
```
Mnemonic
Assembler        Maschinenbefehl    Wirkung /Bedeutung
      Ope-       Opcode/Binärcode            - Adressierungsart -
Mnem. randen FLAG  g  b yy xx  (g Gruppe, b Bit, yy,xx z.T. Opera./Reg.)
---- ------  ---- --- - -- --  ----------------------------
NOP              000 0 00 00      No OPeration
CMP           NZVC 000 0 00 01    CoMPare D0, D1 (D0-D1)
SWD              000 0 00 10      SWap D0, D1
SWM              000 0 00 11      SWap Memory  Adr(D0), Adr(D1)
                 arithmetische-/ Stack Befehle ------------
MUL           NZVC 000 0 01 00    MULtiplication D0 <- D0*D1
DIV           NZVC 000 0 01 01    DIVision D0 <- D0\D1 Rest SD0
PSA              000 0 01 10      PuSh All Stack <- A,D,SR Reg.
POA           NZVC 000 0 01 11    POp All SR,D,A Reg. <- Stack
                 Sprungbefehle/ Unterprogramm --- IC relativ ---
JSR              000 0 10 00      Jump SubRoutine Stack<-IC+1, A0=0: IC<-IC+A1 sonst IC<-A0+A1+ST
JIN +A           000 0 10 01      N=1: IC <- IC +A1       Jump If Negative
JIZ +A           000 0 10 10      Z=1: IC <- IC +A1       Jump If Zero
JMP +A           000 0 10 11           IC <- IC +A1       JuMP
                      ---         ----- Adr. mit Offset -----
RET              000 0 11 00      RETturn subroutine IC <-Stack
JIN IA           000 0 11 01      N=1: IC <- A0 +A1 +ST   Jump If Negative
JIZ IA           000 0 11 10      Z=1: IC <- A0 +A1 +ST   Jump If Zero
JMP IA           000 0 11 11           IC <- A0 +A1 +ST   JuMP
                                Lade Konstante --- Register direkt ---
LDC Rg        NZ 000 1 00 xx      LoaD Constant  Reg xx <- ## next Byte
##  CCCC CCCC                     ## Constant Byte
                          --- Register direkt ------------
JIN RG           000 1 01 xx      N=1: IC <- Reg xx       Jump If Negative
JIZ RG           000 1 10 xx      Z=1: IC <- Reg xx       Jump If Zero
JMP RG           000 1 11 xx           IC <- Reg xx       JuMP
                 Ein- Ausgabe / Stack Befehle -- Reg. direkt --
INP RG (IO)   NZ 001 0 00 xx      Reg xx          <- INPut
OUT RG (IO)      001 0 01 xx      OUTput          <- Reg xx
PSH RG        NZ 001 0 10 xx      PuSH  Stack     <- Reg xx
POP RG        NZ 001 0 11 xx      POP  Reg xx     <- Stack
                 Schattenreg./ Bittest /Nibble -- Reg. direkt --
SSR RG        NZ 001 1 00 xx      Set Shadow Reg xx -> Sxx
GSR RG        NZ 001 1 01 xx      Get Shadow Reg xx <- Sxx
BTS RG         Z 001 1 10 xx      BitTest Reg xx with Shadowreg.
SWN RG        NZ 001 1 11 xx      SWap Nibble Reg xx
                 Arithmet.-log. Befehle --- Register direkt ---
SHL RG        NZ C 010 0 00 xx    SHift Left   Reg xx
SHR RG         Z C 010 0 01 xx    SHift Right  Reg xx
ROL RG        NZ C 010 0 10 xx    ROtate Left  Reg xx
ROR RG        NZ C 010 0 11 xx    ROtate Right Reg xx
                                  ---
CLR RG         Z   010 1 00 xx    CLear Register Reg xx <- 0
INC RG        NZVC 010 1 01 xx    INCrement Reg xx
DEC RG        NZVC 010 1 10 xx    DECrement Reg xx
NOT RG        NZ   010 1 11 xx    Reg xx <- NOT Reg xx
                  ---   Reg.Reg./*1234) Transportbefehle mit SR/ SP
AND RG  RG    NZ   011 0 yy xx    Reg yy <- Reg yy AND Reg xx *1)
OR  RG  RG    NZ   011 1 yy xx    Reg yy <- Reg yy OR  Reg xx *2)
                 ------      --- Register direkt und indirekt ---
ADD RG RG     NZVC 100 0 yy xx    Reg yy <- Reg yy +   Reg xx *3)
ADD RG [RG]   NZVC 100 1 yy xx    Reg yy <- Reg yy + Adr(Reg xx)
                 ------
SUB RG RG     NZVC 101 0 yy xx    Reg yy <- Reg yy -   Reg xx *4)
SUB RG [RG]   NZVC 101 1 yy xx    Reg yy <- Reg yy - Adr(Reg xx)
                 --- Transportbefehle --- Register indirekt -----
MOV [RG] RG   NZ   110 0 yy xx    MOVe  Adr(Reg yy) <- Reg xx
MOV RG [RG]   NZ   110 1 yy xx    MOVe       Reg yy  <- Adr(Reg xx)
                 ------                  --- Register direkt ---
MOV RG  RG    NZ   111 0 yy xx    MOVe       Reg yy  <- Reg xx *5)
                      ---   Reg.         --- Register indirekt ---
MOV RG  IA    NZ   111 1 00 xx    MOVe       Reg xx  <- Adr(A0+A1+ST)
MOV IA  RG    NZ   111 1 01 xx    MOVe Adr(A0+A1+ST)<- Reg xx
LOD RG        NZ   111 1 10 xx    LOad Data Reg xx  <- Adr(h80+A1+ST)
                   - Speicherbefehle /Kopierbefehl /Stopp ------
STO (IO)       Z   111 1 11 00    STOre Datei <- Adr(h80+A1+ST)   D0 Byte, über D1 (Daten: IO)
RCL (IO)       ZV  111 1 11 01    ReCalL Adr(h80+A1+ST) <- Datei  über D1 (Daten: IO) /D0 Byte gelesen
CPY            ZV  111 1 11 10    CoPY Adr(A1)<-Adr(A0)  D0 Byte  über D1
STP        (Vorbef.) 111 1 11 11  StoP   Programm anhalten, Flags vom Vorbefehl
```

