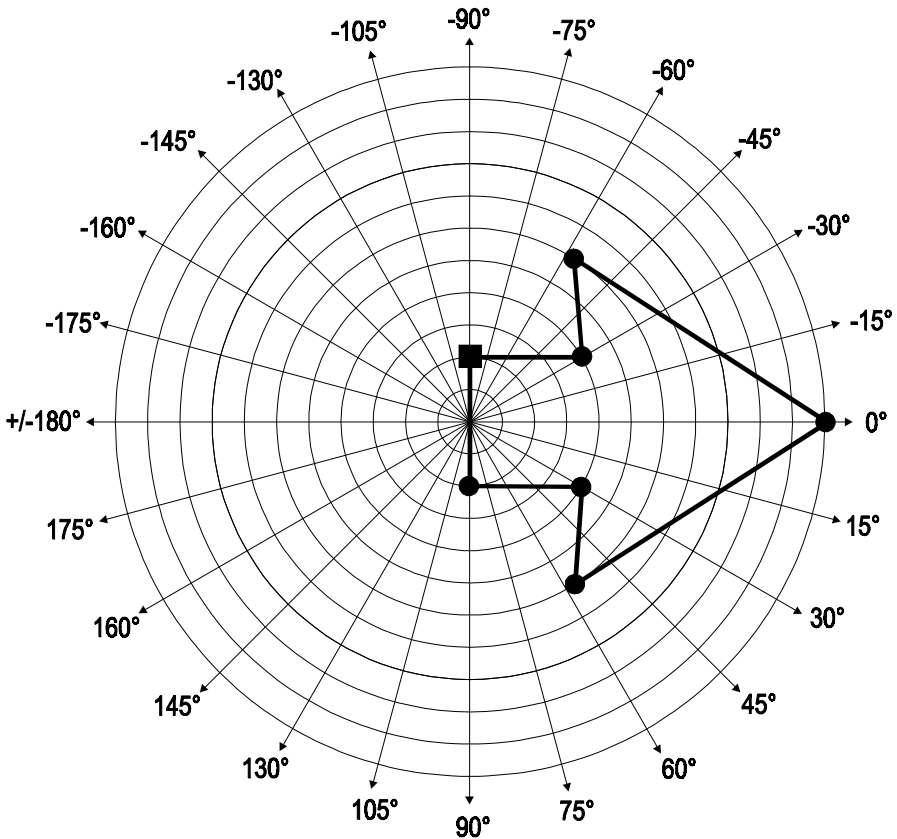
SUB DRAW_ARROW(LONG X0,Y0,SCX,SCY,ROT,MODE) ' <-- draw an Arrow
DRAW_LINE(SCREEN$,240,128,X0,Y0, -15000,5,-9000,2,SCX,SCY,ROT,MODE,0)
DRAW_NEXT_LINE (-9000, 4, 0)
DRAW_NEXT_LINE ( 0, 7, 0)
DRAW_NEXT_LINE ( 9000, 4, 0)
DRAW_NEXT_LINE ( 9000, 2, 0)
DRAW_NEXT_LINE (15000, 5, 0)
CLOSE_LINE (0)
END

```


Polarkoordinaten - PHI-Skalierung (2)

Ein ähnlicher Pfeil wie zuvor wird in Polarkoordinaten definiert und anschließend in verschiedenen Versionen transformiert gezeigt:

1. Originalform
2. 45-Grad rotiert (4500), PHI ist skaliert auf 80% (800)
3. 45-Grad rotiert (4500), PHI ist skaliert auf 60% (600)
4. 45-Grad rotiert (4500), PHI ist skaliert auf 40% (400)
5. 45-Grad rotiert (4500), PHI ist skaliert auf 30% (300)
6. 45-Grad rotiert (4500), PHI ist skaliert auf 18% (180)



Pfeil lt. Definition in Beispiel: DRAW_POLAR_ROT_SCALE2.TIG

Der Vergleich mit vorigem Beispiel zeigt die Unterschiede auf Grund andersartiger Definition der Grafik.

Programmbeispiel:

4

```

'-----
' Name: DRAW_POLAR_ROT_SCALE2.TIG -      Vector graphic
' polar: rotate and scale PHI
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel

TASK MAIN
  DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b    ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 '
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
  SCREEN$ = FILL$("<00>",4K)    ' init screen-string "white"
  CALL DRAW_ARROW ( 15,35,1000,7000,0,32) ' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW (100,15, 800,7000,4500,32)' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW (175,15, 600,7000,4500,32)' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW ( 15,65, 400,7000,4500,32)' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW (100,65, 300,7000,4500,32)' X0,Y0,SCX,SCY,ROT,MODE
  CALL DRAW_ARROW (175,65, 180,7000,4500,32)' X0,Y0,SCX,SCY,ROT,MODE
  PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic in LCD
END

SUB DRAW_ARROW(LONG X0,Y0,SCX,SCY,ROT,MODE) ' <-- draw an Arrow
  DRAW_LINE(SCREEN$,240,128,X0,Y0,-9000,2,-3000,4,SCX,SCY,ROT,MODE,0)
  DRAW_NEXT_LINE (-5800, 6, 0)
  DRAW_NEXT_LINE ( 0, 11, 0)
  DRAW_NEXT_LINE ( 5800, 6, 0)
  DRAW_NEXT_LINE ( 3000, 4, 0)
  DRAW_NEXT_LINE ( 9000, 2, 0)
  CLOSE_LINE (0)
END

```

Programmbeispiel:

```

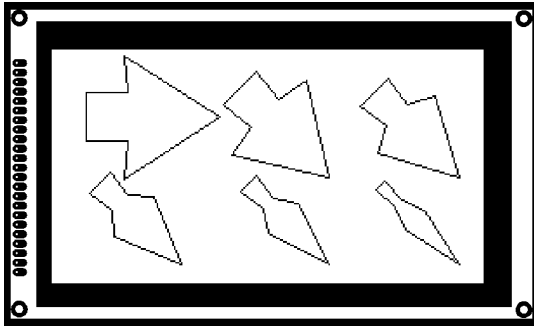
-----
' Name: DRAW_POLAR_ROT_SCALE2.TIG -      Vector graphic
' polar: rotate and scale PHI
-----
' Note: connect L71 to reset LCD
-----
USER_VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string "white"
CALL DRAW_ARROW ( 15,35,1000,7000,0,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (100,15, 800,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (175,15, 600,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW ( 15,65, 400,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (100,65, 300,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW (175,65, 180,7000,4500,32) ' X0,Y0,SCX,SCY,ROT,MODE
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic in LCD
END

SUB DRAW_ARROW(LONG X0,Y0,SCX,SCY,ROT,MODE) ' <-- draw an Arrow
DRAW_LINE(SCREEN$,240,128,X0,Y0,-9000,2,-3000,4,SCX,SCY,ROT,MODE,0)
DRAW_NEXT_LINE (-5800, 6, 0)
DRAW_NEXT_LINE ( 0, 11, 0)
DRAW_NEXT_LINE ( 5800, 6, 0)
DRAW_NEXT_LINE ( 3000, 4, 0)
DRAW_NEXT_LINE ( 9000, 2, 0)
CLOSE_LINE (0)
END

```



4

Polarkoordinaten - R-Skalierung

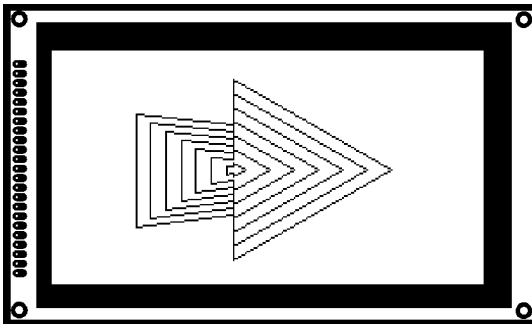
Bei der R-Skalierung (Radien) bleiben alle Winkel erhalten, alle Strecken werden entsprechend dem Skalierung-Faktor (1000=100%) gedehnt. Dehnungszentrum ist der Koordinaten-Ursprung (liegt im Basispunkt lt. DRAW_LINE Funktion).

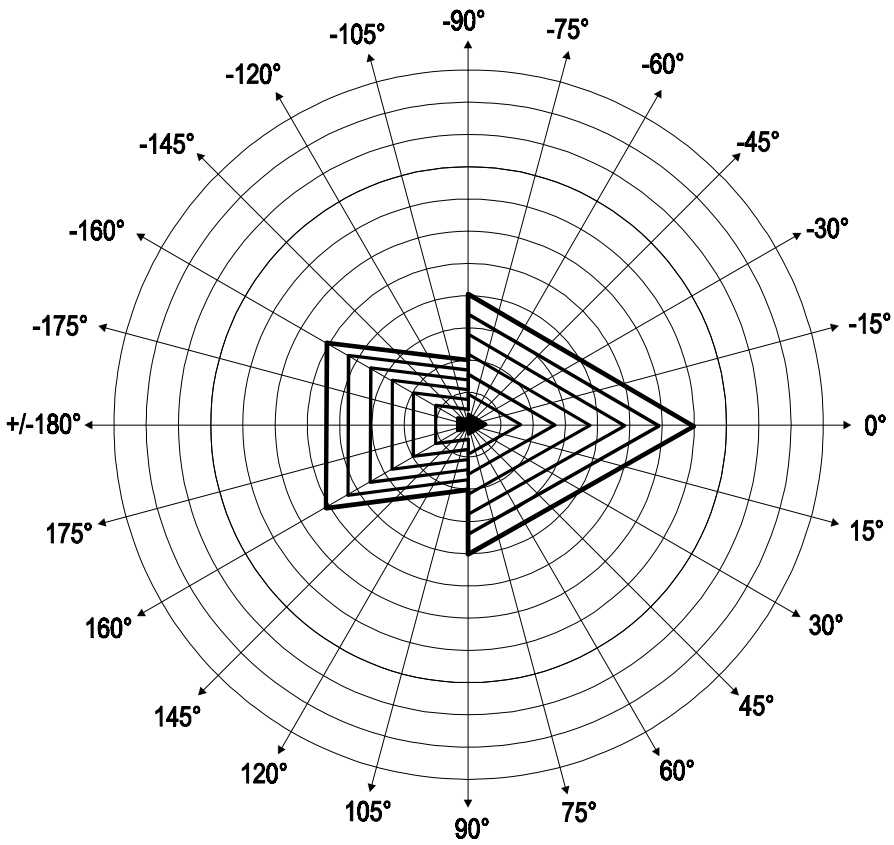
Der Pfeil aus Beispiel VLES_17.TIG liegt in diesen R-Skalierungen vor:

1.	13.000	=	1.300%	= 13-fache Vergrößerung
2.	11.000	=	1.100%	= 11-fache Vergrößerung
3.	9.000	=	900%	= 9-fache Vergrößerung
4.	7.000	=	700%	= 7-fache Vergrößerung
5.	5.000	=	500%	= 5-fache Vergrößerung
6.	3.000	=	300%	= 3-fache Vergrößerung
7.	1.000	=	100%	= 1-fache Vergrößerung

4

Entsprechend erwarten wir folgende Darstellung auf dem LCD:





Pfeil in verschiedenen R-Skalierungen in Beispiel: DRAWPS3.TIG

Programmbeispiel:

```

-----
' Name: DRAW_POLAR_SCALE3.TIG -      Vector graphic
' polar, scale R
-----
' Note: connect L71 to reset LCD
-----
USER_VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b     ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$("<00>",4K)    ' initiate screen-string white
CALL DRAW_ARROW(100,65,1000,13000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000,11000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000, 9000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000, 7000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000, 5000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000, 3000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(100,65,1000, 1000,0,32)' X0,Y0,SCX,SCY,ROT,MODE
PUT #12, #1, SCREEN$, 0,0,3840    ' <-- print graphic to LCD
END

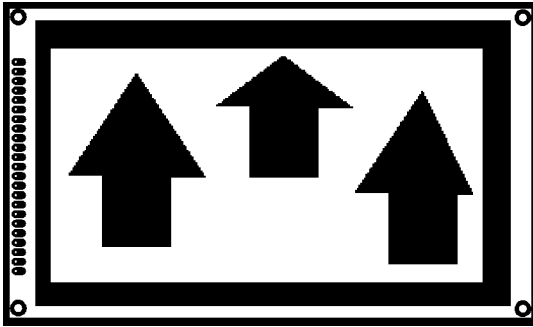
SUB DRAW_ARROW (LONG X0,Y0,SCX,SCY,ROT,MODE)' draw an Arrow
DRAW_LINE(SCREEN$,240,128, X0,Y0,-15000,5,-9000,2,SCX,SCY,ROT,MODE,0)
DRAW_NEXT_LINE (-9000, 4, 0)
DRAW_NEXT_LINE ( 0, 7, 0)
DRAW_NEXT_LINE ( 9000, 4, 0)
DRAW_NEXT_LINE ( 9000, 2, 0)
DRAW_NEXT_LINE (15000, 5, 0)
CLOSE_LINE (0)
END

```

Automatische Begrenzung von Vektorgrafik auf aktiven Grafikbereich

Alle effektiven Koordinaten, d.h. Koordinatenwerte, die nach Verschiebungen und Transformationen entstehen, werden auf den Bereich der aktiven Grafikfläche begrenzt. In Beispiel DRAW_LIM.TIG entspricht die aktive Grafikfläche genau dem sichtbaren Bereich des LC-Bildschirms. Es wird eine Kontur (Pfeil) in 3 Positionen platziert, wobei in 2 Fällen die Kontur über den Rand der aktiven Grafikebene hinausragt und deshalb begrenzt wird. Die Ecke bzw. Spitze des Pfeils wird hierdurch entsprechend verbogen.

4



Programmbeispiel:

```

'-----
' Name: DRAW_LIMIT.TIG - Vector graphic
' touch active graphic area limits
'-----
' Note: connect L71 to reset LCD
'-----
USER_VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b       ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)   ' init screen-string white dots
CALL DRAW_ARROW( 45,70,10000,10000,0,3) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(130,30,10000,10000,0,3) ' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(210,80,10000,10000,0,3) ' X0,Y0,SCX,SCY,ROT,MODE
FILL_AREA (SCREEN$)          ' fill screen_area with "1" Bits
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

SUB DRAW_ARROW (LONG X0,Y0,SCX,SCY,ROT,MODE) ' draw an Arrow
DRAW_LINE (SCREEN$,240,128, X0,Y0, -2,4, -2,0, SCX,SCY,ROT,MODE,0)
DRAW_NEXT_LINE (-4, 0, 0)
DRAW_NEXT_LINE ( 0, -6, 0)
DRAW_NEXT_LINE ( 4, 0, 0)
DRAW_NEXT_LINE ( 2, 0, 0)
DRAW_NEXT_LINE ( 2, 4, 0)
CLOSE_LINE (0)
END

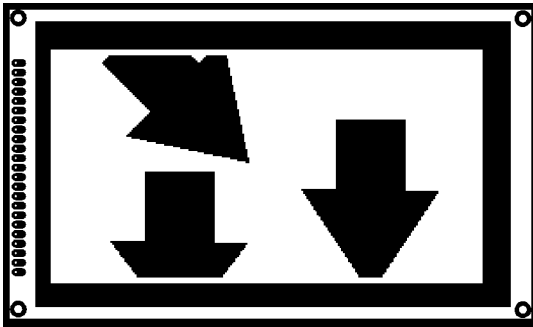
```

Vektorgrafik ohne automatische Begrenzung auf aktiven Grafikbereich

Sollen Vektorgrafiken aufgebaut und über die Grenzen eines Fensters oder Bildschirms hinausreichen, so wird eine entsprechend große Grafikebene verwendet, und der interessierende Teil der Grafik wird ausgeschnitten und in das gewünschte Fenster einkopiert.

Das folgende Beispiel zeigt 3 sich bewegende Pfeile, die ohne Verformungen über den Rand des LC-Bildschirms hinauswandern.

4



Programmbeispiel:

```
' Name: DRAW_UNLIMIT.TIG - Vector graphic
' touch active graphic area limits
-----
' Note: connect L71 to reset LCD
-----
USER VAR STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K), G_AREA$(10K) ' Global Strings for Pixel-data
LONG ANG, X0, Y0, Y02, Y03

TASK MAIN
DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b     ' L71 output: LCD reset
OUT_P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT_P_LCD_RESET,M_LCD_RESET,255 ' "
OUT_P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT_P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT_P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

SCREEN$ = FILL$( "<00>",4K)      ' init screen graphic-string
X0=100
Y0=MODULO_UPDO (100, 40, 160, 2, 0) ' Value, Min, Max, Step, ID
Y02=100
Y03=50
LOOP 999999999
FOR ANG=0 TO 71500 STEP 500
G_AREA$ = FILL$( "<00>",10K)
X0 = MODULO_INC (X0,10,330, 3)      ' init screen: white dots
Y0 = MODULO_UPDO(Y0,0)              ' advance X0-Position
Y02 = MODULO_INC (Y02, 0, 240, 1)   ' advance Y0-Position
Y03 = MODULO_INC (Y03, 0, 240, 1)
CALL DRAW_ARROW( X0, Y0,10000,10000, ANG,3)'X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(130,Y02,10000,10000,18000,3)'X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_ARROW(240,Y03,10000,10000,18000,3)'X0,Y0,SCX,SCY,ROT,MODE
FILL_AREA (G_AREA$)                ' fill screen area with 1-bits

'                                     Format          Format          Window
'                                     <->            <--->           <---->
'                               DESTIN, SRC, B,H, X,Y, B2,H2, X2,Y2, B3,H3,MODE
GRAPHIC_COPY(SCREEN$,G_AREA$,240,128,0,0,360,220,60,60, 240,128, 0)
'                               <----DEST----><-----SOURCE----->

PUT #12, #1, SCREEN$, 0,0,3840      ' print graphic on LCD
NEXT
ENDLOOP
END

SUB DRAW_ARROW (LONG X0,Y0,SCX,SCY,ROT,MODE)' draw Arrow
DRAW_LINE (G_AREA$,360,220,X0,Y0,-2,4, -2,0, SCX,SCY,ROT,MODE,0)'
DRAW_NEXT_LINE (-4, 0, 0)           '
DRAW_NEXT_LINE ( 0, -6, 0)          ' Graphic-area =
```

```
DRAW_NEXT_LINE ( 4, 0, 0)      ' 360 x 220 Dots
DRAW_NEXT_LINE ( 2, 0, 0)      '
DRAW_NEXT_LINE ( 2, 4, 0)      '
CLOSE_LINE (0)                  '
END
```

Gefüllte Flächen und kleine Wiedergabegrößen

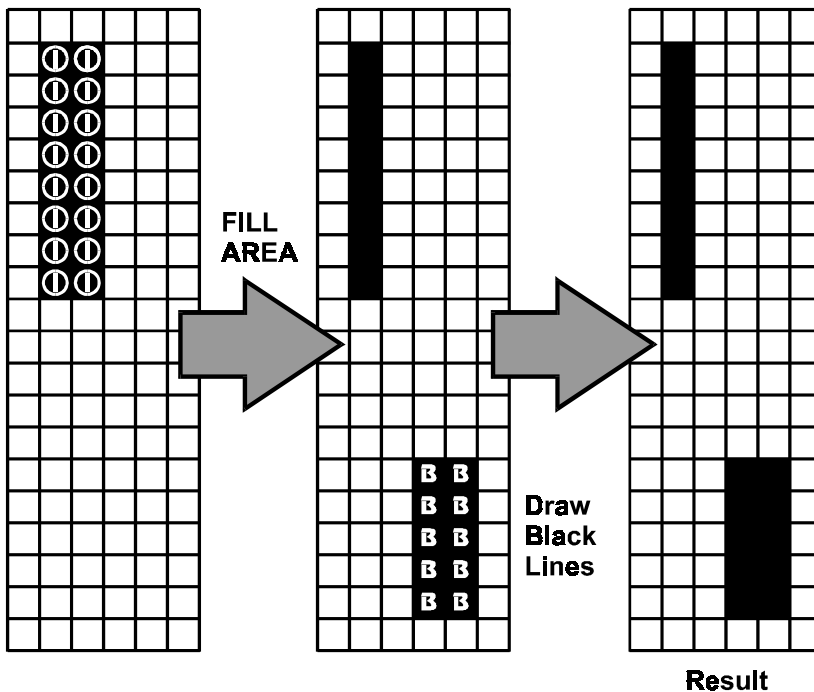
Bei besonders kleinen Darstellungen tritt ein Unterschied zwischen gezeichneten Linien (MODE=0 oder 1 in DRAW_LINE), und gefüllten Flächen (Inversions-Linien, MODE=3 in DRAW_LINE) deutlich zu Tage:

- normale Linien „addieren“ sich: ODER bei „schwarzen“ Linien, AND bei „weißen“ Linien.
- Inversions-Linien arbeiten: „subtraktiv“, also 2 übereinander liegende Linien löschen sich gegenseitig aus.
- Entsprechend erzeugen 2 parallele Linien einer geschlossenen Kontur in beiden Fällen unterschiedliche Resultate.
- Zwei benachbarte, parallele „schwarze“ Linien ergeben ein 2 Pixel breites „schwarzes“ Flächenstück.
- Zwei benachbarte, parallele Inversions-Linien ergeben - nach Anwendung von FILL_AREA - ein 1 Pixel breites „schwarzes“ Flächenstück

4

Diesen Sachverhalt verdeutlicht folgende Vergrößerung:

Inversion-Lines



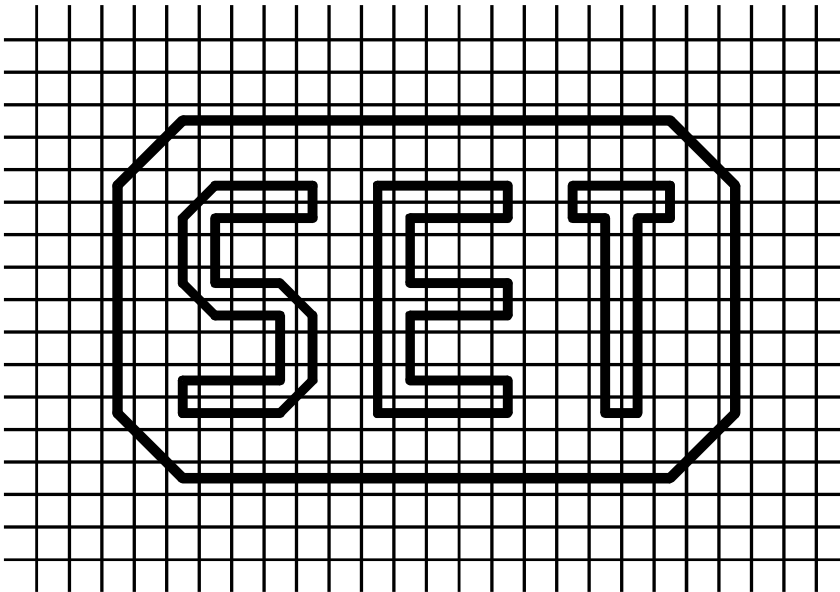
Die Punkte einer Inversions-Linie werden von der FILL_AREA Funktion dazu verwendet, um bei einem waagerechten, zeilenweisen Scan aller Pixel die Farbe des Schreibstrahls im Inversions-Punkt zu invertieren. In obigem Beispiel führt dies dazu, daß der linke obere Inversion-Punkt beim FILL_AREA zu einem schwarzen Ergebnis-Punkt an der gleichen Stelle zu erzeugen. Der ursprüngliche „weiße“ Schreibstrahl wird zu „schwarz“ invertiert.

Der nächste Inversions-Punkt gleich rechts daneben, invertiert den nun „schwarzen“ Schreibstrahl wieder zu „weiß“ und es wird daher ein „weißer“ Ergebnis-Punkt geschrieben.

Anders bei den 2 parallelen, normalen „schwarzen“ Linien, die als ein 2 Pixel breites Flächenstück erscheinen.

Entsprechend ergeben sich auch verschieden breite Strukturen wenn z.B. 2 weiße Pixel zwischen 2 parallelen Linien liegen:

- Die gefüllte Fläche ist 3 Pixel breit,
- die Struktur mit „schwarzen“ Linien bleibt 4 Pixel breit:



Programmbeispiel:

```

'-----
' Name: DRAW_SCALE2.TIG - Vector graphic
' scale to smallest sizes
'-----
' Note: connect L71 to reset LCD
'-----

USER VAR STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

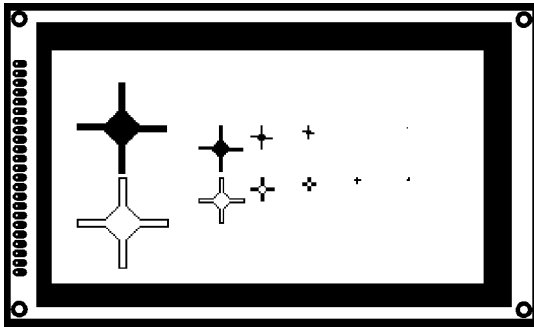
TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' init screen string white dots
CALL DRAW_CROSS ( 10,15,4000,4000,0, 3)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_CROSS ( 80,40,2000,2000,0, 3)' "
CALL DRAW_CROSS (110,40,1000,1000,0, 3)' "
CALL DRAW_CROSS (140,40, 500, 500,0, 3)' "
CALL DRAW_CROSS (170,40, 200, 200,0, 3)' "
CALL DRAW_CROSS (200,40, 100, 100,0, 3)' "
FILL AREA (SCREEN$) ' fill screen-area with 1-bits
CALL DRAW_CROSS ( 10,70,4000,4000,0, 0)' X0,Y0,SCX,SCY,ROT,MODE
CALL DRAW_CROSS ( 80,70,2000,2000,0, 0)' "
CALL DRAW_CROSS (110,70,1000,1000,0, 0)' "
CALL DRAW_CROSS (140,70, 500, 500,0, 0)' "
CALL DRAW_CROSS (170,70, 200, 200,0, 0)' "
CALL DRAW_CROSS (200,70, 100, 100,0, 0)' "
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

SUB DRAW_CROSS (LONG X0,Y0,SCX,SCY,ROT,MODE) ' <-- draw an Arrow
DRAW_LINE (SCREEN$,240,128, X0,Y0, 4,6, 6,4, SCX, SCY, ROT, MODE,0)
DRAW_NEXT_LINE ( 6, 0, 0) '
DRAW_NEXT_LINE ( 7, 0, 0) '
DRAW_NEXT_LINE ( 7, 4, 0) '
DRAW_NEXT_LINE ( 9, 6, 0) '
DRAW_NEXT_LINE ( 9, 6, 0) '
DRAW_NEXT_LINE (13, 6, 0) '
DRAW_NEXT_LINE (13, 7, 0) '
DRAW_NEXT_LINE ( 9, 7, 0) '
DRAW_NEXT_LINE ( 7, 9, 0) '
DRAW_NEXT_LINE ( 7, 13, 0) '
DRAW_NEXT_LINE ( 6, 13, 0) '
DRAW_NEXT_LINE ( 6, 9, 0) '
DRAW_NEXT_LINE ( 4, 7, 0) '
DRAW_NEXT_LINE ( 0, 7, 0) '
DRAW_NEXT_LINE ( 0, 6, 0) '
CLOSE_LINE (0) '

```

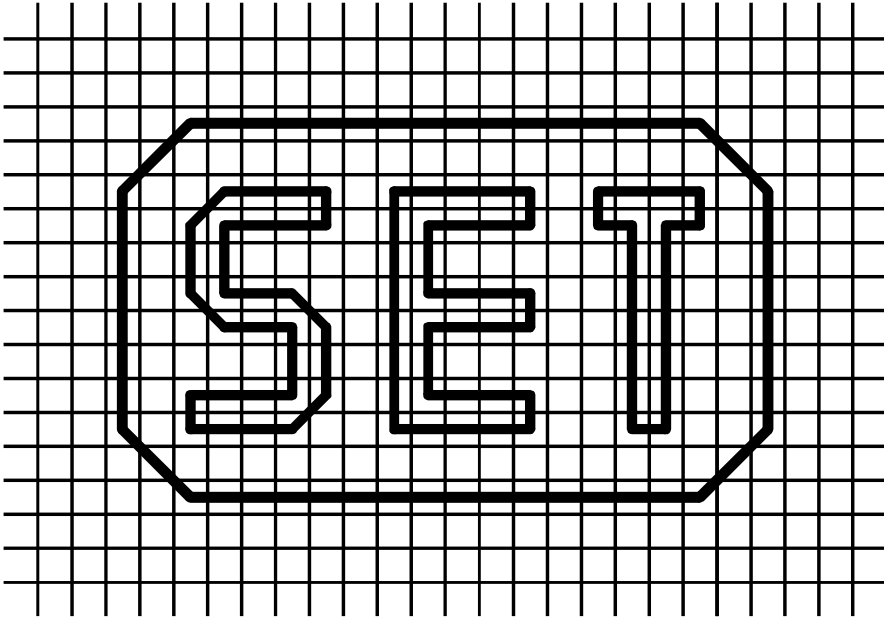
END

4



Kleine Strukturen mit DRAW_LINE und FILL_AREA

DRAW_LINE kann verwendet werden um z.B. skalierbare Schriftzeichen zu zeichnen. Im vorliegenden Fall wurde für die Definition des Schriftzuges „SET“ in einer schwarzen Box ein kleines Raster gewählt, das durch Skalierung ggf. nach Bedarf vergrößert werden kann:


4

Das Beispielprogramm erzeugt auf dem LC-Bildschirm mehrere Ausdrücke des Schriftzuges „SET“ in einer Box in verschiedenen Ausführungen.



Hieran kann man die Besonderheiten von Inversions-Linien und FILL_AREA gut erkennen:

FILL_AREA füllt das gesamte umschlossene Innengebiet sowie **teilweise** die Ränder.

Sollen die Ränder komplett dazugehören, zeichnet man nach FILL_AREA die Kontur nochmals mit „schwarzem“ Stift nach, sollen die Ränder komplett ausgeschlossen werden, überschreibt man die Ränder nach FILL_AREA noch mit einem weißen Stift.

Programmbeispiel:

```

-----
'
' Name: SET_BOX.TIG Vector graphic: scaling + filling + lines: "SET-
Box"
-----
'
' Note: connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC            ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K)            ' Global Strings for Pixel-data

TASK MAIN
  DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H '240x128,150KB/s
  SCREEN$ = FILL$ ("<00>",4K)    ' init screen-string white dots
  CALL DRAW_SETBOX (0, 4000, 10, 30, 3) ' ANGLE, SCALING, X, Y, MODE=
  CALL DRAW_SETBOX (0, 4000, 70, 30, 3) ' " " " " "
  CALL DRAW_SETBOX (0, 4000,130, 30, 3) ' " " " " "
  CALL DRAW_SETBOX (0, 2000, 10,100, 3) ' " " " " "
  CALL DRAW_SETBOX (0, 2000, 40,100, 3) ' " " " " "
  CALL DRAW_SETBOX (0, 2000, 70,100, 3) ' " " " " "

  FILL_AREA (SCREEN$)          ' fill screen-area with 1-bits
  CALL DRAW_SETBOX (0, 4000, 70, 65, 0) ' ANGLE, SCALING, X, Y, MODE=
  CALL DRAW_SETBOX (0, 4000, 70, 30, 0) ' " " " " " black-line
  CALL DRAW_SETBOX (0, 4000,130, 30, 1) ' " " " " " WHITE line
  CALL DRAW_SETBOX (0, 2000, 40,120, 0) ' " " " " " black-line
  CALL DRAW_SETBOX (0, 2000, 40,100, 0) ' " " " " " black-line
  CALL DRAW_SETBOX (0, 2000, 70,100, 1) ' " " " " " WHITE-line
  PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

-----
' Draw the sign plate "SET" in a B O X
-----
SUB DRAW_SETBOX (LONG ANGLE, SCALING, X,Y,MODE) ' Draw sign plate "SET"
  DRAW_LINE(SCREEN$,240,128,X,Y,0,-6,1,-7,SCALING,SCALING,ANGLE,MODE,0)
  DRAW_NEXT_LINE (12, -7, 0)
  DRAW_NEXT_LINE (13, -6, 0)
  DRAW_NEXT_LINE (13, -1, 0) ' this is the box around "SET"
  DRAW_NEXT_LINE (12, 0, 0)
  DRAW_NEXT_LINE ( 1, 0, 0)
  DRAW_NEXT_LINE ( 0, -1, 0)
  CLOSE_LINE (0)
  CALL DRAW_SET (ANGLE, SCALING, X, Y, MODE)
END

```

```

'-----
' Draw the sign plate "SET"
'-----
SUB DRAW_SET (LONG ANGLE, SCALING, X, Y, MODE) ' Draw "SET"
DRAW_LINE (SCREEN$, 240, 128, X, Y, 4, -6, 4, -5, SCALING, SCALING, ANGLE, MODE, 0)
DRAW_NEXT_LINE ( 2, -5, 0)
DRAW_NEXT_LINE ( 2, -4, 0)
DRAW_NEXT_LINE ( 3, -4, 0)
DRAW_NEXT_LINE ( 4, -3, 0) ' this is letter "S"
DRAW_NEXT_LINE ( 4, -2, 0)
DRAW_NEXT_LINE ( 3, -1, 0)
DRAW_NEXT_LINE ( 1, -1, 0)
DRAW_NEXT_LINE ( 1, -2, 0)
DRAW_NEXT_LINE ( 3, -2, 0)
DRAW_NEXT_LINE ( 3, -3, 0)
DRAW_NEXT_LINE ( 2, -3, 0)
DRAW_NEXT_LINE ( 1, -4, 0)
DRAW_NEXT_LINE ( 1, -5, 0)
DRAW_NEXT_LINE ( 2, -6, 0)
CLOSE_LINE (0)

DRAW_LINE (SCREEN$, 240, 128, X, Y, 8, -6, 8, -5, SCALING, SCALING, ANGLE, MODE, 0)
DRAW_NEXT_LINE ( 6, -5, 0)
DRAW_NEXT_LINE ( 6, -4, 0)
DRAW_NEXT_LINE ( 8, -4, 0)
DRAW_NEXT_LINE ( 8, -3, 0) ' this is letter "E"
DRAW_NEXT_LINE ( 6, -3, 0)
DRAW_NEXT_LINE ( 6, -2, 0)
DRAW_NEXT_LINE ( 8, -2, 0)
DRAW_NEXT_LINE ( 8, -1, 0)
DRAW_NEXT_LINE ( 5, -1, 0)
DRAW_NEXT_LINE ( 5, -6, 0)
CLOSE_LINE (0)

DRAW_LINE (SCREEN$, 240, 128, X, Y, 12, -6, 12, -5, SCALING, SCALING, ANGLE, MODE, 0)
DRAW_NEXT_LINE (11, -5, 0)
DRAW_NEXT_LINE (11, -1, 0)
DRAW_NEXT_LINE (10, -1, 0)
DRAW_NEXT_LINE (10, -5, 0) ' this is letter "T"
DRAW_NEXT_LINE ( 9, -5, 0)
DRAW_NEXT_LINE ( 9, -6, 0)
CLOSE_LINE (0)
END

```

Dicke Linien

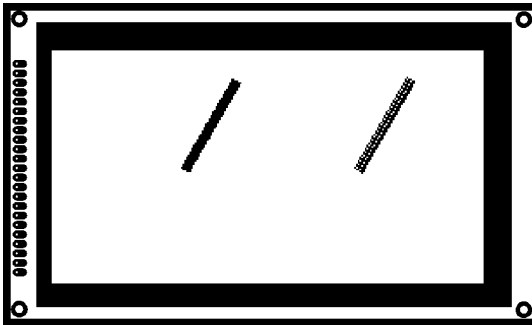
„Schwarze“ oder „weiße“ Linien besitzen stets die Dicke von einem Pixel. Sollen Strukturen mit dickeren Linien in Vektorgrafik gezeichnet werden so bieten sich 2 Möglichkeiten an:

1. Die Struktur mit dicken Linien wird mit Inversions-Linien und FILL_AREA erzeugt, oder
2. durch mehrfaches, paralleles Zeichnen von Linien wird eine dickere Linie erzeugt.

Im zweiten Fall ist jedoch zu beachten, daß die mehrmals gezeichneten 1 Pixel dicken Linien auch wirklich parallel und **lückenlos** nebeneinander liegen. Dies ist z.B. immer dann nicht gewährleistet, wenn unterschiedliche Rundungsfehler zwischen den verschiedenen Linien bestehen (treten z.B. auf bei Rotation oder Koordinaten-Transformationen).

Das vorliegende Beispiel zeigt die Auswirkungen dieses Falles.

Im Beispielprogramm THICKLI2.TIG wird gezeigt, wie man es richtig macht.



Programmbeispiel:

4

```

'-----
' Name: THICK LINE1.TIG - Vector graphic
' scaling + filling + lines
'-----

' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K), SCREEN2$(4K) ' Global Strings for Pixel-data
LONG ANG

TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' LCD-4=240x128,
150KB/s

FOR ANG=0 TO 36000*10 STEP 600
SCREEN$ = FILL$ ("<00>",4K) ' init screen-string "white"
dots
CALL DRAW_THICK_LINE(ANG,6000,70,64,3)' AN-
GLE,XSCALING,X,Y,MODE=invers.
FILL AREA (SCREEN$) ' fill screen-area with "1" Bits
CALL
DRAW_MANY_LINES(ANG,6000,170,64,0)'ANGLE,XSCALING,X,Y,MODE=invers.
SCREEN2$=SCREEN$
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
WAIT_DURATION 1000
NEXT
END

'-----
' Draw a thick line - for demo. actually a box that gets filled later
'-----

SUB DRAW_THICK_LINE (LONG ANGLE, SCALING, X, Y, MODE)
DRAW_LINE (SCREEN$,240,128,X,Y, 0,0, 1,0, SCAL-
ING,SCALING,ANGLE,MODE,0)
DRAW_NEXT_LINE ( 1,-10, 0)
DRAW_NEXT_LINE ( 0,-10, 0)
CLOSE_LINE (0)
END

'-----
' Draw many lines - for demo

```

DRAW_LINE

```
'-----  
-  
SUB DRAW_MANY_LINES (LONG_ANGLE, SCALING, X, Y, MODE)  
  DRAW_LINE (SCREEN$, 240, 128, X, Y, 0, 0, 0, -10, 1000, SCALING, ANGLE, MODE  
, 0)  
  DRAW_NEXT_LINE ( 1, -10, 0)      '  
  DRAW_NEXT_LINE ( 1,  0, 0)      '  
  DRAW_NEXT_LINE ( 2,  0, 0)      ' Many individual line segments  
  DRAW_NEXT_LINE ( 2, -10, 0)      ' without filling  
  DRAW_NEXT_LINE ( 3, -10, 0)      '  
  DRAW_NEXT_LINE ( 3,  0, 0)      '  
  DRAW_NEXT_LINE ( 4,  0, 0)      '  
  DRAW_NEXT_LINE ( 4, -10, 0)      '  
  DRAW_NEXT_LINE ( 5, -10, 0)      '  
  DRAW_NEXT_LINE ( 5,  0, 0)      '  
END
```

4

Beispiel: Dicke Linien (2)

Hier werden dicke Linien gezeichnet, indem man mehrere Linien parallel zueinander und ohne Lücke zeichnet.



4

Programmbeispiel:

```

-----
' Name: THICK_LINE2.TIG -           Vector graphic
' draw (flat) thick lines
-----
' Note: connect L71 to reset LCD
-----

USER_VAR STRICT
#include DEFINE_A.INC           ' general definitions
#include UFUNC3.INC             ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC             ' definitions for Graphic Toolkit

STRING SCREEN$(4K)              ' Global Strings for Pixel-data
LONG X,X2,X3

TASK Main                        ' begin task MAIN
DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0  ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0     ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255    ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s

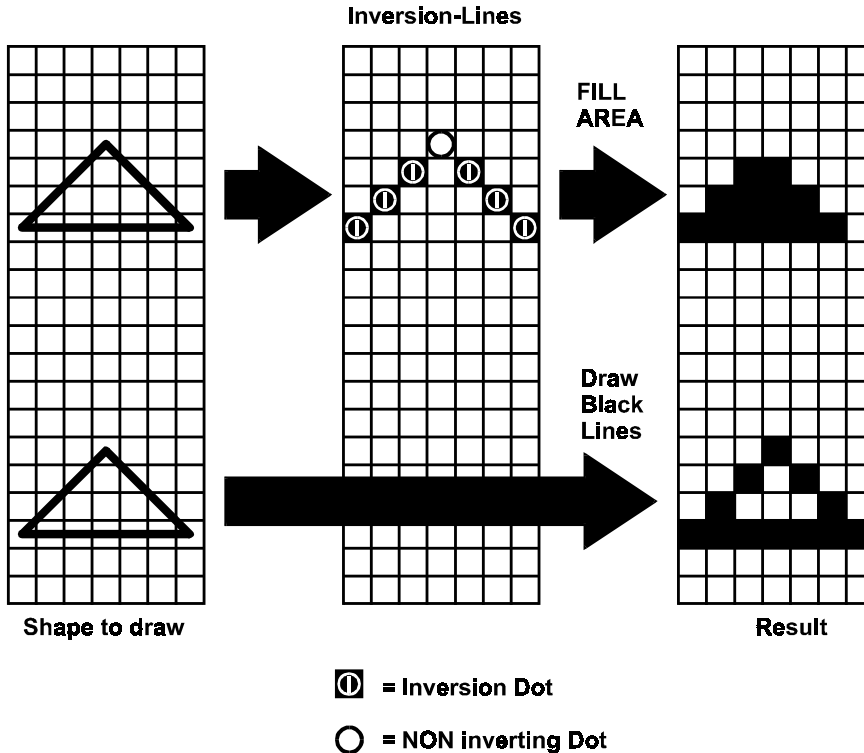
FOR X=-20 TO 170 STEP 1
  X2=X-70
  X3=X-130
  SCREEN$ = FILL$("<00>",4K)      ' screen <-- "white" dots
  CALL THICK_LINE ( 30,10, X,100, 7)' X0,Y0,RX1,RY1,THICK
  CALL THICK_LINE (110,10, X2,100, 15)' X0,Y0,RX1,RY1,THICK
  CALL THICK_LINE (180,10, X3,100, 35)' X0,Y0,RX1,RY1,THICK
  PUT #12, #1, SCREEN$, 0,0,3840    ' <-- print graphic to LCD
  WAIT_DURATION 50
NEXT
END

SUB THICK_LINE(LONG X0,Y0,RX1,RY1,THICK)' draw thick line to be filled
LONG N, XX1                      ' later with "FILL_AREA"
  XX1=RX1
  DRAW_LINE (SCREEN$,240,128,X0,Y0, 0,0, RX1,RY1, 1000,1000, 0,0,0)
  XX1=XX1+1
  FOR N=1 TO THICK-1 STEP 2
    DRAW_NEXT_LINE ( XX1, RY1,0)
    DRAW_NEXT_LINE ( N, 0,0)
    DRAW_NEXT_LINE ( N+1, 0,0)
    DRAW_NEXT_LINE ( XX1+1, RY1,0)
    XX1=XX1+2
  NEXT
END

```

Spitze Konturen

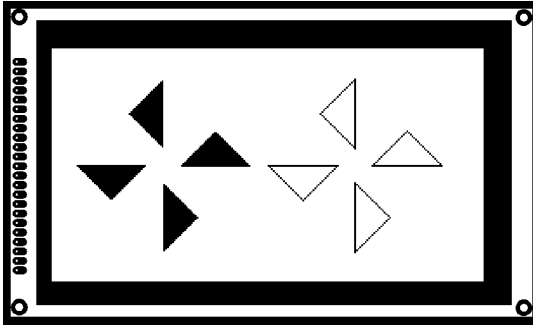
In diesem Beispiel wird auf das Zeichnen von spitzen Konturen und Flächen eingegangen. Die folgende Abbildung zeigt schematisch den Vorgang beim Zeichnen von Linien und Flächen:



Beachten Sie die Spitze der Dreiecke in beiden Fällen: beim Zeichnen mit Inversions-Linien ergibt sich aus der spitzen Dreiecks-Spitze eine abgeflachte Spitze, was wie schon beschrieben, daran liegt, daß der rechte Rand einer gefüllten Kontur nicht mehr zur gefüllten Fläche zählt.

Im Gegensatz dazu, wird beim Zeichnen mit „schwarzem“ (oder „weißem“) Stift die Spitze wie erwartet ausgeführt. Wird eine Fläche einschließlich der Randlinie gewünscht, so zeichnet man nach dem Herstellen der gefüllten Fläche noch einmal die Randkurve.

Das Beispielprogramm zeigt diesen Sachverhalt.



Programmbeispiel:

4

```

'-----
' Name: NOSEPEAK.TIG      -   Vector graphic
' shows how to get - or not to get - nose peaks
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K), SCREEN2$(4K)      ' global strings: pixel data

TASK MAIN
DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b     ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0      ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255    ' "
OUT P_LCD_CTRL,M_INVERT,255        ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0        ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255       ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s

SCREEN$ = FILL$("<00>",4K)          'init screen string white dots
CALL DRAW_45_NOSE ( 0,4000, 80,64,3)' ANGLE,SCALE,X,Y, MODE=invers
CALL DRAW_45_NOSE ( 9000,4000, 70,74,3)' "
CALL DRAW_45_NOSE (18000,4000, 60,64,3)' "
CALL DRAW_45_NOSE (27000,4000, 70,54,3)' "
FILL_AREA (SCREEN$)              ' fill screen area with 1-bits
CALL DRAW_45_NOSE ( 0,4000,180,64,0)' ANGLE,SCALE,X,Y,BLACK line
CALL DRAW_45_NOSE ( 9000,4000,170,74,0)' "
CALL DRAW_45_NOSE (18000,4000,160,64,0)' "
CALL DRAW_45_NOSE (27000,4000,170,54,0)' "
PUT #12, #1, SCREEN$, 0,0,3840    ' print graphic on LCD
END