Register (0 bis 3 allg. Reg.)

Nr	Bez.	xx Bin	Art
0	D0	00	Datenregister
1	D1	01	Datenregister
2	A0	10	Adr.-Basisreg
3	A1	11	Adr.-Indexreg

SD0/SD1 Schattenreg. zu D0/D1
SA0/SA1 Schattenreg. zu A0/A1
IR Instruction Register
IC Instruction Counter
SR Status Register
SP Stack Pointer
ST Startadresse

A1 ist als Offset +A, IA vorzeichenbehaftet

Status Register SR

IOXYNZVC

Ein/ IO=00 Zeichen
Ausg. =01 dezimal
=10 hexadezimal
=11 binär
frei verfügbar XY
Flags N Negative
Z Zero
V oVerflow
C Carry

Adressierungsarten in Assembler

* Adr./Register fest

Wert im/in Register:
* Register direkt
Reg xx, Reg yy **.Rg.**

Wert im/in RAM (Adr. im Reg.)
* Register indirekt
Adr(Reg xx), yy **[Rg]**

Adr. mit Offset
* IC relativ
Adr.: IC + A1 **.+A.**

* IndexAdresse
Adr.: A0 + A1 + ST **.IA.**

* bei LOD/STO/RCL
Adr.: h80 + A1 + ST

A1 als Index mit Vorzeichen

|*12345) mit xx≠yy,
| bei xx=yy Code mit SR/SP:
|*1) MOV RG SR NZ Reg yy <- SR
|*2) MOV RG SP NZ Reg yy <- SP
|*3) MOV SR RG NZ SR <- Reg xx
|*4) MOV SP RG NZ SP <- Reg xx
|*5) MOV SR CC SR <- CC..
| CC setzt Datenart IO im SR

RAM-Adressen:

h00 Standard Startadresse
h80 Basisadr. Daten (down)
hF5 Basisadr. Stack (up)
hF6 Adresse Tastatureingabe
hF7 Adresse Display-Pixel
hF8 - hFF MGA Display 8x8

Läuft der IC unter h00/über hFF wird er zyklisch in den gültigen Bereich gesetzt, ebenso die Operanden-Adr(h80/A0 +A1+ST).
Keine Angabe bei FLAG: Flag wird 0 gesetzt. Daten IO: 00 Zeichen, 01 dezimal, 10 hexadezimal, 11 binär (ein Wert/Zeile).

Erklärung und Details

Die bei den Befehlen nicht angegebenen FLAG*'s* NZVC *werden stets* 0 *gesetzt. Das höherwertige Nibble* IOXY *des* SR *ist nicht betroffen.*

```
      Ope-       Opcode/Binärcode    Bedeutung  /Erklärung
Mnem. randen FLAG   g  b yy xx  (g Gruppe, b Bit, yy,xx z.T. Opera./Reg.)
-----------------------------------------------------------------------
             NZVC
```

NOP 000 0 00 00 No OPeration

Es wird keine Operation ausgeführt, Flags alle 0, der Befehlszähler IC wird um 1 erhöht.