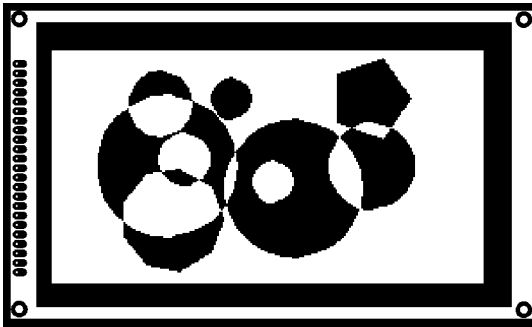
SUB DRAW_45_NOSE (LONG ROT,SCALING,X0,Y0,MODE)' draw a nose
DRAW_LINE (SCREEN$,240,128,X0,Y0,0,0,10,0,SCALING,SCALING,ROT,MODE,0)
DRAW_NEXT_LINE (5, -5, 0)
CLOSE_LINE (0)
END

```

Überlagerung von Flächen

In diesem Beispiel wird - zufallsgesteuert - eine Anzahl von N-Eck-Flächen gezeichnet, die sich teilweise überlappen. Wie man sieht wird der Farbwert jeder Fläche an ihrem Rand invertiert. Dabei spielt es keine Rolle wie viele Flächen ineinander verschachtelt sind, sich teilweise überlappen, oder gemeinsame Randkurven besitzen mit anderen Flächen.

An Hand dieses LCD Ausdrucks kann man auch feststellen, daß je nach dargestellter Größe in dieser Auflösung bereits 16- bis 32-Ecke näherungsweise als Kreise angesehen werden können.

**4**

Programmbeispiel:

4

```

'-----
' Name: OVERLAP.TIG - Vector graphic
' many filled N-ogons
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K), SCREEN2$(4K)  ' Global Strings for Pixel-data
LONG RAD, APPRX, X, Y, N

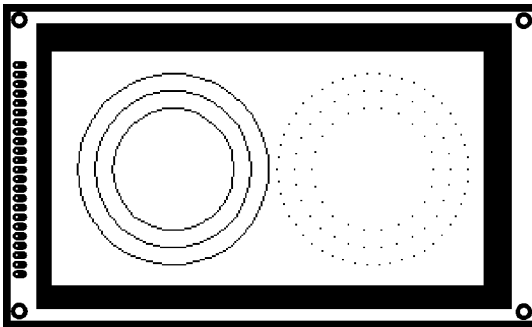
TASK MAIN
  DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b    ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0  ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255  ' "
  OUT P_LCD_CTRL,M_INVERT,255      ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0      ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255     ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
  LOOP 9999999000             ' many loops
  SCREEN$ = FILL$ ("<00>",4K)   ' initiate screen to white dots
  LOOP 9
    RAD  =(MOD(RND(N),45))      ' radius
    APPRX =(MOD(RND(N),32))+3    ' approximation steps
    X     =(MOD(RND(N),240-2*RAD))+RAD  ' X-Position (Center)
    Y     =(MOD(RND(N),128-2*RAD))+RAD  ' Y-Position (Center)
    CALL CIRCLE (RAD, APPRX, X, Y,3)  ' RADIUS,APPROXIMATION,X,Y,Mode
  ENDLOOP
  FILL_AREA (SCREEN$)           ' fill screen with "1" Bits
  SCREEN2$=SCREEN$
  PUT #12, #1, SCREEN2$, 0,0,3840  ' print graphic on LCD
ENDLOOP
END

SUB CIRCLE (LONG R, APP, X0, Y0, MODE)' RADIUS,APPROXIMATION,X,Y,Mode
LONG ANGLE, ASTEP
  ASTEP = 36000/APP
  DRAW_LINE (SCREEN$,240,128, X0,Y0, 0,R,ASTEP,R,1000,1000,0,MODE+32,0)
  FOR ANGLE =2*ASTEP TO 36000-ASTEP STEP ASTEP
    DRAW_NEXT_LINE ( ANGLE,R, 0)
  NEXT
  CLOSE_LINE (0)
END

```

Zeichen mit „unsichtbarer“ Linie als Test.

Um einen Eindruck zu vermitteln, wie der Mode-Parameter = 2 arbeitet, hier eine beispielhafte Darstellung, die die Aufteilung eines Kurvenzuges in einzelne Liniensegmente verdeutlicht. So kann man besser beurteilen, ob man z.B. eine feinere oder gröbere Aufteilung von Randkurven vornehmen sollte, um optimale Ergebnisse zu erhalten.

**4**

Programmbeispiel:

```

'-----
' Name: INVISIBLE.TIG      -   Vector graphic
' using invisible lines
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)   ' initiate screen to "white" dots
CALL CIRCLE (55, 50, 65, 64,0)' RADIUS, APPROXIM, X,Y,MODE
CALL CIRCLE (55, 50, 180, 64,2)' RADIUS, APPROXIM, X,Y,Mode=2 (invis)

CALL CIRCLE (45, 30, 65, 64,0)' RADIUS, APPROXIM, X,Y,MODE
CALL CIRCLE (45, 30, 180, 64,2)' RADIUS, APPROXIM, X,Y,Mode=2 (invis)

CALL CIRCLE (35, 22, 65, 64,0)' RADIUS, APPROXIM, X,Y,MODE
CALL CIRCLE (35, 22, 180, 64,2)' RADIUS, APPROXIM, X,Y,Mode=2 (invis)
PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
END

SUB CIRCLE (LONG R, APP, X0, Y0, MODE)' RADIUS,APPROXIMATION,X,Y,Mode
LONG ANGLE, ASTEP
ASTEP = 36000/APP
DRAW_LINE (SCREEN$,240,128, X0,Y0, 0,R,ASTEP,R,1000,1000,0,MODE+32,0)
FOR ANGLE =2*ASTEP TO 36000-ASTEP STEP ASTEP
    DRAW_NEXT_LINE ( ANGLE,R, 0)
NEXT
CLOSE_LINE (0)
END

```


DRAW_NEXT_LINE

DRAW_NEXT_LINE (X1, Y1, Pen)

DRAW_NEXT_LINE: Fügt weiteres Linienelement an Linie an
(nur in Projektmodell 'PM_FULL')

Vektorgrafik. Führt die Linie mit Stift-Nr. **Pen** mit den z.Zt. gültigen Parameter-Einstellungen fort. Diese Funktion bearbeitet den Argumentstring String **Str1**. Es findet keine Zuweisung statt.

X1, Y1

sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und beschreiben den neuen Punkt der Linie relativ zum „Basispunkt“ (X0,Y0) in
a) kartesischen Koordinaten im Wertebereich: -8191...0...8191, kleinste Einheit: 1 Pixel.
oder
b) Polarkoordinaten: Winkel (PHI), Radius (R). Wertebereich Winkel: -36.000...0...36.000, kleinste Einheit: 1=0,01-Grad, Wertebereich Radius: 0...8191, kleinste Einheit = 1 Pixel

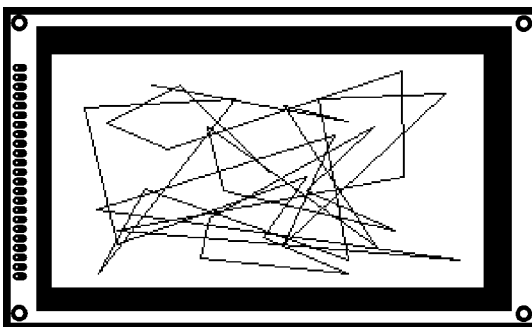
Pen

ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Durch die Stift-Nr. wird eine Referenz zu einem Parametersatz hergestellt, wie er unter der gleichen Stift-Nr. zuvor in der DRAW_LINE Funktion spezifiziert worden ist. Insgesamt können gleichzeitig bis zu 16 unterschiedliche Stifte / Parametersätze verwendet werden.

Funktionswert

gibt es bei dieser Funktion nicht. Die Daten in **Str1** werden direkt verändert.

DRAW_NEXT_LINE arbeitet eng mit der DRAW_LINE Funktion zusammen. Dort werden alle Parameter-Einstellungen erläutert und mit Beispielprogrammen demonstriert.



Programmbeispiel:

4

```

'-----
' Name: DRAW_NEXT_LINE2.TIG - Vector graphic: draw a polygon line
' draws more lines using DRAW_NEXT_LINE
'-----
' Note: connect L71 to reset LCD
'-----

USER VAR STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC             ' definitions for LCD Typ 4
#include GR_TK1.INC            ' definitions for Graphic Toolkit

STRING SCREEN$(4K)            ' Global Strings for Pixel-data

TASK Main                      ' begin task MAIN
  DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b        ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
  SCREEN$ = FILL$ ("<00>",4K)   ' initiate screen-string: white dots
'-----
'          <-Format-> Base   Pnt-1 Pnt-2<scaling>Rot Draw Pen
'          Destin,wide high X Y   X Y X Y   X   Y ang Mode No
DRAW_LINE (SCREEN$,240,128,120,60,-77,0,50,-70,1000,1000, 0, 0, 0)

DRAW_NEXT_LINE ( 33, 50,0)      ' X= 33, Y= 50 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (-99,-40,0)      ' X=-99, Y=-40 rel to BASE pt.(120,60)
DRAW_NEXT_LINE (111, 40,0)      ' X=111, Y= 40 rel to BASE pt.(120,60)

PUT #12, #1, SCREEN$,0,0,3840 ' <-- print graphic on LCD
END

```

CLOSE_LINE

CLOSE_LINE (*Pen*)

CLOSE_LINE: Schließt einen offenen Linienzug
(nur in Projektmodell 'PM_FULL')

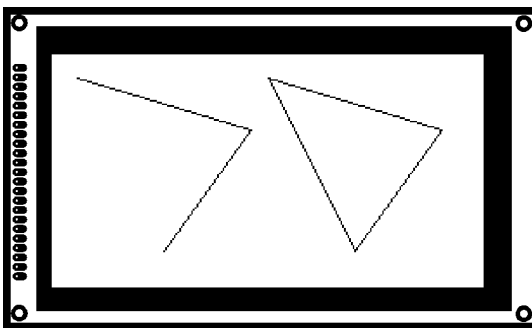
Vektorgrafik. CLOSE_LINE schließt eine mit DRAW_LINE gezeichneten Kurvenzug des Grafiksets **Pen**. Der letzte Punkt eines Linienzugs wird mit dem ersten verbunden. Diese Funktion bearbeitet den Argumentstring String **Str1**. Es findet keine Zuweisung statt.

Pen ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Durch die Stift-Nr. wird eine Referenz zu einem Parametersatz hergestellt, wie er unter der gleichen Stift-Nr. zuvor in der DRAW_LINE Funktion spezifiziert worden ist. Insgesamt können gleichzeitig bis zu 16 unterschiedliche Stifte / Parametersätze verwendet werden.

Funktionswert gibt es bei dieser Funktion nicht. Die Daten im betroffenen String werden direkt verändert.

Für das Füllen von grafischen Flächen ist das Beenden eines Kurvenzuges durch die CLOSE_LINE Funktion obligatorisch. In allen anderen Fällen kann ein Linienzug offen bleiben oder geschlossen werden.

DRAW_NEXT_LINE arbeitet eng zusammen mit der CLOSE_LINE Funktion. Dort werden alle Parameter-Einstellungen erläutert und mit Beispiel-Programmen demonstriert.



Programmbeispiel:

```

'-----
' Name: CLOSE_LINE.TIG - Vector graphic: CLOSE_LINE demo
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

TASK MAIN
DIR PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR PORT 7,11111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' initiate screen to white dots

DRAW_LINE (SCREEN$,240,128, 10,10, 0,0, 100,30, 1000,1000, 0,0,0)
DRAW_NEXT_LINE ( 50,100, 0) ' <-- o p e n line

DRAW_LINE (SCREEN$,240,128, 120,10, 0,0, 100,30, 1000,1000, 0,0,0)
DRAW_NEXT_LINE ( 50,100, 0) '
CLOSE_LINE (0) ' <-- c l o s e d line

PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

```

Siehe auch: DRAW_LINE, DRAW_NEXTLINE

SET_BASE**SET_BASE (X0, Y0, PEN)**

SET_BASE: Setzt den Koordinaten-Ursprung neu
(nur in Projektmodell 'PM_FULL')

Instruktion: setzt für den Stift **Pen** einen neuen Koordinaten-Ursprung bei **X0, Y0**.

X0, Y0

sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und beschreiben den Basispunkt in kartesischen-Koordinaten im Wertebereich: -8191...0...8191. Die Koordinaten aller nachfolgenden Linienpunkte beziehen sich auf diesen Basispunkt, der beliebig in allen 4 Quadranten positioniert werden darf. Diese Angabe erfolgt stets in kartesischen Koordinaten, auch wenn das polare Koordinatensystem für die nachfolgenden Linienpunkte gewählt wird (siehe: MODE → POLAR).

Alle Punkte der Vektorgrafik werden automatisch auf den aktiven Grafikbereich begrenzt, kein Linienpunkt kann außerhalb dieses Bereichs liegen. Wird also z.B. der Basispunkt außerhalb des aktiven Grafikbereichs plaziert, so benötigen die Linienpunkte entsprechende relative Koordinaten, so daß sie, zum Wert dieses Basispunktes addiert, tatsächlich in den aktiven Grafikbereich fallen.

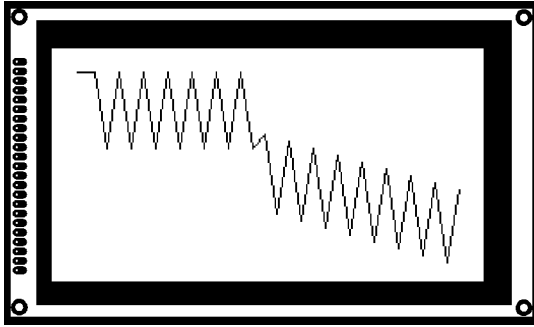
Pen

ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Durch die Stift-Nr. wird eine Referenz zu einem Parametersatz hergestellt, wie er unter der gleichen Stift-Nr. zuvor in der DRAW_LINE Funktion spezifiziert worden ist. Insgesamt können gleichzeitig bis zu 16 unterschiedliche Stifte / Parametersätze verwendet werden.

Funktionswert

gibt es bei dieser Funktion nicht. Die Einstellung im Parametersatz für diesen Pen wird direkt verändert.

SET_BASE verschiebt den Basispunkt, auch während ein Linienzug gezeichnet wird.



4

Programmbeispiel:

```

-----
' Name: SET_BASE.TIG - Vector graphic: SET_BASE demo
-----
' Note: connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC            ' definitions for user function codes
#include LCD_4.INC              ' definitions for LCD Typ 4
#include GR_TK1.INC             ' definitions for Graphic Toolkit

STRING SCREEN$(4K)             ' Global Strings for Pixel-data
LONG N

TASK MAIN
DIR_PORT 8,1000000b             ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b           ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255     ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0     ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255    ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150 KB/s
SCREEN$ = FILL$ ("<00>",4K)     ' initiate screen to white dots

DRAW_LINE (SCREEN$,240,128, 10,10, 0,0, 10,0, 1000,1000, 0,0,0)
FOR N=1 TO 30
  DRAW_NEXT_LINE (10+N*7,(N BITAND 1)*44, 0)' X1, Y1, PEN
  IF N>12 THEN
    SET_BASE (10,20+N*2,0)       ' reset base point to new pos.
  ENDIF
NEXT
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

```

SET_SCALE

SET_SCALE (XScale, YScale, Pen)

SET_SCALE: Setzt X-/Y-Skalierungs-Faktoren neu
(nur in Projektmodell 'PM_FULL')

Setzt den Skalierungsfaktor in X-Richtung und in Y-Richtung.

XScale ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Skalierungsfaktor in X-Richtung in zehntel Prozent an. Wertebereich: 10 ... 30.000 (1%...3000%). Bei Polarkoordinaten bezieht sich die Skalierung auf den Winkel.

YScale ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und gibt den Skalierungsfaktor in Y-Richtung in zehntel Prozent an. Wertebereich: 10 ... 30.000 (1%...3000%). Bei Polarkoordinaten bezieht sich die Skalierung auf den Radius.

Pen ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Durch die Stift-Nr. wird eine Referenz zu einem Parametersatz hergestellt, wie er unter der gleichen Stift-Nr. zuvor in der DRAW_LINE Funktion spezifiziert worden ist. Insgesamt können gleichzeitig bis zu 16 unterschiedliche Stifte / Parametersätze verwendet werden.

Funktionswert gibt es bei dieser Funktion nicht. Die Einstellung im Parametersatz für diesen Pen wird direkt verändert.

SET_SCALE verändert die aktuelle Einstellung der Skalierungsfaktoren, auch während ein Linienzug gezeichnet wird.

Programmbeispiel:

```

-----
' Name: SET_SCALE.TIG      -   Vector graphic:  SET_SCALE demo
-----
' Note: connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

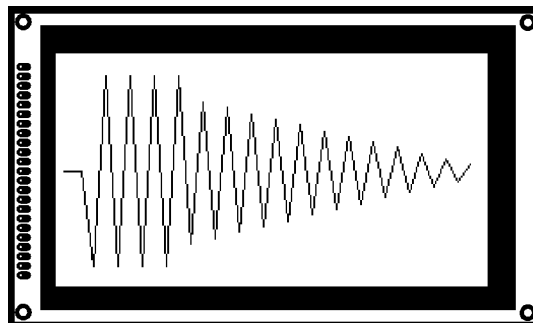
STRING SCREEN$(4K)        ' Global Strings for Pixel-data
LONG N

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255   ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0   ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255  ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K)    ' initiate screen to "white" dots

DRAW_LINE (SCREEN$,240,128, 0,65, 0,0, 10,0, 1000,1000, 0,0,0)
FOR N=1 TO 32
  DRAW_NEXT_LINE (10+N*7,(((N BITAND 1)*2)-1)*55, 0) ' X1, Y1, PEN
  IF N>7 THEN
    SET_SCALE (1000,1000-N*30,0) ' Re-Scale next line-drawing
  ENDIF
NEXT
PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
END

```

4



SET_ROTATION

SET_ROTATION (*Rotation*, *Pen*)

SET_ROTATION: Setzt Schreibstift Rotation
(nur in Projektmodell 'PM_FULL')

Setzt den Wert für Rotation in X-Richtung und in Y-Richtung.

Rotation

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist der Rotationswinkel in 0,01-Grad Schritten gegen den Uhrzeigersinn. Wertebereich: -36.000...0...36.000, 1=0,01 Grad, 100=1-Grad.

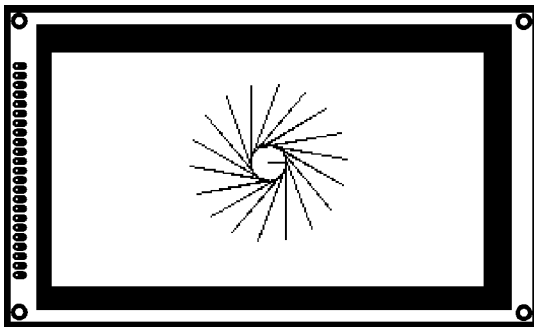
Pen

ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Es können bis zu 16 Stifte mit unterschiedlichen Einstellungen gleichzeitig verwendet werden. Alle zuvor in dieser Funktion getroffenen Einstellungen können bei jedem der max. 16 Stifte übereinstimmen oder sich voneinander unterscheiden.

Funktionswert

gibt es bei dieser Funktion nicht. Die Einstellung im Parametersatz für diesen Pen wird direkt verändert.

In den folgenden zwei Beispielen wird der Rotationswinkel im FOR-Loop jeweils neu gesetzt. Entsprechend wird die Orientierungsrichtung des Schreibstiftes abgelenkt.



Programmbeispiel:

```

-----
' Name: SET_ROTATE1.TIG - Vector graphic: SET_ROTATION demo
-----
' Note: connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data
LONG N

TASK MAIN
DIR_PORT 8,1000000b        ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b      ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150 KB/s
SCREEN$ = FILL$ ("<00>",4K) ' initiate screen to "white" dots

DRAW_LINE (SCREEN$,240,128, 120,60, 0,0, 10,0, 1000,1000, 0,0,0)
FOR N=1 TO 35
  DRAW_NEXT_LINE (10,(N BITAND 1)*44,0) ' X1, Y1, PEN
  SET_ROTATION (N*1000,0) ' reset ROTATION angle
NEXT
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

```

Programmbeispiel:

```

'-----
' Name: SET_ROTATE2.TIG - Vector graphic: SET_ROTATION demo
'-----
' Note: connect L71 to reset LCD
'-----

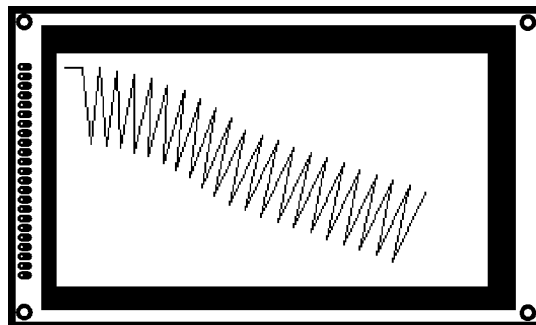
USER_VAR_STRICT
#include DEFINE_A.INC          ' general definitions
#include UFUNC3.INC           ' definitions for user function codes
#include LCD_4.INC            ' definitions for LCD Typ 4
#include GR_TK1.INC           ' definitions for Graphic Toolkit

STRING SCREEN$(4K)           ' Global Strings for Pixel-data
LONG N

TASK MAIN
DIR_PORT 8,1000000b          ' L80...L86 output: LCD ctrl
DIR_PORT 7,11111101b         ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H'240x128,150 KB/s
SCREEN$ = FILL$( "<00>",4K)    ' initiate screen to "white" dots

DRAW_LINE (SCREEN$,240,128, 0,5, 0,0, 10,0, 1000,1000, 0,0,0)
FOR N=1 TO 42
  DRAW_NEXT_LINE (10+N*5,(N BITAND 1)*44,0)' X1, Y1, PEN
  IF N<20 THEN
    SET_ROTATION (N*100,0)      ' reset ROTATION angle
  ENDIF
NEXT
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic to LCD
END

```



SET_GRAREA**SET_GRAREA (Dest, FB, FH, Pen)**

SET_GRAREA: Setzt Grafik-Area
(nur in Projektmodell 'PM_FULL')

Setzt eine Grafik-Area: einer Stift-Nr. wird ein Zielstring sowie die Formatbreite und Formathöhe zugeordnet.

Dest ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält Pixelgrafikdaten. Der hier angegebene String ist gleichzeitig der Zielstring. Die DRAW_LINE Funktion erwartet, daß dieser String bereits genügend Bytes enthält, um die Grafik mit der angegebenen Formatgröße (FB x FH / 8 Bytes) vollständig aufzunehmen. Üblicherweise wird dieser Zielstring vor der ersten Bearbeitung mit einer Pixelgrafik gefüllt bzw. mit der FILL-Funktion, z.B: SCREEN\$ = FILL\$ ("<00>",4K) einheitlich mit „weißen“ bzw. „schwarzen“ Hintergrund-Pixeln gefüllt.

FB ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formatbreite in Pixeln. FB ist stets ein Vielfaches von 8: 0...nnnn. FB gibt die Breite des Grafikbereiches an, in dem man arbeiten möchte. Dies kann die Breite des Grafikbildschirms oder Druckers sein, auf dem die Grafik ausgegeben werden soll. FB kann jedoch auch größer oder kleiner gewählt werden entsprechend der gewünschten Größe der Grafikebene in der man arbeiten möchte.

FH ist eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Formathöhe in Pixeln: 0...nnnn, in 1 Pixel-Schritten.

Durch die Angabe von Formatbreite und -Höhe legt man den aktiven Grafikbereich fest. Nur innerhalb dieses Bereiches kann man Linien ziehen und Pixel sichtbar machen.

Pen ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Es können bis zu 16 Stifte mit unterschiedlichen Einstellungen gleichzeitig verwendet werden. Alle zuvor in dieser Funktion getroffenen Einstellungen können bei jedem der max. 16 Stifte übereinstimmen oder sich voneinander unterscheiden.

Funktionswert gibt es bei dieser Funktion nicht. Die Einstellung im Parametersatz für diesen Pen wird direkt verändert.

SET_GRAREA wird z.B. vor einem SET_DOT eingesetzt um einer Stiftnummer, ein bestimmtes grafisches Gebiet zuzuordnen (siehe Beispielprogramm unter: „SET_DOT“).

SET_DOT**SET_DOT (X, Y, Mode, Pen)**

SET_DOT: Setzt einzelnes Pixel in Grafik
(nur in Projektmodell 'PM_FULL')

Verändert einzelne Pixel im spezifizierten Grafikgebiet. Diese Funktion bearbeitet den Argumentstring String Str1. Es findet keine Zuweisung statt.

X	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und ist der 1. Punkt der Linie relativ zum „Basispunkt“ (X0,Y0) in kartesischen absoluten Koordinaten an. Wertebereich: 0...(Formatbreite - 1)
Y	sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und ist der 1. Punkt der Linie relativ zum „Basispunkt“ (X0,Y0) in kartesischen absoluten Koordinaten an. Wertebereich: 0...(Formathöhe - 1)
Mode	Art des Dots: 0: schwarz 1: weiß 2: invertiert
Pen	ist eine Variable, Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Stift-Nr.: 0...15. Durch die Stift-Nr. wird eine Referenz zu der entsprechenden Grafik-Area hergestellt, wie sie im letzten DRAW_LINE oder SET_GRAREA definiert worden ist.
Funktionswert	gibt es bei dieser Funktion nicht.

Beachte: SET_DOT arbeitet mit ABSOLUT-Koordinaten, die innerhalb des aktiven Bereichs liegen. Bei Überschreitung wird die entsprechende Koordinate auf den Rand des Grafikbereichs begrenzt.

Beachte: SET_DOT arbeitet vollkommen unabhängig von den Transformationen, die für das Linienzeichen gelten (DRAW_LINE, DRAW_NEXT_LINE, etc.). Es werden also keinerlei Verschiebungen oder Transformationen an den Dot-Koordinaten vorgenommen.

Das Beispiel zeigt das Setzen von Pixeln in einer Quasi-Zufallsverteilung auf dem LCD.

Programmbeispiel:

4

```

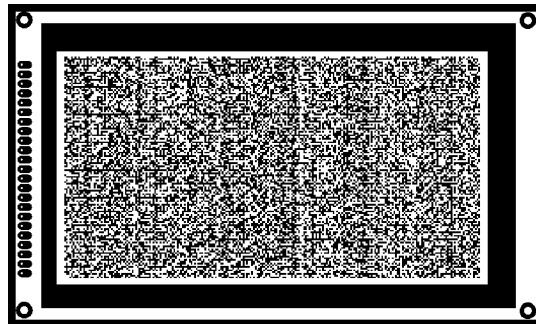
'-----
' Name: SET_DOT.TIG - Vector graphic: Set-Dot demo
'-----
' Note: connect L71 to reset LCD
'-----

USER_VAR_STRICT
#include DEFINE_A.INC      ' general definitions
#include UFUNC3.INC        ' definitions for user function codes
#include LCD_4.INC         ' definitions for LCD Typ 4
#include GR_TK1.INC        ' definitions for Graphic Toolkit

STRING SCREEN$(4K)        ' Global Strings for Pixel-data
LONG N,K

TASK MAIN
  DIR_PORT 8,1000000b      ' L80...L86 output: LCD ctrl
  DIR_PORT 7,11111101b    ' L71 output: LCD reset
  OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
  OUT P_LCD_RESET,M_LCD_RESET,255 ' "
  OUT P_LCD_CTRL,M_INVERT,255    ' CTRL pin of T6963C: no invert
  OUT P_LCD_CTRL,M_FONT_SEL,0    ' CTRL pin of T6963C: font
  OUT P_LCD_CTRL,M_DISP_ON,255   ' CTRL pin of T6963C: display ON
  INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
  SCREEN$ = FILL$("<00>",4K)    ' initiate screen: white dots
  DRAW_LINE (SCREEN$,240,128, 0,0, 0,0, 0,0, 1000, 1000, 0,0,0)
  FOR K=1 TO 9999999000      ' many loops
    FOR N=1 TO 33
      SET_DOT (MOD(K*TICKS()+N,240), MOD(RND(N*K),128),0,0)
    NEXT N
    PUT #12, #1, SCREEN$, 0,0,3840 ' <-- print graphic to LCD
  NEXT N
END

```



DISTANCE

DISTANCE (X1, Y1, X2, Y2)

DISTANCE: Ermittelt den Abstand zweier Koordinaten-Paare
(nur in Projektmodell 'PM_FULL')

Ermittelt die Entfernung zwischen 2 Punkten in einem kartesischen Koordinatensystem.

X1, Y1 sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die X-/Y-Koordinaten des 1. Punktes: Wertebereich: -8191...+8191.

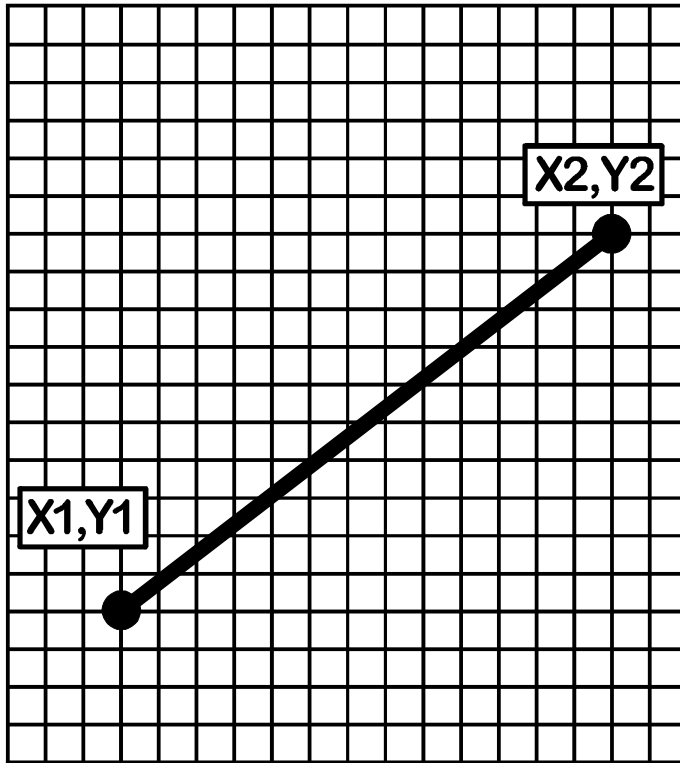
X2, Y2 sind Variable, Konstante oder Ausdrücke vom Typ BYTE, WORD oder LONG und sind die X-/Y-Koordinaten des 2. Punktes: Wertebereich: -8191...+8191.

eine Variable, eine Konstante oder ein Ausdruck vom Typ BYTE, WORD oder LONG und ist die Y-Koordinate des ersten bzw. zweiten Punktes.

Funktionswert ist vom Typ LONG und immer positiv. Eine Typwandlung bei der Zuweisung findet automatisch statt.

Vorteil dieser Funktion ist, daß sie direkt ein LONG Ergebnis liefert und keine Wandlung nach REAL und wieder zurück erforderlich ist.

4



Distance

FILL_AREA

FILL_AREA (Src_Dest)

FILL_AREA: Füllt Gebiete mit beliebiger Kontur

Füllt ein Gebiet, das von Inversions-Dots umgeben ist, mit „1“ und „0“ Bits. FILL_AREA arbeitet mit der DRAW-Funktion in der Vektorgrafik zusammen.

Src_Dest ist eine Variable, eine Konstante oder ein Ausdruck vom Typ STRING und enthält zunächst Inversions-Dots. Nach Ausführung der Funktion liegen hier nur noch die gefüllten Bits vor.

Funktionswert gibt es bei dieser Funktion nicht. Die Daten im String werden direkt verändert.

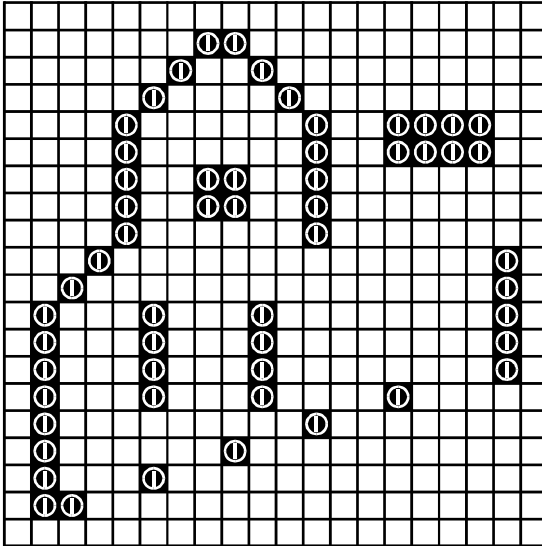
FILL_AREA wird angewendet, um aus gezeichneten Linien (DRAW_LINE), Flächen zu bilden. Dazu wird in der DRAW-Funktion das zu füllende Gebiet mit einer Kurve umschlossen, wobei der Draw-Modus „Inversions-Linie“ gewählt wird.

„Inversions-Dots“ markieren Pixel-Positionen, an denen der Farbwert wechselt. „Inversions-Dots“ treten immer paarweise in jeder waagerechten Pixelzeile auf. Der Inversions-Dot selbst wird durch ein Pixel in der neuen Farbe (0=„weiß“, 1=„schwarz“) ersetzt. Nachfolgende Abbildung demonstriert die Zusammenhänge:

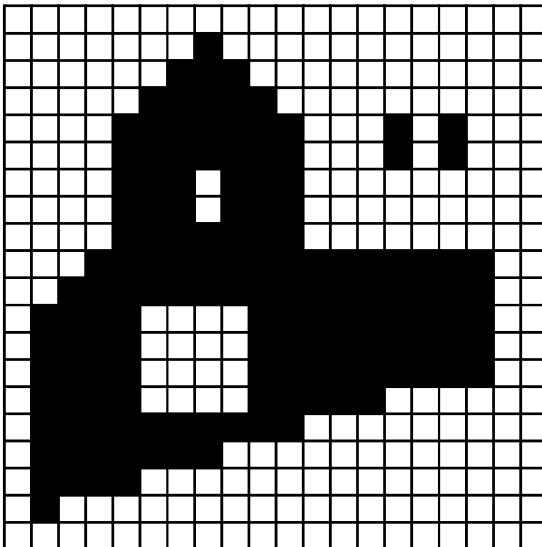
4

Erzeugung gefüllter Flächen mit FILL_AREA

Inversion-Lines



Filled Areas



Programmbeispiel:

```

-----
' Name: FILL_AREA.TIG - Vector graphic: FILL_AREA demo
-----
' Note: connect L71 to reset LCD
-----
USER_VAR_STRICT
#include DEFINE_A.INC ' general definitions
#include UFUNC3.INC ' definitions for user function codes
#include LCD_4.INC ' definitions for LCD Typ 4
#include GR_TK1.INC ' definitions for Graphic Toolkit

STRING SCREEN$(4K) ' Global Strings for Pixel-data

TASK MAIN
DIR_PORT 8,1000000b ' L80...L86 output: LCD ctrl
DIR_PORT 7,1111101b ' L71 output: LCD reset
OUT P_LCD_RESET,M_LCD_RESET,0 ' RESET LCD
OUT P_LCD_RESET,M_LCD_RESET,255 ' "
OUT P_LCD_CTRL,M_INVERT,255 ' CTRL pin of T6963C: no invert
OUT P_LCD_CTRL,M_FONT_SEL,0 ' CTRL pin of T6963C: font
OUT P_LCD_CTRL,M_DISP_ON,255 ' CTRL pin of T6963C: display ON
INSTALL_DEVICE #12,"LCD-6963.TDD",0,0,0EEH,4,150,11H' 240x128,150KB/s
SCREEN$ = FILL$ ("<00>",4K) ' initiate screen to "white" dots

DRAW_LINE (SCREEN$,240,128, 0,0, 10,10, 20,110, 1000, 1000, 0,0,0)
DRAW_LINE (SCREEN$,240,128, 180,0, 40,10, 30,110, 1000, 1000, 0,0,0)
DRAW_LINE (SCREEN$,240,128, 0,0, 70,20, 70,90, 1000, 1000, 0,0,0)
DRAW_LINE (SCREEN$,240,128, 0,0, 71,20, 79,90, 1000, 1000, 0,0,0)

FILL_AREA (SCREEN$) ' Fill area now
PUT #12, #1, SCREEN$, 0,0,3840 ' print graphic on LCD
END

```

4



Im Device-Treiber-Handbuch befinden sich im Abschnitt Applikationen unter der Überschrift LCD-Kit die Beispiele zum BASIC-Tiger® Graphic-Toolkit. Die Beispiele sind eine Weiterführung dieses Kapitels. Das Touch-Panel als Tastatur ist dort ebenfalls beschrieben.

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC[®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

5 Tiger-BASIC® für Fortgeschrittene

Sie finden in diesem Kapitel Informationen über Tiger-BASIC®-Programmiertechniken, die Programme zur Laufzeit schnell, effizient und übersichtlich machen.

Allgemeines zur Software-Entwicklung:

Fehler vermeiden, bevor sie entstehen. Hier einige Tipps in aller Kürze:

- USER_VAR_STRICT wählen (immer)
- Strings immer initialisieren: z.B. A\$=““, B\$=FILL\$(„“, 4K), etc
- Variablen initialisieren
- In Abbruch-Bedingungen möglichst viel Information anwenden:
Test auf Ende von Loops, Routinen etc:
Wenn möglich bevorzugt:
IF A >= B THEN ...
an Stelle:
IF A=B THEN
- Keine „Trick“-Programmierung, Variablen-Benutzung möglichst klar halten:
 - gut verstehbare Variablen-Namen wählen (auch lange Namen vergrößern keineswegs den entstehenden Programm-Code)
 - Variablen möglichst immer nur für einen Zweck einsetzen. Also, nicht „N“ in der gleichen Routine als Laufvariable einer Schleife, später als ein Temperaturwert und dann noch als Flag für eine andere Routine benutzen. Der vermeintliche Vorteil, etwas Variablen Platz einzusparen, kostet viel Debug-Zeit, reduziert die Übersichtlichkeit und führt in größer werdenden Projekte schließlich zu unkalkulierbaren Risiken.

5

Dokumentation

Zahlreiche Untersuchungen zu Fehlerfreiheit bzw. Fehlerhaftigkeit von Software und Produktivität von Software-Entwicklern sind angestellt worden. Immer wieder gleichen sich die Ergebnisse. Die Produktivität eines Software-Entwicklers über den gesamten Verlauf einer Entwicklung hinweg, zeigt desillusionierende 2 bis 5 Codezeilen pro Stunde. Bei großen Projekten sind es weniger als bei kürzeren Projekten.

Ebenso zeigen solche Untersuchungen klar, dass unerkannt gebliebene Fehler um so teurer werden, je früher im Projekt-Verlauf sie entstanden sind.

Bei nüchterner Betrachtung kann man also Termindruck kaum als Ursache für mangelnde Dokumentation verantwortlich machen. Bei einem durchschnittlichen Zeitbedarf von 12 bis 30 Minuten für eine Programmzeile (!), ist wohl ausreichend Zeit

Tiger-BASIC® für Fortgeschrittene

vorhanden, die Absicht hinter einer Programmzeile zu erläutern, die Datenstrukturen klar auszuzeigen oder Variablen-Namen plausibel auszuwählen.

Gute, klar verstehbare Dokumentation ist eine wesentliche Voraussetzung für stabile Projekte von langer Lebensdauer, und guter Pflegebarkeit und späterer Erweiterbarkeit.

Symbolische Werte verwenden

Zahlenwerte und andere Konstanten weitestgehend aus dem Programm-Körper heraushalten und im Kopfteil eines Programms definieren. Dies erleichtert das Verständnis und trägt zur besseren Pflegebarkeit entscheidend bei.

Schlecht:

```
AREA1 = 51
```

Etwas besser:

```
AREA1 = 3 * 17
```

Viel besser:

```
LENGTH1 = 17           \ 17 meters, length of area-1
WIDTH1 = 3              \ 3 meters, width of area-1
AREA1 = LENGTH1 * WIDTH1 \ square meters in unloading area "1"
```

Eine Konstante wie „17“ ist schwerlich in einem Programm wiederzufinden, wenn man, wie im ersten Fall, bereits ein abgeleitetes Ergebnis (Fläche = 51) im Programmcode einsetzt.

Etwas besser ist es, wenn zumindest die Komponenten des neuen Wertes AREA1 als Produkt von 3 und 17 erkennbar werden, wie im zweiten Beispiel.

Schwierig dürfte es außerdem fallen, die Bedeutung von Zahlen im Programm-Körper sofort richtig zu erkennen. Man stelle sich nur vor, die Zahl „3“ taucht 28 mal in einem Programm auf und gibt dabei 6 verschiedene Sachverhalte wieder. Viel Vergnügen bei der Fehlersuche.

Und schlussendlich denke man nur daran, was passiert, wenn man in einem Programm bestimmte Zahlenwerte ändern will. Sind alle Konstanten im Kopfteil definiert und dokumentiert, ist dies ein Leichtes. Eine äußerst fehlerträchtige und zeitaufwendige Prozedur ergibt sich beim Durchforsten zahlreicher Programmdateien

nach dem gesuchten Zahlenwert - in Reinform oder bereits eingearbeitet in anderen Zahlen.

Elementare Werte werden definiert, abgeleitete Werte werden berechnet.

Beispiel: Quadervolumen

```
B= 12      ` Breite in mm
H= 50      ` Hoehe in mm
L= 100     ` Laenge in mm
V = 60000  ` Volumen in mm**3
```

Schlechtes Vorgehen. Einer von den 4 Werten ist überzählig und vermindert die Orthogonalität. Man stelle sich die Wirkung vor, wenn man irgendwo im Programmablauf mit einem anderen Volumen arbeiten will und die Zuordnung: $V = 80000$ verwendet. Das Chaos ist perfekt: $B * H * L < V$.

Also:

```
B = 16      ` neue Breite (mm)
V = B*H*L    ` neues Volumen berechnen (mm**3)
```

5

Gegentests machen, Fehler, Schwächen oder Grenzen aufdecken:

Wenn ein Projekt nahezu abgeschlossen ist und bereits „ganz gut“ läuft, sollte man einige „Stress-Situationen“ herbeiführen, um das Projektes auf Stabilität, Fehlertoleranz und versteckte Bugs zu untersuchen.

Solche Tests können verdeckte Schwächen oder selten auftretender Fehler ans Tageslicht bringen, noch bevor das Projekt erstmalig installiert wird oder in Serie geht.

Selbstverständlich ist bei solchen Tests immer mit der erforderlichen Sorgfalt auf alle anwendbaren Sicherheitsvorschriften zu achten. Insbesondere, wenn in den Projekten mit elektrischen Spannungen, hohen Energien, Kräften, gefährlichen Substanzen und dergleichen gearbeitet wird. Die Testumgebung muss jeglichen Schaden an Mensch, Material und Umwelt zuverlässig ausschließen.

Wie verhält sich das Projekt in diesen Fällen?

- Alle Variablen mit schwierigen / unplausiblen Werten vorbesetzen
- Unplausible Eingangs-Zustände oder Eingangs-Daten herbeiführen
- Unplausible, widersprüchliche Bedienungen testen.
- Gezielte Worst-Case Situationen herbeiführen.
- Projekt an seinen Leistungsgrenzen testen.
- „Real World“ Tests, Installation unter realistischen Bedingungen.
- „Idioten Tests“ - Produkt von Leuten benutzen lassen, die über keine oder wenig Vorinformationen verfügen.

Systemdefinitionen

Es gibt einige Standard-Include-Dateien, die nützliche Definitionen enthalten. Die Dateien: `DEFINE_A.INC`, `DEFINE_I.INC`, `UFUNCn.INC` und für das Plug & Play Lab `KEYB_PP.INC` sollten in keiner größeren Anwendungen fehlen. Benutzen Sie die Standard-Include-Dateien **stets unverändert** in der gelieferten Form und schreiben Sie eigene Definitionen, Routinen oder Code-Stücke unbedingt in **eigene** Include-Dateien. So ist sicher gestellt, daß Ihre eigenen Codes niemals durch ein Update überschrieben werden.

DEFINE_A.INC	enthält Definitionen, die außerhalb einer Task stehen.
DEFINE_I.INC	enthält Zuweisungen, die innerhalb einer Task stehen müssen.
UFUNC.INC	enthält Definitionen zu Device-Treibern.
KEYB_PP.INC	enthält sowohl Definitionen als auch Zuweisungen und das Unterprogramm <code>INIT_KEYB</code> (Gerätenr.) zur Initialisierung des LCD1-Treibers passend zur Tastatur des Plug & Play Labs. <code>KEYB_PP.INC</code> muß außerhalb einer Task eingefügt werden.

Variablendefinition

Tiger-BASIC® erlaubt es, Variablen explizit oder automatisch zu deklarieren. Ist es für kleinere Programme nützlich Variablennamen direkt zu benutzen ohne sie zuvor deklarieren zu müssen, so empfiehlt es sich bei Programmen, die hunderte oder tausende von Codezeilen umfassen, grundsätzlich alle Variablen explizit zu definieren. Zur Unterstützung des Programmierers stellt Tiger-BASIC® dazu die Compilerdirektive **USER_VAR_STRICT** bereit. Sie bewirkt, daß der Compiler nicht definierte Variablen oder Konstanten als Fehler meldet.

Vorteil der expliziten Variablendeklaration ist die Klarheit über den Variablen-Typ und den Geltungsbereich. Ferner kann man den verwendeten Speicherplatz besser kontrollieren. Strings werden explizit in der Länge deklariert, die maximal benötigt wird, Byte und Word-Variablen werden als solche definiert.

Die Typ-Deklaration allein weist Variablen noch keinen Wert zu. Dies ist von Bedeutung in Systemen mit Batterie-Backup für den RAM, wo Variablen noch die gleichen Werte enthalten (nur globale und Task-lokale Variable), die sie vor dem Power-Down des Systems besaßen. Lokale Variable in Unterprogrammen können niemals durch Batterie-Backup erhalten bleiben! Gibt es keinen geschützten RAM, sind die Variablen entsprechend zu initialisieren.

Namen in Tiger-BASIC®-Programmen können bis zu 32 Zeichen lang werden und neben Buchstaben auch Ziffern und Underline-Zeichen enthalten. Groß- und Kleinschreibung im Namen hat keine unterscheidende Wirkung, so daß die Schreibweise so gewählt werden kann, wie sie dem Programmierer jeweils am übersichtlichsten erscheint. Bei der Namensvergabe von Konstanten und Variablen haben sich u.a. folgende Methoden bewährt - eine Ideenliste:

- Je größer der Geltungsbereich einer Variablen um so ausgeprägter der Name: N, PHI, WINKEL, Anstell_Winkel4, ...
- Namen von globalen Variablen werden nicht als lokale Variablennamen benutzt
- Der Variablen-Typ wird durch den 1. Buchstaben des Namens gekennzeichnet, z.B:
 bEmpfang = BYTE
 wCount = WORD
 lAbsolut_X = LONG
 rWinkel_Geschwindigkeit4 = REAL
- Lange Namen werden optisch aufbereitet: „HauptSchleusenEingang“ oder „Haupt_Schleusen_Eingang“ statt HAUPTSCHLEUSENEINGANG

5

Variablen-Typen richtig benutzen

Eine Gruppe gängiger Variablen Typen in Tiger BASIC sind die ganzzahligen Variablen: BYTE, WORD und LONG. Bei Zuweisungen des einen Typs zu einer Variablen eines anderen Typs sorgt Tiger-BASIC® automatisch für eine entsprechende Typwandlung.

Man beachte hierbei:

BYTE und WORD sind immer vorzeichenlos!

LONG-Variablen können ebenso positive wie negative Werte einnehmen.

Daher ergeben sich diese Wertebereiche:

Variable	Wertebereich hexadezimal	Wertebereich dezimal
BYTE	00...FFH	0...255
WORD	0000...FFFFH	0...65535
LONG	8000 0000H ...0 ...7FFF FFFFH	-2.147.483.648 ...0 ...2.147.483.647

Tiger-BASIC® für Fortgeschrittene

Bei Abfragen wie:

IF A >= 0 THEN ...

stellen Sie daher sicher, dass „A“ vom Type LONG ist. Als BYTE oder WORD wäre diese IF-Abfrage ziemlich sinnlos, denn sie wäre ohnehin immer wahr.

Eine ganze, negative Zahl wird immer als LONG generiert, also z.B:

-33 = FFFF FFDDH

Wird diese Konstante nun einer Byte Variablen „B“ zugewiesen:

B = -33

so enthält B anschliessend den Wert: DDH = 201 (dez). Die oberen 24 Bit werden einfach abgeschnitten. Entsprechend bei WORDs.

Die Vorzeichenhaftigkeit und Vorzeichenlosigkeit ist immer im Auge zu behalten. Ein Beispiel:

```
BYTE B1, B2
LONG L1, L2
.
L1 = -6
B1 = 30
B2 = L1
IF B1 > B2 THEN ...
```

\ Zuweisung LONG mit Vorzeichen
\ nach BYTE ohne Vorzeichen
\ hier unerwartetes Ergebnis

In diesem Fall wurde B2 der Wert von L1 (= -6) zugewiesen. Dem Wert -6 entspricht in HEX-Darstellung = FFFF FFFA. Bei der Zuweisung dieses LONG-Wertes zur Byte-Variablen B2 werden die oberen 3 Bytes abgeschnitten, es entsteht der Wert B2 = FA (hex) = 250 (dez). Der Vergleich: IF B1 > B2 THEN bringt daher das möglicherweise unerwartete Ergebnis, daß B1 **nicht** größer als B2 ist.

Genauso verhält es sich bei Rechenoperationen:

```
BYTE B
LONG L
B=0      \ B = 00H
B=B-1    \ B wird nun der Wert FFH = 255 zugewiesen
L=7FFF FFFFH \ L <- größte positive Zahl
PRINT #2,L   \ drucke L als Dezimal: 2147483647
PRINT #2,L+1 \ L+1 = -2147483648 = kleinste Zahl
```

Die Bereichsenden von Ganzzahl-Typen können durch Rechenoperationen mühelos überschritten werden, ohne dass dies zu einem formalen Fehler führt. Hingegen muss der Programmierer selbst kontrollieren, ob er solche Ergebnisse erzielen will.

Beim Rechnen mit ganzen Zahlen greift man meist auf LONG Typen zurück, die den größten Wertebereich aufweisen und vorzeichenbehaftet sind. Rechenoperationen mit LONG werden um ein Vielfaches schneller ausgeführt als REAL Berechnungen, und ganze Zahlen eignen sich auch für Vergleiche wie:

IF A=B ..., bzw. IF X <> 87654 THEN ...

oder als Laufvariablen oder Abbruchbedingungen in Schleifen.

Bei REAL ist zu beachten, dass man Vergleiche der Art:

IF R1 = 12.345678 THEN ...

niemals verwenden sollte, da sie höchstwahrscheinlich niemals WAHR bzw. bei „<>“ niemals FALSE werden.

Tatsächlich können LONG Variablen auch sehr nützlich sein für schnell ablaufende Festkomma-Berechnungen. Hierzu wird von einer Re-Skalierung von Zahlenwerten Gebrauch gemacht, z.B. in der Art, dass man statt in °C in „m°C“ = Milli-Grad-Celsius rechnet. Die Zahl 123456 bedeutet dann: 123,456 °C.

Zu beachten ist die Einhaltung des LONG-Wertebereiches auch bei allen Zwischenergebnissen. Werte-Überlauf führt zu unzutreffenden Resultaten:

Noch OK:

46 340 * 46 341 = 7FFF 5D14H = 2 147 441 940

Überlauf:

46 340 * 46 342 = 8000 1218H = -2 147 479 016 statt: 2 147 488 280

Beispiel: will man etwa einen LONG-Wert „N“, dessen Wertebereich mit – 100 000...+100 000 definiert ist, mit einer Festkommazahl „1,234“ multiplizieren, so geht man in zwei Schritten vor:

- Multiplikation mit 1234 - LONG Zahlenbereich wird **nicht** überschritten.
- Division durch 1000 - ebenfalls OK

Also: $RESULT = N * 1234 / 1000$ ‘zuerst „*“, danach „/“

LONG Variablen besitzen zwar die Vorzeichenhaftigkeit, jedoch kann man LONG-Variablen immer auch als eine Gruppe von 32 Bits ansehen. So können alle 32 Bits, also auch das Vorzeichen-Bit, mit logischen Operatoren (BITAND, BITOR, BITXOR, ..), Schiebe-Operatoren (SHL, SHR, ...), und Funktionen (BITNOT), Spiegelungen (BIT_MIRR), etc. manipuliert werden. Der Begriff des Vorzeichens kommt erst ins Spiel wenn wir eine LONG-Variable als Zahlenwert, insbesondere als Dezi-

Tiger-BASIC® für Fortgeschrittene

mal-Zahl sehen (siehe: PRINT und PRINT_USING). Als Bit-Muster existiert kein Vorzeichen.

Zum Beispiel:

```
L1=7FFF FFFFh           \ die groesste positive Zahl
PRINT #LCD, L1
PRINT #LCD, L1+1         \ Ueberlauf nach -0
PRINT #LCD, L1+2         \ die groesste negative Zahl

USING "UH<8><8> kV8.8.8.8" \ das gleiche in HEX ausgegeben
PRINT USING #LCD, L1
PRINT USING #LCD, L1+1
PRINT USING #LCD, L1+2

L1=8000H
PRINT #LCD, L1 SHR 1
PRINT #LCD, L1 SHR 5
PRINT #LCD, L1 SHR 8
PRINT #LCD, L1 SHR 17    \ Ergebnis wird 0, da das Bit
                          \ verschwindet
PRINT #LCD, L1 SHL 16     \ Ergebnis wird -0
```

5

Format-Umwandlungen

Es ist ganz nützlich, eine gute Vorstellung von den Datenformaten in Tiger-BASIC® und deren Umwandlungen untereinander zu besitzen. Sei es um kürzere und schneller ablaufende Programme zu erzeugen oder um Programmierfehler von vornherein zu vermeiden.

Bei der Konversion von Datentypen, betrachten wir zwei Bereiche:

- automatische Konversionen, die vom Compiler vorgenommen werden
- explizite Konversionen, die der Programmierer gezielt einsetzt.

Automatisch konvertiert werden numerische Variablen.

Die Variablen Typen BYTE, WORD und LONG bilden eine Gruppe, deren Mitglieder ohnehin fast immer nach Belieben verwendet werden können. Der Compiler wird vor jeder Rechenoperation alle BYTES oder WORDs zunächst mal auf LONG Format bringen, dann die Berechnungen ausführen und abschließend wieder den Ergebnis-Typ herbeiführen. Der Programmierer selbst kontrolliert, ob es zu gewollten oder ungewollten Zahlenwert-Überläufen dabei kommt.

Ebenso sei man sich im Klaren, dass bei I/O-Instruktionen die dem Typ entsprechende Zahl von Bytes übertragen wird:

```
BYTE1 = 77H
WORD1 = 66H
```



```
LONG7 = 55H  
PUT #23, BYTE1,WORD1, LONG7
```

überträgt diese Daten-Bytes an den I/O-Kanal #23:

„77 66 00 55 00 00 00“

Bei gemischter Verwendung von REAL und Ganzzahl Typen (BYTE, WORD, LONG), wandelt der Compiler zunächst alle Ganzzahl-Typen in REAL um, berechnet den Ausdruck und erzeugt im Anschluss den gewünschten Ergebnis-Typ, z.B:

PRINT #2, 4 * SIN (4)

druckt ein REAL Ergebnis aus: -3,027209981231716 E+000

Entsprechend:

- 1.234 * 8^c REAL
- 5/4 ^c LONG
- 5./4 ^c REAL

5

STRING, ARRAY oder FIFO werden nicht automatisch konvertiert.

Programmieren Sie vorzugsweise so, daß der Compiler keine oder nur wenige automatische Typwandlungen von Integer zu Real und umgekehrt durchführen muß. Typwandlungen kosten sowohl Rechenzeit als auch Speicherplatz, und reduzieren ev. die Genauigkeit der Berechnung. Vermeiden Sie unnötige Typwandlungen. Beispiel:

N=123.45*SIN(X/10)

Wenn N und X Integer-Variablen sind, so werden hier 2 Typwandlungen ausgeführt.

Bei expliziten Typwandlungen verfolgt man Absichten, die sich aus den Anforderungen des Projektes ergeben. Dies kann z.B. die Analyse oder Synthese eines Datenstroms sein, die Umsetzung in eine andere Darstellungsform oder einfach die Verbesserung des Programmablaufes.

Die Funktionen für die Typ-Umsetzungen transferieren numerische Werte: BYTE, WORD, LONG, REAL in STRINGS bzw. holen sie aus STRINGS.

Da man jeweils freien Zugriff auf jede beliebige String-Position hat, können Strings so zu beliebigen Strukturen aufgebaut werden, die allerlei verschiedene Datentypen enthalten: Text, Graphik, numerische Werte, ... etc.. Solche Strukturen werden z.B. in Datenbanken und DFÜ Anwendungen immer wieder verwendet.

Geschwindigkeit

Im Folgenden sollen einige Betrachtung über die Ablaufgeschwindigkeit von Tiger-BASIC® Programmen angestellt werden. Bei Anwendungen, die einen hohen Durchsatz an Daten haben, in denen umfangreiche Berechnungen angestellt werden oder wenn kurze Antwortzeiten erzielt werden sollen, lohnt es sich bei der Programmierung auf Faktoren zur Erhöhung der Ablaufgeschwindigkeit zu achten.

Geschwindigkeit messen

Die Ablaufgeschwindigkeit von Tiger-BASIC-Instruktionen und Funktionen kann sehr leicht gemessen werden. Dazu gibt es im Verzeichnis 'APPLICATIONS' das Beispiel-Programm 'SPEED1.TIG', um Instruktionen auf ihr Laufzeitverhalten untersucht werden. Man kann diese Programme auch dazu verwenden, eigene Programm-Konstruktionen unter dem Geschwindigkeits-Aspekt unter die Lupe zu nehmen und mit alternativen Programmiertechniken zu vergleichen. Hier das Programm-Beispiel SPEED1.TIG:

5

PC-Mode

BASIC-Instruktionen laufen im PC-Mode stets langsamer ab als im RUN-Mode. Ferner ist das Zeitverhalten eines Programms, besonders von Multitasking Anwendungen, im PC-Mode vollkommen anders als im RUN-Mode. Manche Instruktionen verlangsamen sich stark - alle schnell ablaufenden Instruktionen - andere wenig oder gar nicht. Dazu zählen Instruktionen, wie WAIT DURATION, oder gepufferte I/Os, wenn ihr Zeitverhalten hauptsächlich vom aktuellen Datenverkehr abhängt.

Dies kann zu überraschenden Effekten führen, z.B., daß eine Task im PC-Mode schneller abläuft als im RUN-Mode und sich die relative Leistung zwischen den Tasks im PC-Mode ganz anders verteilt als im RUN-Mode. Soll das Zeitverhalten einer Anwendung studiert werden, so sollte man dies stets im RUN-Mode durchführen.

Numerik

Tiger-BASIC® verfügt über eine schnelle 32 Bit Integer Arithmetik sowie die hochgenaue double precision REAL-Arithmetik. Wenn Sie auf Geschwindigkeit Wert legen, planen Sie Ihre Anwendung so, daß sie in den betreffenden Programm-Phasen möglichst nur Integer-Arithmetik (BYTE, WORD, LONG) einsetzen und REAL Arithmetik nur dann benutzen, wenn es nicht auf maximale Geschwindigkeit ankommt.

Die 32-Bit-Integer-Arithmetik verfügt über eine für viele Anwendungen bei weitem ausreichende Genauigkeit. Bei entsprechender Abbildung kann mit dem verfügbaren Wertebereich von: -2.147.483.648 ... + 2.147.483.647 z.B. eine Skala von -20.000 ... +20.000 aufgebaut werden mit einer Auflösung von 0,000 01 (= 1/100.000)! Dies können Meßwerte wie Temperatur, Druck, Weg, Geschwindigkeit, und vieles mehr

sein, wie sie in Forschungs-Projekten und Steuerungs-Aufgaben immer wieder vorkommen.

Wenn Sie zur Laufzeit dennoch Funktionswerte von REAL-Funktionen benötigen besteht unter Umständen die Möglichkeit, eine hinreichend genaue Integer-Wertetabelle (ARRAY oder STRING) gleich zu Programm-Beginn aufzubauen und dann mit diesen Integer-Werten zu rechnen. Wenn solche Tabellen mit einem Faktor skaliert werden, der als 2-er Potenz darstellbar ist (2, 4, 8, 16, ... 256, 512, 1024 ...), so können die etwas langsameren Divisionen durch die schnellere Integer-Operation SHR (Siehe Operatoren, Seite 34ff) ersetzt werden. Beispiel: Sinus-Tabelle mit Skalier-Faktor 16384.

Stets laufen Berechnungen schneller ab, wenn BASIC-Instruktionen möglichst viele Operationen in einer Zeile enthalten. 5 Additionen in 1 Zeile laufen ca. 3 bis 4 mal so schnell ab wie 5 Zeilen mit jeweils 1 Addition.

Zwischenergebnisse

Mehrmals verwendete Zwischenergebnisse werden als Variable geführt und brauchen nicht immer wieder erneut berechnet zu werden. Die mögliche Zahl der Variablen ist praktisch unbegrenzt, wenn man bedenkt, daß ein LONG Wert nur 4 Byte und ein REAL-Wert nur 8 Byte belegt. Schon das kleinste Tiger-Modul mit 128 Kbyte RAM kann viele tausend solcher Variablen aufnehmen.

Wie schon oben angemerkt, sollten (unnötige) Typwandlungen vermieden werden. Sie reduzieren die Genauigkeit, verbrauchen Rechenzeit und vergrößern den Programm-Code. Dies trifft besonders auf die automatisch durchgeführten Typwandlungen zwischen REAL und LONG zu. Der Wechsel zwischen den 3 Integer-Typen BYTE, WORD und LONG ist hiervon nicht betroffen. Es ist OK, diese in einem Ausdruck zu mischen, ohne daß dies zu erkennbarem Geschwindigkeitsverlust führen würde.

Strings

Für das schnelle und effektive String-Handling gibt es eine Reihe neuer String-Funktionen:

Funktion	
NTOSS	baut die Bytes einer numerischen Variablen vom Typ BYTE, WORD oder LONG in einen String ein.
NFROMS	baut eine numerischen Variable vom Typ BYTE, WORD oder LONG aus Bytes, die sich in einem String befinden.
RTOS\$	baut die Bytes einer numerischen Variablen vom Typ REAL in einen String ein.
RFROMS	baut eine numerischen Variable vom Typ REAL aus Bytes, die sich in einem String befinden.

Tiger-BASIC® für Fortgeschrittene

STOS\$

baut die Bytes eines Strings in einen anderen String ein.

Immer wiederkehrende Aufgabenstellungen wurden als Funktionen realisiert, die sehr schnell ablaufen und jeweils nur 1 oder 2 Zeilen BASIC-Code erfordern. Weiterhin ist bei der Handhabung von Strings bedeutend, daß das Kopieren von Strings, und besonders wenn die Stringlänge zunimmt, viel Zeit erfordern.

Statt:

```
A$ = LEFT$ (A$,40) + „XYZ“ + RIGHT$ (A$,(LEN (A$)-3-40))
```

Viel besser:

```
A$ = STOS$ (A$, 40, „XYZ“, 3)
```

STOS\$ (String-To-String) kopiert einen Teil eines Strings direkt in eine vorgegebene Stelle eines anderen Strings hinein. Das geht um ein Vielfaches schneller als die erste Variante, bei der mehrere langdauerende Kopiervorgänge ausgelöst werden. Weitere Funktion in dieser Art sind z.B. NTOS\$ (Num-To-String), RTO\$ (Real-To-String), NFROM\$ (Num-From-String), RFROM\$ (Real-From-String). Noch ein Beispiel dazu:

Statt:

```
N = ASC(MID$(A$,POS,1)) + (256*ASC(MID$(A$,POS+1,1)))
```

Viel besser:

```
N = NFROM$ ( A$, POS, 2)
```

Gewinnbringend lassen sich auch die String-Funktionen SELECT\$ und INDEX einsetzen, die den Programm-Code sehr übersichtlich halten und schnell ablaufen. SELECT\$ und INDEX werden u.a. dazu benutzt, um Menü-Auswahlen zu programmieren, Daten-Kompression, Kommando-Decoder und komplexe Code-Konverter aufzubauen, schnelle Programm-Verzweigungen zu realisieren etc.

Beispiel:

```
SEL=2
AUSWAHL$ = &
„:Klappe links:Klappe oben :Luefter-1      :Luefter-2      :Not-Programm: :“
PRINT #LCD, Z2$; SELECT$ ( AUSWAHL$, SEL)
```

AUSWAHL\$ enthält eine Reihe von Einträgen (Records), die durch ein Trennzeichen (hier: „:“) getrennt sind. Die Funktion SELECT\$ wählt nun über den Index SEL den gewünschten Eintrag aus, in diesem Fall also: „Lüfter-1“. Dies kann man dazu benutzen, um einer Bedienperson auf einem Display durchrollende Auswahl-Felder mit Selektionen anzubieten, oder es können häufig verwendete Ausdrücke oder Wen-

dungen durch einen Index abgekürzt werden, was zu enormer Platz-Einsparung führen kann.

Die inverse Funktion hierzu ist INDEX. Man hat einen Haupt-String und einen String mit einem Schlüsselwort oder einer Bytefolge, und man sucht den Index des Schlüssels im Haupt-String.

Beispiel:

```
COMMAND$ = „RIGHT“  
COMSET$ = „-START-STOP-HALT-LEFT-RIGHT-UP-DOWN-RETURN-EXIT--“  
N = INDEX (COMSET$, COMMAND$)
```

Hier liefert INDEX den Wert 4 an die Variable N. Das ist der Index des Begriffs „RIGHT“ im String COMSET\$. Der Aufbau eines Kommando-Interpreters wird so mit wenigen Programmzeilen ermöglicht. Auch sinnvoll:

Beispiel:

```
COMMAND$ = „Return“  
COMSET$ = „-START-STOP-HALT-LEFT-RIGHT-UP-DOWN-RETURN-EXIT--“  
N = INDEX (COMSET$, UPPER$ (COMMAND$) )
```

Hier sorgt die zusätzlich eingeführte Funktion UPPER\$ dafür, daß auch gemischte Groß-/Kleinschreibung richtig als Kommando erkannt wird. Gegenüber der Programmierung in einer Programmschleife wird ein Vielfaches der Geschwindigkeit erreicht.

Programmschleifen

Programm-Schleifen und insbesondere ineinander verschachtelte Schleifen können schnell zu wahren Zeitfressern werden. Man denke nur an 3 ineinander geschachtelte Schleifen à 30 Durchläufe --> $30 * 30 * 30 = 27000$ Loops! Wenn es sinnvollen Ersatz gibt, der weniger Durchläufe erfordert kann dies große Vorteile bringen.

Schleife	Vorteil
LOOP...ENDLOOP	Geschwindigkeit sehr hoch, Anzahl Durchläufe festgelegt, vorzeitiger Aussprung möglich.
FOR...NEXT	Geschwindigkeit hoch, Anzahl Durchläufe festgelegt, Laufvariable kann in der Schleife manipuliert werden
WHILE...ENDWHILE	eine Bedingung bestimmt die Anzahl der Durchläufe

Tiger-BASIC® für Fortgeschrittene

In Multitasking Umgebungen sollte man sich stets bewußt machen, daß andere Tasks gerne freie CPU Zeit benutzen, wenn es in einer Task nichts mehr zu tun gibt. Das betrifft z.B. immer wieder Abfrageschleifen. Egal, ob man auf eine Tasten-Eingabe, einen Endschalter, eine serielle Übertragung o.dergleichen wartet: hat man einmal in der betreffenden Task getestet und ist die gewünschte Bedingung noch nicht eingetroffen, so sollte man immer die restliche CPU-Zeit dieser Task an die anderen Tasks abgeben. Dies geschieht mit der Instruktion `RELEASE_TASK`. Die gleiche Wirkung mit zusätzlich einer Totzeit wird durch `WAIT_DURATION <ms>` erreicht. Hier wird die CPU-Zeit ebenfalls sofort freigegeben und diese Task erhält für `<ms>` Zeitdauer keine CPU-Zeit mehr.

1. Beispiel - warten auf Tasten-Eingabe:

```
A$=""
WHILE A$=""
  GET #1, 1, A$           ' auf Tastatureingabe warten
ENDWHILE
' Eingabe verarbeiten
```

5

In diesem Beispiel wird ein bedeutender Teil der gesamten CPU-Leistung in einer sinnlosen WHILE-Schleife verbraucht. Wenn es genau jetzt keine Tasten-Eingabe vom Benutzer gegeben hat, ist es nicht so wichtig, dies bereits 1/10.000 Sek später erneut zu prüfen. Man könnte sinnvollerweise 10, 20, 30 ms oder mehr damit warten. Siehe nächstes Beispiel:

```
A$=""
FOR N = 0 TO 0 STEP 0
  GET #1, 1, A$           ' auf Tastatureingabe warten
  WAIT_DURATION 50        ' gebe CPU-Zeit frei
NEXT
' Eingabe verarbeiten
```

IN, OUT

Um digitale Pins des BASIC-Tigers zu setzen oder zu lesen, gibt es die Instruktionen `IN` und `OUT`. Diese Instruktionen prüfen stets, ob die gewünschte Operation nicht eventuell einen Device-Treiber bei seiner Arbeit stören würde. Dies hat den Vorteil, daß man nicht versehentlich eine Reaktion in einem über Device-Treiber angeschlossenen Gerät auslöst, wenn man einen bereits belegten I/O-Pin anspricht. Unter Geschwindigkeits-Gesichtspunkten kostet dies natürlich Laufzeit. Wenn Sie also bereits bei der Projektierung richtig planen, welche Pins was tun, können hier auch die schnelleren Low-Level Instruktionen wie z.B. `LL_IPORT_IN` und `LL_IPORT_OUT` angewendet werden, die entsprechend schneller ablaufen.

Pulse

Ebenso gibt es eine einfache und schnelle Möglichkeit, mit der Instruktion `LL_IPOINT_PULSE` eine bestimmte Anzahl von Pulsen auf einen Pin zu setzen, an Stelle einer entsprechenden Schleife mit `OUT`-Instruktionen. `LL_IPOINT_PULSE` kann z.B. verwendet werden um eine 1 oder 2-stellige Ziffern-Anzeige mit einem Wert zu setzen oder eine andere Schaltung mit Counter und Funktions-Selektion in Gang zu setzen.

Funktionen

Die ständig wachsende Bibliothek von Funktionen bildet ein mächtiges Tool, um Programmieraufgaben effektiv und mit wenigen Instruktionen zu realisieren. Tiger-BASIC Funktionen decken die Bereiche

- Integer Arithmetik
- Floating-Point Arithmetik
- String-Operationen
- Spezial-Funktionen

ab. Immer wiederkehrende Programmier-Aufgaben können auf fertige Funktionen zurückgreifen, die hohe Verarbeitungsgeschwindigkeit und kompakten Code erreichen.

Multi-Tasking

Mehrere Tasks greifen auf den gleichen Device-Treiber zu. Dies sollte mit Vorsicht geschehen, so daß die Plausibilität der Ergebnisse nicht gestört wird. In folgendem Beispiel geben zwei Tasks 'Count1' und 'Count2' ihren Zählstand auf das LC-Display aus. Die Zahlen werden gemischt ausgegeben. Mit einem Tastendruck schalten Sie das Flag 'F' um, wodurch nun eine Task 'Drucke' den Zählstand geordnet ausgibt, während die Zähler weiterlaufen:

Programmbeispiel:

```

'-----
' Name: LCD-OUT.TIG
'-----
LONG C1,C2,N,F                                ' LONG-Variablen deklarieren

TASK MAIN                                     ' Beginn Task MAIN
  STRING A$                                  ' STRING-Variable deklarieren
' LCD-Treiber installieren (BASIC-Tiger)
INSTALL DEVICE #1, "LCD1.TDD"
' LCD-Treiber installieren (TINY-Tiger)
INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
RUN_TASK Count1,50                            ' Starte Count1, Prio. 100
RUN_TASK Count2,10                            ' Starte Count2, Prio. 50
F = 0                                          ' Flagge vorbesetzen
LOOP 9999999                                  ' Sehr viele Loops
  FOR N = 0 TO 0 STEP 0                      ' Endlosschleife bis N=1(GET!)
    RELEASE_TASK                             ' Rest der Task-Zeit freigeben
    GET #1, #0, #1, 1, N                    ' N=Zeichen im Tastatur-Buffer
  NEXT                                       ' Ende Endlosschleife
  GET #1, 1, A$                             ' 1 Zeichen a. Tastatur-Buffer
  F = (F BITXOR 1)                          ' Flagge toggeln
  IF F = 1 THEN                             ' Falls Flagge=1:
    RUN_TASK CPrint                         ' Starte Task CPrint
  ELSE                                       ' Sonst:
    EXIT_TASK CPrint                       ' Beende Task CPrint
  ENDIF
ENDLOOP
END                                           ' Ende Task MAIN

'-----
' Task Count1: Zaehlt C1 hoch und gibt akt. Wert aus
'-----
TASK Count1
  FOR C1=0 TO 9999999
    IF F=0 THEN
      PRINT #1, C1
    ENDIF
  NEXT
END                                           ' naechster Wert fuer C1
                                           ' Ende Task Count1

'-----
' Task Count2: Zaehlt C2 hoch und gibt akt. Wert aus
'-----
TASK Count2
  FOR C2=0 TO 9999999
    IF F=0 THEN
      PRINT #1, C2
    ENDIF
  NEXT
END                                           ' Ende Task Count2
  
```



```

      ENDIF
      NEXT
      END
      ' -----
      ' Task CPrint: Ausgabe der Zaehler C1 & C2
      ' -----
      TASK CPrint
      LOOP 9999999
      PRINT #1, "<1>"; C1
      PRINT #1, C2
      WAIT DURATION 100
      ENDLOOP
      END
      ' Beginn Task CPrint
      ' sehr viele Loops
      ' Ausgabe von C1
      ' Ausgabe von C2
      ' 100 ms warten
      ' Ende Task CPrint

```

Ferner können bei Devices mit gepufferter Ein- und Ausgabe noch unregelmäßige Interferenz-Effekte auftreten, deren Ursache in einem größeren Programm dann schwer zu finden ist. Das Prinzip ist folgendes: wenn der Ausgabe-Kanal eines Device-Treibers gepuffert arbeitet, und 2 Tasks wollen Daten auf den gleichen Kanal ausgeben, z.B. beide Tasks wollen ihre Daten-Pakete in ein Netzwerk einspeisen, dann kann es passieren, daß Task X es immer wieder schafft, ein Paket zu senden, Task Y, jedoch immer wieder einen vollen Sendepuffer vorfindet und deswegen seine Pakete nicht absetzen kann. Während Task X einwandfrei zu arbeiten scheint, glaubt man, Task Y liefere nicht richtig. Unter Umständen tritt dieser Effekt nur gelegentlich auf oder wechselt, so daß manchmal Task X nicht senden kann. Man beobachtet dann eine unerklärliche Diskontinuität, mit der die Tasks ihre Daten-Pakete absenden.

Abhilfe: entweder vermeidet man es, mit 2 Tasks den gleichen Device-Treiber zu verwenden, oder man installiert einen entsprechenden Manager (Unterprogramm oder Task), der die Anforderungen von verschiedenen Tasks und die Gegebenheiten des Devices sinnvoll koordiniert.

WAIT Instruktionen verändern ebenso wie I/O-Instruktionen mit gepufferten Device-Treibern, das Zeitverhalten des Programm-Ablaufs. Eine WAIT Instruktion gibt, wenn sie wartet, sofort CPU Zeit frei für die anderen Tasks. Das selbe geschieht, wenn ein Device-Treiber-Puffer leer/voll ist (Input / Print). Dies führt dazu, daß sich die relative Leistung aller Tasks verschiebt. Tasks, die WAIT Instruktionen oder I/O-Instruktionen auf gepufferte Kanäle enthalten, geben immer CPU-Zeit ab. Sie neigen also dazu auf lange Sicht weniger CPU-Leistung zu beanspruchen, als ihnen von ihrer Priorität her zusteht.

Arbeiten mehrere Tasks gemeinsam mit den gleichen globalen Variablen, so ist darauf zu achten, daß es niemals zu unplausiblen Zuständen kommen kann. Solche Zustände können eintreten auf Grund der Tatsache, daß mehrere Tasks (quasi) gleichzeitig ausgeführt werden.

Beispiel: eine Task will eine Manipulation in einem Array durchführen und prüft den entsprechenden Semaphore, ob das Array gerade

für den Zugriff frei oder gesperrt ist. Wenn der Semaphore „frei“ signalisiert, setzt diese Task den Semaphore selbst auf „gesperrt“ und beginnt mit seiner Arbeit. Die kritische Phase in diesem Fall liegt zwischen den 2 Zeitpunkten ‘Semaphore prüfen’ und gleich darauf ‘Semaphore auf „gesperrt“ setzen’. Da wir uns in einer Multitasking Umgebung befinden, kann es jederzeit vorkommen, daß sofort nach dem Prüfen des Semaphore ein Task-Switching vollzogen wird und eine andere Task den gleichen Ablauf unternimmt: Semaphore prüfen, Semaphore „gesperrt“ setzen. Wenn etwas später die erste Task mit der Instruktion weiter ausgeführt wird, die den Semaphore „gesperrt“ setzt, so war der Semaphore bereits so gesetzt und eine zweite Task glaubt auch, dieses Array manipulieren zu dürfen.

Der Konflikt wird dadurch vermieden, daß in der kritischen Phase zwischen prüfen und setzen des Semaphore, der Task-Switch-Mechanismus gestoppt wird (DISABLE_TSW), und gleich nach der kritischen Situation wieder frei gegeben wird (ENABLE_TSW).

5

Initialisierung von Variablen

Strings, wie auch Integer und Real Variablen, werden durch die Typ-Deklaration nicht initialisiert. Bei Strings bedeutet das, daß sowohl die Länge als auch der Inhalt des Strings nach der Deklaration völlig unbekannt sind, numerische Variablen tragen einen unbekannten Zufallswert.

In Systemen ohne gepufferten RAM ist darauf zu achten, daß alle Variablen mit Werten besetzt werden, bevor man sie weiter auswertet. In Systemen mit gepuffertem RAM werden 2 Fälle unterschieden:

- Recovery nach Power-Down, Werte bleiben erhalten und werden nicht initialisiert
- Erststart oder Daten-Verlust im RAM: Werte werden initialisiert.

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

6 Häufig gestellte Fragen - Support

Das Programm wird aus nicht ersichtlichem Grund bei jedem Step neu compiliert, obwohl im Listing nichts verändert wurde?

Der Compiler vergleicht Datum und Uhrzeit des Quelltextes mit dem der zuletzt compilierten Datei (.TGR). Ist das Datum der Quelldatei neuer als die aktuelle Rechnerzeit, dann wird immer wieder neu compiliert, scheinbar unnötig.

Abhilfe: Quelltext abspeichern, damit das Datum und die Uhrzeit der Quelldatei aktualisiert werden. Im Netzwerk bleibt manchmal nur das Arbeiten auf dem lokalen Server.

Müssen BASIC-Tiger[®] einen externen Watch-Dog haben?

Natürlich spricht nichts gegen eine zusätzliche Sicherheit durch einen externen Watch-Dog, aber im BASIC-Tiger[®] und Tiny-Tiger[®] ist bereits ein Watchdog enthalten.

Wie wende ich die Instruktion PWM an?

Beim BASIC-Tiger[®] wird an Stelle einer speziellen BASIC-Instruktion ein entsprechender Device-Treiber verwendet. Durch die Wahl des Device-Treibers wird z.B. festgelegt, ob man die internen PWM-Pins des BASIC-Tigers[®] benutzt oder externe PWM-/DA-Bausteine anspricht. Nachdem der entsprechende Treiber eingebunden ist, wird einfach der gewünschte Wert, beim Treiber „PWM1.TDD“ zwischen 0 und 255, auf den Ausgabekanal des PWM-Treibers ausgegeben. Da BASIC-Tiger[®] ein Multitasking-System ist, bleibt der einmal eingestellte Wert so lange am PWM-Ausgang erhalten, bis man ihn gezielt durch einen neuen Wert überschreibt. Weder bleibt der BASIC-Tiger[®] bei der PWM-Ausgabe stehen, noch endet der eingestellte PWM-Wert, wenn die Programm-Ausführung zur nächsten BASIC-Instruktion kommt.

Kann ich das BASIC-Tiger[®]-Modul auch in Maschinensprache programmieren?

Nein. Die einzig verfügbare Ebene ist die Programmierung in Tiger-BASIC[®]. Tiger-BASIC[®] erzeugt sehr schnell ablaufenden Programmcode der dem Wunsch nach hoher Geschwindigkeit entspricht. Je nach Anforderungen an Durchsatz oder Antwortverhalten, wird ein Tiger Modul der entsprechenden Leistungsklasse ausgewählt.

Die Abwesenheit einer niedrigeren Sprachebene hat den Vorteil, daß Tiger-BASIC[®] Programme sehr leicht zwischen verschiedenen Modultypen portierbar sind. Ebenso ist es praktisch unmöglich, ungewollt das Laufzeit-System zum Absturz zu bringen, was zu der hohen Zuverlässigkeit des BASIC-Tigers[®] entscheidend beiträgt.

Häufig gestellte Fragen - Support

Wieviel Strom können die I/O-Pins des Moduls liefern oder ziehen?

Wenn alle Pins gleichmäßig belastet werden, sollten 1mA pro Pin nicht wesentlich überschritten werden (<1,5mA). Wenn maximal 8 Pins belastet sind, dann können diese bis zu 3,5mA ziehen oder liefern.

Was mache ich mit dem Pin 1 beim Modul ANN-X/X?

Mit VCC verbinden oder offen lassen.

Wie spreche ich die erweiterten Outputs auf dem Plug & Play Lab an?

Am Besten mit den ab Version 5.0 neuen Funktionen XOUT, XIN usw. direkt mit der physikalischen Adresse. Es geht aber auch mit den normalen IN- und OUT-Instruktionen, dann geht es genauso wie die internen Ports, lediglich mit anderer Adresse. Der mögliche Adress-Bereich für erweiterte Ports liegt zwischen: 10H...FFH (16...255). Auf dem Plug & Play Lab liegen die 8 ePort-Ausgänge im (logischen) Adress-Bereich: 10H...17H (16...23). Oberhalb der Tastatur sind alle 8 x 8 Ausgangs-Bits dieser ePorts durch ein LED vertreten. Ein Befehl zur Ausgabe eines Bytes auf den untersten ePort (10H) könnte also lauten: OUT 10H,0FFH,data_byte. Die physikalischen Adressen liegen auf dem Plug & Play Lab um 8 niedriger als die logischen Adressen.

6

Ich setze Port-Pins zum Beispiel mit der OUT-Instruktion, aber an den Pins tut sich nichts. Was ist der Grund?

Die Ports eines BASIC-Tiger® Moduls können grundsätzlich für ganz verschiedene Zwecke eingesetzt werden. Mit Instruktionen wie IN und OUT kann direkt auf die Ports eines Moduls zugegriffen werden - sofern diese Pins nicht schon anderweitig belegt sind. Die Mechanismen, die außerdem Ports verwenden sind: Device-Treiber sowie das ePort-System zur Erweiterung auf bis zu 1920 zusätzliche digitale Eingänge und Ausgänge.

Verwendet z.B. ein Device-Treiber wie „LCDI.TDD“ einen Port-Pin, so kann der Treiber bei Bedarf ihn für alle anderen Zugriffe (mit IN oder OUT) sperren. Es ist zwar noch erlaubt auf einen solchen Port mit OUT eine Ausgabe auszuführen, die reservierten Pins bleiben jedoch vollkommen unverändert. Dies ist ein Schutzmechanismus um die richtige Funktion der angeschlossenen Peripherie-Geräte zu sichern und es vereinfacht die Programmierung, da man selbst keine ausdrückliche Vorkehrung treffen muß um solche Pins unverändert zu lassen

Im Plug & Play Lab sind die Pins L60...L67 und L30...L37 zur Ansteuerung erweiterter Ports belegt und können somit nicht mit IN oder OUT angesprochen werden.

*Sie können auf die Aktivität erweiterter Ports mit der Compiler-Anweisung **USER_EPORT_NOACTIVE** verzichten (Siehe Hardware-Handbuch, 'Die Software-seite der erweiterten I/Os'). Dann sind diese Pins wie ursprünglich erwartet ansprechbar.*

Sind die Device-Treiber re-entrant?

Device-Treiber können wie auch Subroutinen grundsätzlich von beliebig vielen Tasks gleichzeitig verwendet werden. Im Unterschied zu Subroutinen steht hinter jedem Device-Treiber jedoch ein Ein- oder Ausgabe-Gerät. So lange das entsprechende Peripherie-Gerät sinnvoll angesprochen wird erhält man auch sinnvolle Resultate.

Ein Beispiel. Auf dem LC-Display des Plug & Play Labs können verschiedene Tasks gleichzeitig ihre Ausgaben machen. Man sollte dabei lediglich die Position des LCD-Cursors und die Puffer-Situation beachten. Eine Möglichkeit besteht darin, daß jede Task, die etwas auf dem LCD ausgeben will, in jeder PRINT-Instruktion zu Beginn eine absolute Cursor-Positionierung ausführt und danach die entsprechende Ausgabe. Da eine einzelne BASIC-Instruktion niemals mitten drin vom Multitasking-System unterbrochen werden, wird eine PRINT-Instruktion immer vollständig abgearbeitet. Die Cursor-Positionierung und die nachfolgende Ausgabe wird daher nicht gestört.

Weiterhin sollte man darauf achten, daß der Ausgabe Puffer des LCDs nicht ständig voll ist - was auch keinen Sinn machen würde. Ein vorwiegend voller Ausgangs-Puffer könnte folgenden Effekt erzeugen: Task X schreibt in den Ausgangs-Puffer bis er voll ist. Kurz danach will auch Task Y in diesen Puffer schreiben, kann es jedoch nicht, weil der Puffer immer noch zu voll ist. Task Y „wartet“ also. Etwas später, wenn Task X wieder ein PRINT auf diesen Device ausführt, kann der Puffer grade wieder genug Platz für weitere Daten haben, wodurch er wieder voll wird. Task X hat also erneut Daten abgesetzt. Die kurz darauf folgende Task Y würde ev. abermals einen vollen Puffer vorfinden Es entstehen Interferenzen von zufälligem Charakter.

6

Sind RAM oder Flash-Speicher über den Bus erweiterbar?

Nein. BASIC-Tiger[®]-Module sind autonome Computer Module, nicht CPU-Bausteine. BASIC-Tiger[®] haben keinen nach außen geführten Bus für Speicher-Erweiterungen, man setzt jeweils ein entsprechendes Modul mit der gewünschten Speichergöße ein. Dies stellt die hohe Zuverlässigkeit und gute EMV-Werte sicher.

In manchen Anwendungsfällen kann es jedoch sinnvoll sein zusätzlichen Speicher (RAM, ROM, Disks,...) anzuschließen z.B. um sehr große Daten-Mengen aufzunehmen oder bewegliche Speicher zu verwenden. Dies kann entweder über einen eigenen Device-Treiber, I2C-Bus oder über entsprechende BASIC-Subroutinen erreicht werden.

Kann man auf die Hardware direkt zugreifen via PEEK und POKE?

Die Instruktionen PEEK und POKE erlauben den Zugriff auf freie Bereiche des FLASH-Speichers. Üblicherweise werden hier Daten abgelegt, die auch nach AUS und EIN schalten verfügbar sein sollen. Dies können z.B. Calibrierungs-Tabellen, registrierte Meßwerte, Datenbanken, Betriebs-Stundenzähler etc. sein. Tiger-BASIC[®] stellt dem Programmierer den nicht vom BASIC-Programm belegten FLASH-Bereich für diesen Zweck zur Verfügung.

Häufig gestellte Fragen - Support

Entsprechend der Eigenschaft des FLASH-Speichers stehen jeweils Bereiche in ganzen Blöcken zur Verfügung. Tiger-BASIC[®] sorgt automatisch dafür, daß ein BASIC-Programm sich niemals versehentlich selbst löschen kann. Die Zahl und Größe der freien Sektoren hängt vom Modultyp und der Programm-Größe ab. Der FLASH-Speicher wird stets in ganzen Sektoren gelöscht (Inhalt = FFh). Weitere Hinweise zur Verwendung des FLASH siehe Hardware-Handbuch.

Eine Task bleibt ohne ersichtlichen Grund nach einer Weile stehen. Was ist der Grund?

Wahrscheinlich läuft der Stack von Zeit zu Zeit über. Stellen Sie sicher, daß in der Task und in von der Task benutzten Unterprogrammen keine endlos rekursiven Aufrufe stattfinden. Dann kann der Stack mit der Compiler-Anweisung `USER_STACK_SIZE` (Seite 40) vergrößert werden.

Warum werden nach dem Einschalten oder einem Reset die DIP-Schalter zunächst immer falsch gelesen?

Es wird stets das Ergebnis des letzten Scans gelesen. Nach einem Reset muss ein Scan abgewartet werden, also ca. 20msec.

6

Manche Zahlen werden von der USING-Formatierung ausgeschlossen, während andere Zahlen völlig korrekt formatiert werden. Wieso?

Wahrscheinlich haben Sie als Füllzeichen 'Null' angegeben. Als Füllzeichen sind keine Ziffern erlaubt, also auch nicht '0' (Null). Erhöhen Sie stattdessen die Anzahl der mindestens dargestellten Stellen.

Unterstützt BASIC-Tiger[®] die 1-wire-Chips von Dallas?

Wir empfehlen zur Unterstützung der 1-wire-Chips den ebenfalls von Dallas gelieferten Chip DS2480, der eine Umwandlung von RS-232 nach 1-wire vornimmt.

Rat und Hilfe

Funktioniert einmal etwas nicht wie erwartet? – Dann nehmen Sie Kontakt auf zum Tiger Support Team, das Ihnen gerne mit Rat und Hilfe für Ihr Problem zur Verfügung steht.

Zur Selbsthilfe sei auch auf den Abschnitt ‚Tiger-BASIC® für Fortgeschrittene‘ hingewiesen, wo häufig gemachte Fehler kurz zusammengefasst sind.

Damit es möglichst schnell geht, senden uns Sie bei jeder Anfrage mindestens diese Informationen und Files zu:

- Lauffähiges Programm-Beispiel als „TIG“ File:
- Reduziert auf ein möglichst einfaches Programm-Beispiel bei dem der Fehler auftritt. Maximal sollte dabei eine Seite herauskommen, meist sogar nur ein paar Zeilen.
- Service-Informationen: Senden Sie uns auch die Datei, die Tiger-BASIC® mit dem Befehl ‚Service-Informationen‘ aus dem Menü ‚Hilfe‘ als Fenster öffnet.
- Welches Tiger-Modul setzen Sie ein (Typ und Version)?
- Beschreiben Sie den Fehler so genau wie möglich.
- In welchem Zusammenhang tritt das Problem auf?
- Tritt es immer nachvollziehbar auf oder nur gelegentlich?
- Wie sind Sie am besten erreichbar: E-mail, Fax- und Telefonnummern für ev. Rückfragen und Antwort nicht vergessen!

6

BASIC-Tiger®-Service-Hotline, Mo...Fr. 8.00–17.00 Uhr:

0241 / 15 15 99

Wilke Technology GmbH
Krefelder Str. 147
Postfach 1727

D-52070 Aachen / Germany

Tel: 0241 / 918 900
Fax: 0241 / 918 9044
eMail: support@wilke.de

Häufig gestellte Fragen - Support

Leere Seite

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Leere Seite

7 Stichwortverzeichnis

—#—

#cm	37
#comment	37
#define	36
#endcm	37
#endcomment	37
#include	36
#project_model	38

—A—

ABS	72
ACOS	73
ACOT	74
ADD\$	75
AND\$	77
AND2\$	589
AND3\$	589
AND4\$	589
ANIB_BYTE\$	79
ANSI-Steuersequenzen	792
Approximation	
LIN_APROX	270
ARRAY	28
ARRAY of STRINGS	28
ASC	81
ASC_BCD\$	82
ASC_PBCD\$	84
ASCII-Tabelle	789
ASIN	86
ATAN	87
Auswahl	
SELECT\$	370

—B—

BASIC-Funktionen	39
Baudot-Code	791
BCD_ASC\$	88
BCD_PBCD\$	90
Berührtasten	
INDEX_2D	243
Betrag	
ABS	72
BIT	91
Bit Operatoren	34
BIT_DEMIX\$	92

BIT_MIRR	93
BIT_MIRR\$	95
BIT_MIX\$	97
Bitmap-Dateien	48
BITNOT	105
BYTE	24
BYTE_ANIB\$	106
BYTE_DEMIX\$	108
BYTE_INIB\$	110
BYTE_MIRR	112
BYTE_MIX\$	114
Bytemuster	
FILL\$	200

—C—

CALC_CRC	118
CALL	120
CHECK_FLASH	125
CHR\$	127
CLEAR_FIFO	128
CLOSE_LINE	731
CODE_1TON\$	129
Codieren	
BIT_DEMIX\$	92
BIT_MIX\$	97
DISTRIBUTE\$	173
SCAN_TO_CHAR\$	361
SCAN_TO_TOGGLE	365
comment	37
Compiler-Anweisungen	40
CONT_TASK	133
CONVERT\$	136
CONVERT\$	134, 136
COS	138
COSH	139
COT	140
COTH	141
CUT_AND_PASTE	142
CUT_AND_PASTE\$	144
CUT_BITS	146

—D—

DATA	44
DATA FILE	47
DATA FILTER	48
DATALABEL	27

Stichwortregister

Data-Labels	49
Data-Marken	27, 49
Datei ins Flash einbinden	47
Datentypen	24
DEBOUNCE	148
DEC	
MODULO_INC	290
MODULO_UPDO	293
DECODE	151
define	36
DELETE_PROG	164
Device-Treiber	12
GET	209
INPUT	250
INPUT_LINE	252
INSTALL_DEVICE	254
PRINT	328, 331
PUT	334
DIFF_TICKS	167
DIR_PIN	169
DIR_PORT	170
DISABLE_TSW	171
DISTANCE	745
DISTRIBUTE\$	173
DRAW_LINE	646
DRAW_NEXT_LINE	729

—E—

EBCDIC Codes	790
ENABLE_TSW	190
END	191
Entfernen	
von doppelten Zeichen aus einem	
String mit REMDOUBLE\$..	347
von Zeichen aus einem String mit	
REMOVE\$	348
Entprellen mit DEBOUNCE	148
ERASE_FLASH	192
erweiterte Ports	
XIN\$	534
XSETUP	541
EXIT_TASK	195
EXP	197
EXPE	198
Explizite Fehlerbehandlung	51

—F—

Fehlerbehandlung während der	
Laufzeit	51
Fehlerkorrigierend	

DECODE	151
FIFO	30
CLEAR_FIFO	128
deklarieren FIFO	199
freier Platz FREE_FIFO	208
GET_FIFO	212
INTEGRAL_FIFO	259
LEN_FIFO	266
PUT_FIFO	336
PUTU_FIFO	338
READ_FIFO	345
START_FIFO	437
UNGET_FIFO	492
FILL\$	200
FILL_AREA	747
FIX	203
Flash	
CHECK_FLASH	125
DATA	44
Datei ins Flash einbinden	47
ERASE_FLASH	192
PEEK_FLASH	318
POKE_FLASH	320
POKEM_FLASH	322
SET_DATA_ADDR	43
Flashdaten lesen	50
FOR...NEXT	204
Formatstrings	499
FRAME	206
FREE_FIFO	208
Funktionen	14
Funktionen, math.	543

—G—

Ganzzahl	
INT	258
GET	209
GET_FIFO	212
GOTO	214
Grafik einbinden	48
GRAPHIC_COPY	608
GRAPHIC_EXP\$	639
GRAPHIC_FILL_MASK	630
GRAPHIC_MASK_COPY	600
GRAPHIC_MIRROR	605
Grenzwert	
LIMIT	268
Großbuchstaben	
UPPER\$	496
UPPER7\$	497

—H—

HREAL 215

—I—

I2C_ACK_POLL 216
 I2C_READ\$ 218
 I2C_RESULT 220
 I2C_SETUP 222
 I2C_WRITE 225
 IF...THEN 227
 IMASK 229
 Implizite Fehlerbehandlung 52
 IN230
 INC
 MODULO_INC 290
 MODULO_UPDO 293
 include 36
 Index
 SELECT\$ 370
 INDEX 231
 INDEX_1D 233
 INDEX_2D 243
 INIB_BYTE\$ 248
 INPUT 250
 INPUT_LINE 252
 INSTALL_DEVICE 254
 INSTR 256
 INT 258
 INTEGRAL_FIFO 259
 INV_BIT 262
 INVERT 628
 Invertieren
 BITNOT 105

—K—

Kennzeichnung R+C 798
 Kommentare 18
 Komprimieren
 ANIB_BYTE\$ 79
 ASC_BCD\$ 82
 ASC_PBCD\$ 84
 BCD_ASC\$ 88
 BCD_PBCD\$ 90
 Komprimierung
 BYTE_ANIB\$ 106
 BYTE_INIB\$ 110
 INIB_BYTE\$ 248
 PACK_BYTE\$ 313
 PBCD_ASC\$ 315

PBCD_BCD\$ 317
 UNPACK_BYTE\$ 494

—L—

Labels im Flash 27, 49
 LD 263
 Leerzeichen
 anhängende abschneiden mit
 TRIM\$ 490
 LEFT\$ 264
 LEN 265
 LEN_FIFO 266
 LET\$ 267
 LIMIT 268
 LIN_APROX 270
 Liste
 FRAME 206
 SELECT\$ 370
 LL_IPOINT_AND 272
 LL_IPOINT_IN 273
 LL_IPOINT_OR 274
 LL_IPOINT_OUT 275
 LL_IPOINT_PULSE 276
 LL_IPOINT_XOR 277
 LLTOR 278
 LN 279
 LOG 280
 LONG 24, 25
 LOOP 281
 LREAL 283
 LTR 284

—M—

MASK 285
 MAX_LEN 286
 Menü
 INDEX 231
 SELECT\$ 370
 MID\$ 287
 Mirror
 BIT_MIRR 93
 BIT_MIRR\$ 95
 BYTE_MIRR 112
 GRAPHIC_MIRROR 605
 MIRROR\$ 288
 Mittelwert
 INTEGRAL_FIFO 259
 MOD 289
 MODULO_INC 290
 MODULO_UPDO 293

Stichwortregister

Multitasking 12
 Muster
 FILL\$ 200

—N—

NFROMS 297
 NTO\$ 299

—O—

ON_ERROR_CALL 51
 ON_ERROR_GOTO 51
 ON_ERROR_RESET 303
 ON_ERRTASK 304, 306
 ON_ERRTASK_CALL 52
 ON_ERRTASK_GOTO 52
Operatoren 34
 OR\$ 307
 OR2\$ 589
 OR3\$ 589
 OR4\$ 589
 OUT 309

—P—

PACK_BYTE\$ 313
 PARITY 311
 Pause
 WAIT_DURATION 523
 PBCD_ASC\$ 315
 PBCD_BCD\$ 317
 PC-Modus
 DELETE_PROG 164
 PEEK_FLASH 318
 POKE_FLASH 320
 POKEM_FLASH 322
 Ports, erweiterte
 XSETUP 541
 Präprozessor 36
 PRIME 325
 PRINT 328
 PRINT_USING 331
 project_model 38
 Prüfsumme
 CALC_CRC 118
 PUT 334
 PUT_FIFO 336
 PUTU_FIFO 338

—Q—

QUICK_WORD_COS 340

QUICK_WORD_SIN 342

—R—

RANDOMIZE 344
 READ_FIFO 345
 REAL 24, 25
 RELEASE_TASK 346
 REMDOUBLE\$ 347
 REMOVE\$ 348
 RES_BIT 349
 Reset
 RESTART_PROG 350
 RESTART_PROG 350
 RETURN 351
 RFROMS 352
 RIGHT\$ 354
 RND 355
 RTL 356
 RTOS\$ 357
 RUN_TASK 359
 RUN-Modus
 verlassen mit DELETE_PROG
 164
 Runtime-System
 THIS_VERS 487

—S—

SCAN_TO_CHAR\$ 361
 SCAN_TO_CHAR_SETUP 363
 SCAN_TO_TOGGLE 365
 Schreibweise 17
 Schutz der Software im Modul 42
 SCRAMBLE\$ 366
 SELECT\$ 370
 SERIAL_NO\$ 372
 SERIN/OUT
 Device-Treiber SER2 388
 SET_BASE 733
 SET_BIT 374
 SET_DATA_ADDR 43
 SET_DOT 743
 SET_GRAREA 741
 SET_LEN\$ 375
 SET_ROTATION 738
 SET_SCALE 736
 SET_SERIAL_NO 376
 SET_SYSVARN 379
 SET_TASK_PRIO 381
 SET_TICKS f2 383

SGN	384
SHIFT_IN	385
SHIFT_OUT	389
SHIFTL\$	393
SHIFTM\$	396
SIGNAL_ADD	399
SIGNAL_AVERAGE	401
SIGNAL_BIGGER	403
SIGNAL_CUT_AMP	404
SIGNAL_CUT_MAX	406
SIGNAL_CUT_MIN	408
SIGNAL_GAP_END	409
SIGNAL_GAPI_END	411
SIGNAL_MAX	413
SIGNAL_MAX_AMPL	415
SIGNAL_MIN	417
SIGNAL_MUL	419
SIGNAL_NEXT_GAP	421
SIGNAL_NEXT_GAPI	423
SIGNAL_OFFSET	425
SIGNAL_SCALE	426
SIGNAL_SMALLER	428
SIGNAL_SMOOTH	429
SIGNEXT	431
SIN	434
SINH	435
Softwareschutz im Modul	42
Sound einbinden	48
Spiegeln	
BIT_MIRR	93
BIT_MIRR\$	95
BYTE_MIRR	112
GRAPHIC_MIRROR	605
MIRROR\$	288
SQRT	436
Stack	40
START_FIFO	437
STOP_TASK	439
STOS\$	441
STR\$	444
STRIS\$	445
STRING	24, 26
Strings	41
Struktur	
NFROMS	297
NTOS\$	299
RFROMS	352
RTOS\$	357
STOS\$	441
SUB	446

Suchen	
freie Stelle im CHECK_FLASH	125
im String INSTR	256
in Liste INDEX	231
SWITCH...CASE	450
SWITCHI...CASE	454
Systemzeit	
DIFF_TICKS	167
TICKS	489
SYSVAR\$	458
SYSVARN	460

—T—

Tabelle	
FRAME	206
TABS	463
TAN	470
Trans-Aktions-Nummern	471
TAN_CHK	473
TAN_COUNT	477
TAN_DEL	479
TAN_FIRST	483
TANH	485
Task	
CONT_TASK	133
DISABLE_TSW	171
ENABLE_TSW	190
EXIT_TASK	195
ON_ERROR_RESET	303
ON_ERRTASK	304, 306
Priorität setzen mit	
SET_TASK_PRIO	381
RELEASE_TASK	346
starten RUN_TASK	359
stoppen STOP_TASK	439
TASK	486
Tastatur	
DEBOUNCE	148
INDEX_1D	233
SCAN_TO_CHAR\$	361
SCAN_TO_TOGGLE	365
THIS_VERS	487
TICKS	489
Tiger-Shortcuts	796
Touchpanel	
INDEX_2D	243
SCAN_TO_CHAR\$	361
SCAN_TO_TOGGLE	365
Touch-Screen	

Stichwortregister

DEBOUNCE 148
TRIM\$ 490

—U—

Umwandlung
 LONG zu REAL mit LTR 284
 REAL zu LONG mit RTL 356
UNGET_FIFO 492
UNPACK_BYTE\$ 494
Unterprogramme 21
UPPER\$ 496
UPPER7\$ 497
USER_VAR_INIT 31, 41
USER_VAR_STRICT 32
USER_SECURITY 42
USER_STACK_SIZE 40
USER_STRING_SIZE 41
USING 498

—V—

VAL_NUM 518
VAL_REAL 520
Variable 41
Variable deklarieren 32
Variable initialisieren 31, 41
Variablenamen 33
Verschlüsseln
 BIT_DEMIX\$ 92
 DISTRIBUTE\$ 173
Verschlüsselung
 BIT_MIX\$ 97
 BYTE_DEMIX\$ 108
 BYTE_MIX\$ 114
 SCRAMBLE\$ 366
Version
 THIS_VERS 487
Vorzeichen

erweitern mit SIGNEXT 431
SGN 384

—W—

WAIT_CLOCK 521
WAIT_DURATION 523
WAIT_NEXT 525
WAV-Dateien 48
Wert
 VAL_NUM 518
 VAL_REAL 520
WHILE 527
Widerstandstastatur
 INDEX_1D 233
Windows-Shortcuts 794
WORD 24

—X—

XBUS_INR\$ 530
XBUS_OUTR 532
XIN\$ 534
XOR\$ 537
XOR1\$ 589
XOUT 539
XSETUP 541

—Z—

Zeilen, überlange 18
Zeit
 DIFF_TICKS 167
 SET_TICKS 383
 TICKS 489
Zufallszahl
 RND 355
Zufallszahlen
 RANDOMIZE 344

Einführung	1
Die Tiger-BASIC [®] -Sprache	2
Instruktionen und Funktionen	3
Grafik im BASIC-Tiger [®]	4
Tiger-BASIC [®] für Fortgeschrittene	5
Häufig gestellte Fragen - Support	6
Stichwortverzeichnis	7
Anhang	8

Stichwortregister

Leere Seite

8 Anhang

ASCII-Tabelle

ASCII, gesprochen 'aski', ist eine Abkürzung für **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange'. Der ASCII-Code ist wahrscheinlich der am weitesten verbreitete Code zur Darstellung von Buchstaben, Ziffern, einigen Sonderzeichen und Steuerzeichen. Er findet im PC, Macintosh und Internet Verwendung.

CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC
NUL	00	000	SP	20	032	@	40	064		60	096
SOH	01	001	!	21	033	A	41	065	a	61	097
STX	02	002	"	22	034	B	42	066	b	62	098
ETX	03	003	#	23	035	C	43	067	c	63	099
EOT	04	004	\$	24	036	D	44	068	d	64	100
ENQ	05	005	%	25	037	E	45	069	e	65	101
ACK	06	006	&	26	038	F	46	070	f	66	102
BEL	07	007	'	27	039	G	47	071	g	67	103
BS	08	008	(	28	040	H	48	072	h	68	104
HT	09	009	)	29	041	I	49	073	i	69	105
LF	0A	010	*	2A	042	J	4A	074	j	6A	106
VT	0B	011	+	2B	043	K	4B	075	k	6B	107
FF	0C	012	,	2C	044	L	4C	076	l	6C	108
CR	0D	013	-	2D	045	M	4D	077	m	6D	109
SO	0E	014	.	2E	046	N	4E	078	n	6E	110
SI	0F	015	/	2F	047	O	4F	079	o	6F	111
DLE	10	016	0	30	048	P	50	080	p	70	112
D1	11	017	1	31	049	Q	51	081	q	71	113
D2	12	018	2	32	050	R	52	082	r	72	114
D3	13	019	3	33	051	S	53	083	s	73	115
D4	14	020	4	34	052	T	54	084	t	74	116
NAK	15	021	5	35	053	U	55	085	u	75	117
SYN	16	022	6	36	054	V	56	086	v	76	118
ETB	17	023	7	37	055	W	57	087	w	77	119
CAN	18	024	8	38	056	X	58	088	x	78	120
EM	09	025	9	39	057	Y	59	089	y	79	121
SUB	1A	026	:	3A	058	Z	5A	090	z	7A	122
ESC	1B	027	;	3B	059	[	5B	091	{	7B	123
FS	1C	028	<	3C	060	\	5C	092		7C	124
GS	1D	029	=	3D	061	]	5D	093	}	7D	125
RS	1E	030	>	3E	062	^	5E	094	~	7E	126
US	1F	031	?	3F	063	_	5F	095	DEL	7F	127

EBCDIC Codes

EBCDIC, gesprochen 'eb-si-dik', ist eine Abkürzung für **Extended Binary-Coded Decimal Interchange Code**. EBCDIC ist ein IBM-Code zur Darstellung von Buchstaben, Ziffern und einigen Sonderzeichen. Obwohl der Code auf großen IBM-Computern weit verbreitet ist, verwenden die meisten anderen Computer (PCs, Macintosh) den ASCII-Code.

Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
blank	40	64	a	81	129	A	C1	193	0	F0	240
.	4B	75	b	82	130	B	C2	194	1	F1	241
<	4C	76	c	83	131	C	C3	195	2	F2	242
(	4D	77	d	84	132	D	C4	196	3	F3	243
+	4E	78	e	85	133	E	C5	197	4	F4	244
	4F	79	f	86	134	F	C6	198	5	F5	245
&	50	80	g	87	135	G	C7	199	6	F6	246
!	5A	90	h	88	136	H	C8	200	7	F7	247
\$	5B	91	i	89	137	I	C9	201	8	F8	248
*	5C	92	j	91	145	J	D1	209	9	F9	249
)	5D	93	k	92	146	K	D2	210			
;	5E	94	l	93	147	L	D3	211			
-	60	96	m	94	148	M	D4	212			
/	61	97	n	95	149	N	D5	213			
,	6B	107	o	96	150	O	D6	214			
%	6C	108	p	97	151	P	D7	215			
_	6D	109	q	98	152	Q	D8	216			
>	6E	110	r	99	153	R	D9	217			
?	6F	111	s	A2	162	S	E2	226			
:	7A	122	t	A3	163	T	E3	227			
#	7B	123	u	A4	164	U	E4	228			
@	7C	124	v	A5	165	V	E5	229			
'	7D	125	w	A6	166	W	E6	230			
=	7E	126	x	A7	167	X	E7	231			
"	7F	127	y	A8	168	Y	E8	232			
			z	A9	169	Z	E9	233			

Baudot-Code-Satz

Baudot, gesprochen 'bodoh', wurde 1874 erstmals in einem ,Telegraph' (Fernschreiber) eingesetzt, ist aber bis heute in einigen Bereichen zur datenübertragung in Gebrauch. Die Zeichen ,LTRS=Letters' und ,FIGS=Figures' schalten zwischen den beiden Zeichensätzen um. Das rechts-stehende niederwertige Bit (LSB) wird zuerst gesendet. Den Codes werden entweder Buchstaben oder Sonder- und Steuerzeichen zugeordnet.

LTRS	FIGS	HEX	BITS
A	-	03	00011
B	?	19	11001
C	:	0E	01110
D	\$	09	01001
E	3	01	00001
F	!	0D	01101
G	&	1A	11010
H	STOP	14	10100
I	8	06	00110
J	'	0B	01011
K	(	0F	01111
L	)	12	10010
M	.	1C	11100
N	,	0C	01100
O	9	18	11000
P	0	16	10110
Q	1	17	10111
R	4	0A	01010
S	BELL	05	00101
T	5	10	10000
U	7	07	00111
V	;	1E	11110
W	2	13	10011
X	/	1D	11101
Y	6	15	10101
Z	"	11	10001
n/a	n/a	00	00000
CR	CR	08	01000
LF	LF	02	00010
SP	SP	04	00100
LTRS	LTRS	1F	11111
FIGS	FIGS	1B	11011

ANSI-Steuersequenzen

Die in der Tabelle gelisteten ANSI-Steuersequenzen sind nützlich, wenn ein serielles ANSI-Terminal bzw. ein PC mit Terminalprogramm mit ANSI-Emulation angeschlossen ist. Auf dem Bildschirm lassen sich so die Ausgaben positionieren und eventuell farblich darstellen. Die Liste erhebt keinen Anspruch auf Vollständigkeit.

ANSI-Sequenzen beginnen mit ESC gefolgt von „[“. Es folgen weitere Bytes als Kommando, Parameter und eventuell ein Abschluß. In der Tabelle wird die Darstellung `<0>` für ein Byte verwendet, so wie in Strings in Tiger-BASIC[®] üblich. Folgende Abkürzungen gelten:

`<zb>` = Zeile als Binärwert
`<sb>` = Spalte als Binärwert
`<nb>` = binärer Wert.

Wird ein Wert nicht angegeben, wird der Wert ‚eins‘ angenommen. In Tiger-BASIC[®] wird eine ANSI-Sequenz zum Beispiel so ausgegeben (Cursor in Zeile 2, Spalte 3):

PUT #1, “<27>[<2>;<3>H“

ANSI-Sequenz	Funktion
ESC [2J	Löscht den Bildschirm und setzt den Cursor auf die Home -Position.
ESC [<zb>;<sb>H	Bewegt den Cursor an die angegebene Position (Home ist <1>;<1>)
ESC [<zb>;<sb>f	Bewegt den Cursor an die angegebene Position (Home ist <1>;<1>)
ESC [<nb>A	Bewegt den Cursor um n Zeilen nach oben, ohne die Spalte zu beeinflussen, bis maximal Position 1.
ESC [<nb>B	Bewegt den Cursor um n Zeilen nach unten, ohne die Spalte zu beeinflussen, bis maximal zur letzten Zeile.
ESC [<nb>C	Bewegt den Cursor um n Zeilen nach rechts, ohne die Zeile zu beeinflussen, bis maximal zur letzten Position.
ESC [<nb>D	Bewegt den Cursor um n Zeilen nach links, ohne die Zeile zu beeinflussen, bis maximal zur Position 1.
ESC [6n	Gerät gibt den Cursorpositions-Report in dieser Form aus: ESC [<zb>;<sb>R
ESC [s	Gerät speichert die aktuelle Cursorposition.
ESC [u	Gerät bewegt den Cursor an die zuletzt gespeicherte Position.
ESC [K	Löscht den Inhalt einer Zeile ab der Cursorposition bis zum Ende der Zeile.
ESC [<nb>;...;<nb>m	Funktion ‘Set Graphic Rendition’ schaltet grafische Attribute. Siehe Tabelle im Anschluß

Anhang - ANSI-Steuersequenzen

ESC-m-Parameter	Funktion
0	Alle Attribute aus
1	Fett ein
2	Schwach ein
3	Kursiv ein
5	Blinken ein
6	Schnelles Blinken ein
7	Invers ein
8	Verdeckt ein
30...47	Verschiedene Vorder- und Hintergrundfarben

Windows 95/98/NT Shortcuts

Die in der Tabelle stellt eine Übersicht über die wichtigsten Windows 95/98/NT Shortcuts dar. Die Liste erhebt keinen Anspruch auf Vollständigkeit.

Eingabe	Funktion
F1	Hilfe aufrufen
F10	Menüleiste (de)aktivieren
SHIFT + F10	Kontextmenü aufrufen
CTRL + C	Kopieren
CTRL + X	Ausschneiden
CTRL + V	Einfügen
CTRL + A	Alles markieren
CTRL + S	Speichern
CTRL + N	Neues Fenster öffnen
CTRL + Z	Letzte Aktion widerrufen
CTRL + ESC	Startmenü aufrufen
CTRL + F4	Aktuelles Unter-Fenster in Programm schließen
CTRL + TAB	Vorwärts Unter-Fenster / Registriertkarten wechseln
CTRL + SHIFT + TAB	Rückwärts Unter-Fenster / Registriertkarten wechseln
CTRL + O	Öffnen
CTRL + P	Drucken
CTRL + Pos1	Zum Anfang eines Dokuments
CTRL + Ende	Zum Ende eines Dokuments
CTRL + WIN + F	Suche Computer
CTRL + ALT + DEL	Öffnet Fenster „Anwendung schließen“

Anhang - Windows 95/98/NT Shortcuts

Eingabe	Funktion
WIN + TAB	Umschalten zwischen Programmen in Task-Leiste
WIN + E	Explorer starten
WIN + F1	Hilfe aufrufen
WIN	Startmenü aufrufen
WIN + BREAK	System-Eigenschaften aufrufen
WIN + D	Minimieren bzw. Wiederherstellen aller Fenster
WIN + M	Minimieren aller Fenster
WIN + SHIFT + M	Zurücknehmen minimieren aller Fenster
WIN + R	Dialogfeld „Ausführen“ aufrufen
WIN + F	Dialogfeld „Suchen“ aufrufen
ALT + TAB	Schaltet zum nächsten Programm in der Taskleiste
ALT + SHIFT + TAB	Schaltet zum vorigen Programm in der Taskleiste
ALT + F4	Programm(fenster) schließen
ALT + F10	Menüleiste einer Anwendung aktivieren
ALT + SPACE	Systemmenü der aktuellen Anwendung anzeigen
ALT + ESC	Wechselt zwischen Explorer und anderen anderen Anwendungen
ALT + -	Systemmenü des aktuellen Unterfensters anzeigen

Short-Cuts Tiger-BASIC® Version 5

Eingabe	Funktion
Pfeiltasten	Ein Zeichen vor oder zurück
Pfeiltasten	Eine Zeile vor oder zurück
STRG-Pfeiltasten	Ein Wort vor oder zurück
Bild-auf-, Bild-ab-Taste	Eine Bildschirmseite vor oder zurück
STRG-Pos1	An den Anfang des Textes
STRG-Ende	An das Ende des Textes
STRG-F7/F8	Zum nächsten/vorigen Fehler
Bildlaufleisten	Schnell vor oder zurück
Suchfunktion	An eine bestimmte Stelle im Text
Strg-S	Datei-Speichern
Strg-Z	Bearbeiten-Rückgängig
Strg-X	Bearbeiten-Ausschneiden
Strg-C	Bearbeiten-Kopieren
Strg-V	Bearbeiten-Einfügen
Strg-A	Bearbeiten-Alles markieren
Strg-F	Suchen-Suche nach
Strg-R	Suchen-Ersetzen
F3	Suche-Wiederholen
Strg-G	Gehe zu Zeile
Strg-F9	Anzeige Meldungen
Strg-M	Anzeige-Tigerstatus
ALT-F5	Anzeige-Ausdruck auswerten
Strg-F5	Anzeige-Überwachte Ausdrücke
Strg-W	Anzeige-Überwachte Ausdrücke aktualisieren
Strg-N	Anzeige-zu überwachten Ausdrücken hinzufügen
F4	Starten-Compilieren

Anhang - Short-Cuts Tiger-BASIC® Version 5

Eingabe	Funktion
F5	Starten-Ausführen
Strg-L	Starten-Programm laden
Strg-D	Starten-Programm löschen
F6	Debug-nächste Anweisung (in Task)
ALT-F6	Debug-Anweisungen ausführen (in Task)
F7	Debug-Unterprogramm überspringen (in Task)
ALT-F7	Debug-Unterprogramme überspringen (in Task)
F8	Debug-nächste Anweisung (überall)
ALT-F8	Debug-Anweisungen ausführen (überall)
Strg-T	Debug-bis zum Cursor ausführen
Strg-H	Debug-Programm anhalten
Strg-E	Debug-Programm beenden (Reset)
F2	Debug-Haltepunkt umschalten
ALT-F2	Debug-Haltepunkt hinzufügen
Strg-K	Debug-Alle Haltepunkte löschen
Strg-B	Debug-Editierschutz ein-/ausschalten
Strg-F3	Optionen-Übertragung
F1	Hilfe

Anhang - Kennzeichnung von Widerständen und Kondensatoren

Kennzeichnung von Widerständen und Kondensatoren

Kennzeichnung von Widerständen / Kondensatoren mit Angabe des Temperatur-Koeffizienten (DIN-41429 / DIN-IEC-62 / IEC 115-1-4.5)

Farbcodes

Farbe	Ziffer	Multiplikator	Toleranz	Temp.-Koeff.
schwarz	0	10^0	-	$\pm 250 * 10^{-6}/K$
braun	1	10^1	$\pm 1\%$	$\pm 100 * 10^{-6}/K$
rot	2	10^2	$\pm 2\%$	$\pm 50 * 10^{-6}/K$
orange	3	10^3	-	$\pm 15 * 10^{-6}/K$
gelb	4	10^4	-	$\pm 25 * 10^{-6}/K$
grün	5	10^5	$\pm 0,5\%$	$\pm 20 * 10^{-6}/K$
blau	6	10^6	$\pm 0,25\%$	$\pm 10 * 10^{-6}/K$
violett	7	10^7	$\pm 0,1\%$	$\pm 5 * 10^{-6}/K$
grau	8	10^8	-	$\pm 1 * 10^{-6}/K$
weiß	9	10^9	-	-
silber	-	10^{-2}	$\pm 10\%$	-
gold	-	10^{-1}	$\pm 5\%$	-

Anhang - Kennzeichnung von Widerständen und Kondensatoren

Wertkennzeichnung durch Buchstaben

(DIN 1301/12.93, EN 60 062/10.94)

Kennbuchstabe	Name	Multiplikator
y	Yocto	10^{-24}
z	Zepto	10^{-21}
a	Atto	10^{-18}
f	Femto	10^{-15}
p	Pico	10^{-12}
n	Nano	10^{-9}
μ	Mikro	10^{-6}
m	Milli	10^{-3}
c	Centi	10^{-2}
d	Dezi	10^{-1}
R,F	-	10^0
da	Deka	10^1
h	Hekto	10^2
K	Kilo	10^3
M	Mega	10^6
G	Giga	10^9
T	Tera	10^{12}
P	Peta	10^{15}
E	Exa	10^{18}
Z	Zetta	10^{21}
Y	Yotta	10^{24}

Anhang - Kennzeichnung von Widerständen und Kondensatoren

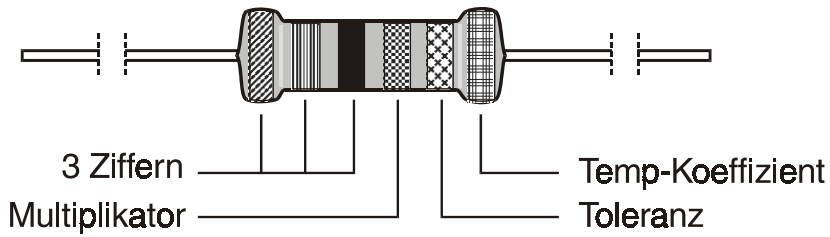
Toleranzkennzeichnung durch Buchstaben

(EN 60 062/10.94)

Abweichungen	
Toleranz	Buchstabe
Symmetrische Abweichung:	
±0,1%	B
±0,25%	C
±0,5%	D
±1%	F
±2%	G
±5%	J
±10%	K
±20%	M
±30%	N
Unsymmetrische Abweichung:	
+30...-10%	Q
+50...-10%	T
+50...-20%	S
+80...-20%	Z
Symmetrische Abweichung für Kapazitätswerte < 10pF:	
±0,1 pF	B
±0,25pF	C
±0,5pF	D
±1pF	F

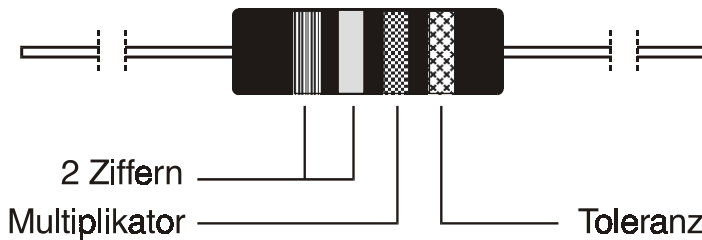
Anhang - Kennzeichnung von Widerständen und Kondensatoren

Anbringung der Farbcodes an Widerständen:



Beispiel: rot-gelb-weiß-rot-braun-rot

24,9 kOhm, +/-1%, +/-50 ppm/°K



Beispiel: gelb-violett-orange-gold

47 kOhm, +/-5%

Anhang - Kennzeichnung von Widerständen und Kondensatoren

Mittlere Schrittweite für Widerstands-Wachstum aufeinanderfolgender Werte:

E3	E6	E12	E24	E48	E96	E192
+115%	+47,8%	+21,2%	+10,07%	+4,914%	+2,428%	+1,206%

Normreihen für Widerstandswerte

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
100	100	100	100	100	100	100
						101
					102	102
						104
				105	105	105
						106
					107	107
						109
			110	110	110	110
						111
					113	113
						114
				115	115	115
						117
					118	118
		120	120			120
				121	121	121
						123
					124	124
						126
				127	127	127
						129
			130		130	130
						132
				133	133	133
						135
					137	137
						138

Anhang - Kennzeichnung von Widerständen und Kondensatoren

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
				140	140	140
						142
					143	143
						145
				147	147	147
						149
	150	150	150		150	150
						152
				154	154	154
						156
					158	158
			160			160
				162	162	162
						164
					165	165
						167
				169	169	169
						172
					174	174
						176
				178	178	178
		180	180			180
					182	182
						184
				187	187	187
						189
					191	191
						193
				196	196	196
						198
			200		200	200
						203
				205	205	205
						208
					210	210
						213

Anhang - Kennzeichnung von Widerständen und Kondensatoren

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
				215	215	215
						218
220	220	220	220		221	221
						223
				226	226	226
						229
					232	232
						234
				237	237	237
			240			240
					243	243
						246
				249	249	249
						252
					255	255
						258
				261	261	261
						264
					267	267
		270	270			271
				274	274	274
						277
					280	280
						284
				287	287	287
						291
					294	294
						298
			300	301	301	301
						305
					309	309
						312
				316	316	316
						320
					324	324
						328

Anhang - Kennzeichnung von Widerständen und Kondensatoren

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
	330	330	330	332	332	332
						336
					340	340
						344
				348	348	348
						352
					357	357
			360			361
				365	365	365
						370
					374	374
						379
				383	383	383
						388
		390	290		392	392
						397
				402	402	402
						407
					412	412
						417
				422	422	422
						427
			430		432	432
						437
				442	442	442
						448
					453	453
						459
				464	464	464
470	470	470	470			470
					475	475
						481
				487	487	487
						493
					499	499
						505

Anhang - Kennzeichnung von Widerständen und Kondensatoren

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
			510	511	511	511
						517
					523	523
						530
				536	536	536
						542
					549	549
						556
		560	560	562	562	562
						569
					576	576
						583
				590	590	590
						597
					604	604
						612
			620	619	619	619
						626
					634	634
						642
				649	649	649
						657
					665	665
						673
	680	680	680	681	681	681
						690
					698	698
						706
				715	715	715
						723
					732	732
						741
			750	750	750	750
						759
					768	768
						777

Anhang - Kennzeichnung von Widerständen und Kondensatoren

E3	E6 ±20%	E12 ±10%	E24 ±5%	E48 ±2%	E96 ±1%	E192 ±0,5%
				787	787	787
						796
					806	806
						816
		820	820	825	825	825
						835
					845	845
						856
				866	866	866
						876
					887	887
						898
			910	909	909	909
						920
					931	931
						942
				953	953	953
						965
					976	976
						998

Anhang - Kennzeichnung von Widerständen und Kondensatoren

Leere Seite

BASIC-Tiger® Modul A – Nutzung der Pins

Übersicht der I/O-Pin-Nutzung der wichtigsten Device-Treiber. Einige Funktionen sind fest an den entsprechenden Pin gebunden (PWM, PLSO1, PLSIN1), andere sind lediglich die Standardbelegung, lassen sich aber auf andere Pins verlegen (LCD, Strobe, Busy):

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	res.	1	46	VCC	
D0	L60	2	45	Batt.	
D1	L61	3	44	AGND	
D2	L62	4	43	Vref	
D3	L63	5	42	An3	
D4	L64	6	41	An2	
D5	L65	7	40	An1	
D6	L66	8	39	An0	
D7	L67	9	38	Alarm	
Busy	L70	10	37	L41/PC	
Strobe	L71	11	36	L40	
PWM0	L72	12	35	L42	Beep
PWM1	L73	13	34	L37	LCD1-RS
LCD-WR	L80	14	33	L36	LCD1-E
LCD-RD	L81	15	32	L35	INE
LCD-CE	L82	16	31	L34	Dclk
LCD-CD	L83	17	30	L33	Aclk
PLSIN1	L84	18	29	L95	RTS
	L85	19	28	L94	Rx1
PLSO1	L86	20	27	L93	Tx1
	L87	21	26	L92	CT0
	Reset-in	22	25	L91	Rx0
	GND	23	24	L90	Tx0

Anhang - BASIC-Tiger® Modul A – Nutzung der Pins

Leere Seite

Anhang - BASIC-Tiger® Modul A – Nutzung der Pins

Freie Tabelle BASIC-Tiger®-Modul A als Kopiervorlage. Projekt:

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	res.	1	46	VCC	
	L60	2	45	Batt.	
	L61	3	44	AGND	
	L62	4	43	Vref	
	L63	5	42	An3	
	L64	6	41	An2	
	L65	7	40	An1	
	L66	8	39	An0	
	L67	9	38	Alarm	
	L70	10	37	L41/PC	
	L71	11	36	L40	
	L72	12	35	L42	
	L73	13	34	L37	
	L80	14	33	L36	
	L81	15	32	L35	
	L82	16	31	L34	
	L83	17	30	L33	
	L84	18	29	L95	
	L85	19	28	L94	
	L86	20	27	L93	
	L87	21	26	L92	
	Reset-in	22	25	L91	
	GND	23	24	L90	

Anhang - BASIC-Tiger® Modul A – Nutzung der Pins

Leere Seite

TINY-Tiger® – Nutzung der Pins

Übersicht der I/O-Pin-Nutzung der wichtigsten Device-Treiber. Einige Funktionen sind fest an den entsprechenden Pin gebunden (PWM, PLSO1, PLSIN1), andere sind lediglich die Standardbelegung, lassen sich aber auf andere Pins verlegen (LCD, Strobe, Busy):

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
D0	L60	1	44	VCC	
D1	L61	2	43	Batt.	
D2	L62	3	42	Vref	
D3	L63	4	41	AGND	
D4	L64	5	40	An3	
D5	L65	6	39	An2	
D6	L66	7	38	An1	
D7	L67	8	37	An0	
Busy	L70	9	36	L41/PC	
Strobe	L71	10	35	res.	
PWM0	L72	11	34	Alarm	
PWM1	L73	12	33	L37	LCD1-RS
LCD-WR	L80	13	32	L36	LCD1-E
LCD-RD	L81	14	31	L35	INE
LCD-CE	L82	15	30	L34	Dclk
LCD-CD	L83	16	29	L33	Aclk
PLSIN1	L84	17	28	L95	RTS
	L85	18	27	L94	Rx1
PLSO1	L86	19	26	L93	Tx1
	L87	20	25	L92	CT0
	Reset-in	21	24	L91	Rx0
	GND	22	23	L90	Tx0

Anhang - TINY-Tiger® – Nutzung der Pins

Leere Seite

Anhang - TINY-Tiger® – Nutzung der Pins

Freie Tabelle TINY-Tiger® als Kopiervorlage. Projekt:

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	L60	1	44	VCC	
	L61	2	43	Batt.	
	L62	3	42	Vref	
	L63	4	41	AGND	
	L64	5	40	An3	
	L65	6	39	An2	
	L66	7	38	An1	
	L67	8	37	An0	
	L70	9	36	L41/PC	
	L71	10	35	res.	
	L72	11	34	Alarm	
	L73	12	33	L37	
	L80	13	32	L36	
	L81	14	31	L35	
	L82	15	30	L34	
	L83	16	29	L33	
	L84	17	28	L95	
	L85	18	27	L94	
	L86	19	26	L93	
	L87	20	25	L92	
	Reset-in	21	24	L91	
	GND	22	23	L90	

Anhang - TINY-Tiger® – Nutzung der Pins

Leere Seite

TINY-Tiger® Modul E – Nutzung der Pins

Übersicht der I/O-Pin-Nutzung der wichtigsten Device-Treiber. Einige Funktionen sind fest an den entsprechenden Pin gebunden (PWM, PLSO1, PLSIN1), andere sind lediglich die Standardbelegung, lassen sich aber auf andere Pins verlegen (LCD, Strobe, Busy):

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
D0	L60	1	28	VCC	
D1	L61	2	27	L37	LCD1-RS
D2	L62	3	26	L36/An3	LCD1-E
D3	L63	4	25	L35/An2	INE
D4	L64	5	24	L34/An1	Dclk
D5	L65	6	23	L33/An0	Aclk
D6	L66	7	22	L41/PC	
D7	L67	8	21	L85	
LCD-WR	L80	9	20	Reset-in	
LCD-RD	L81	10	19	L94	Rx1
LCD-CE	L82	11	18	L93	Tx1
LCD-CD	L83	12	17	L92/L86	CT0/PLSO1
PLSIN1	L84	13	16	L91/L87	Rx0
	GND	14	15	L90	Tx0

Anhang - TINY-Tiger® Modul E – Nutzung der Pins

Leere Seite

Anhang - TINY-Tiger® Modul E – Nutzung der Pins

Freie Tabelle TINY-Tiger®-Modul E als Kopiervorlage. Projekt:

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	L60	1	28	VCC	
	L61	2	27	L37	
	L62	3	26	L36/An3	
	L63	4	25	L35/An2	
	L64	5	24	L34/An1	
	L65	6	23	L33/An0	
	L66	7	22	L41/PC	
	L67	8	21	L85	
	L80	9	20	Reset-in	
	L81	10	19	L94	
	L82	11	18	L93	
	L83	12	17	L92/L86	
	L84	13	16	L91/L87	
	GND	14	15	L90	

Anhang - TINY-Tiger® Modul E – Nutzung der Pins

Leere Seite

BASIC-Tiger®-CAN-Modul – Nutzung der Pins

Übersicht der I/O-Pin-Nutzung der wichtigsten Device-Treiber. Einige Funktionen sind fest an den entsprechenden Pin gebunden (PWM, PLSO1, PLSIN1), andere sind lediglich die Standardbelegung, lassen sich aber auf andere Pins verlegen (LCD, Strobe, Busy). **Die B-Reihe ist nicht aufgeführt**, da die meisten Pins unbelegt sind und die belegten Pins keine Variable Funktion zulassen:

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	res.	1	46	VCC	
D0	L60	2	45	Batt.	
D1	L61	3	44	AGND	
D2	L62	4	43	Vref	
D3	L63	5	42	An3	
D4	L64	6	41	An2	
D5	L65	7	40	An1	
D6	L66	8	39	An0	
D7	L67	9	38	Alarm	
Busy	L70	10	37	L41/PC	
Strobe	L71	11	36	L40	
PWM0	L72	12	35	L42	Beep
PWM1	L73	13	34	L37	LCD1-RS
LCD-WR	L80	14	33	L36	LCD1-E
LCD-RD	L81	15	32	L35	INE
LCD-CE	L82	16	31	L34	Dclk
LCD-CD	L83	17	30	L33	Aclk
PLSIN1	L84	18	29	L95	RTS
	L85	19	28	L94	Rx1
PLSO1	L86	20	27	L93	Tx1
		21	26	L92	CT0
	Reset-in	22	25	L91	Rx0
	GND	23	24	L90	Tx0

Anhang - BASIC-Tiger®-CAN-Modul – Nutzung der Pins

Leere Seite

Anhang - BASIC-Tiger®-CAN-Modul – Nutzung der Pins

Freie Tabelle BASIC-Tiger®-CAN-Modul als Kopiervorlage.

Die B-Reihe ist nicht aufgeführt, da die meisten Pins unbelegt sind und die belegten Pins keine Variable Funktion zulassen: Projekt:

Funktion	Pin-Bez.	Pin-Nr	Pin-Nr	Pin-Bez.	Funktion
	res.	1	46	VCC	
	L60	2	45	Batt.	
	L61	3	44	AGND	
	L62	4	43	Vref	
	L63	5	42	An3	
	L64	6	41	An2	
	L65	7	40	An1	
	L66	8	39	An0	
	L67	9	38	Alarm	
	L70	10	37	L41/PC	
	L71	11	36	L40	
	L72	12	35	L42	
	L73	13	34	L37	
	L80	14	33	L36	
	L81	15	32	L35	
	L82	16	31	L34	
	L83	17	30	L33	
	L84	18	29	L95	
	L85	19	28	L94	
	L86	20	27	L93	
		21	26	L92	
	Reset-in	22	25	L91	
	GND	23	24	L90	

Anhang -

8