CMP NZVC 000 0 00 01 CoMPare D0, D1 (D0-D1)

Vergleicht die Datenregister D0 und D1 durch Differenzbildung D0-D1 und setzt die entsprechenden Flags.
Die Register bleiben unverändert, die Differenz wird verworfen.

SWD 000 0 00 10 SWap D0, D1

Vertauscht den Inhalt der beiden Datenregister D0 und D1.

SWM 000 0 00 11 SWap Memory Adr(D0), Adr(D1)

Vertauscht den Inhalt der Speicherzellen deren Adressen in den Registern D0, D1 stehen.

MUL NZVC 000 0 01 00 MULtiplication D0 <- D0*D1

Multipliziert D0*D1 nach Booth, Ergebnis in D0. Auch bei Überlauf wird das niederwertige Byte zurückgegeben.

DIV NZVC 000 0 01 01 DIVision D0 <- D0\D1 Rest SD0

Ganzzahlige Division D0\D1, Ergebnis in D0, Rest in SD0. Vorzeichen Rest = Vorzeichen Dividend.

PSA 000 0 01 10 PuSh All Stack <- A,D,SR Reg.

Legt die Register A1, A0, D1, D0, SR in dieser Reihenfolge (SR oben) auf den Stack und verringert den SP um 5.

POA NZVC 000 0 01 11 POp All SR,D,A Reg. <- Stack

Setzt die Register in der Reihenfolge SR, D0, D1, A0, A1 mit den Werten aus dem Stack und erhöht den SP um 5.
Die Flags ergeben sich somit aus den entsprechenden Bits auf dem Stack.

JSR 000 0 10 00 Jump SubRoutine Stack<-IC+1, A0=0: IC<-IC+A1 sonst IC<-A0+A1+ST

Sprung zum Unterprogramm. Legt die Rücksprungadresse IC+1 auf den Stack.
Ist A0=0, so wird der IC mit A1 addiert in den Befehlszäler eingetragen und dort weitergearbeitet. Andernfalls wird der neue IC durch die Indexadresse A0+A1+ST bestimmt. A1 ist vorzeichenbehaftet.

JIN +A 000 0 10 01 N=1: IC <- IC +A1 Jump If Negative

Springt bei gesetztem N-Flag A1 Byte relativ zum Befehlszählerstand, wobei A1 vorzeichenbehaftet ist.

JIZ +A 000 0 10 10 Z=1: IC <- IC +A1 Jump If Zero

Springt bei gesetztem Z-Flag A1 Byte relativ zum Befehlszählerstand, wobei A1 vorzeichenbehaftet ist.

JMP +A 000 0 10 11 IC <- IC +A1 JuMP

Springt A1 Byte relativ zum Befehlszählerstand, wobei A1 vorzeichenbehaftet ist.

RET 000 0 11 00 RETturn subroutine IC <-Stack

Rücksprung aus dem Unterprogramm. Holt die Adresse des Befehlszählers vom Stack, um bei korrekter Verwaltung des Stacks beim Befehl nach JSR weiterzuarbeiten.

JIN IA 000 0 11 01 N=1: IC <- A0 +A1 +ST Jump If Negative

Springt bei gesetztem N-Flag zur berechneten Indexadresse A0 +A1 +ST, wobei A1 vorzeichenbehaftet ist.

JIZ IA 000 0 11 10 Z=1: IC <- A0 +A1 +ST Jump If Zero

Springt bei gesetztem Z-Flag zur berechneten Indexadresse A0 +A1 +ST, wobei A1 vorzeichenbehaftet ist.

JMP IA 000 0 11 11 IC <- A0 +A1 +ST JuMP

Springt zur berechneten Indexadresse A0 +A1 +ST, wobei A1 vorzeichenbehaftet ist.

LDC Rg NZ 000 1 00 xx LoaD Constant Reg xx <- ## next Byte

Lädt die im Byte nach dem LDC-Befehl mit dem Konstantenbefehl ## abgelegte Konstante in das Register.
Der Befehlszähler wird bei Ausführung des Befehls um 2 erhöht, ein falsch eingetragener Befehl damit übergangen.

CCCC CCCC ## Constant Byte

Konstantendefinition 1Byte (linkes und rechtes Nibble) für den LDC-Befehl. Muss unmittelbar diesem Befehl folgen.
Steht die Definition nicht nach LDC, wird das Byte als Maschinenbefehl interpretiert und entsprechend ausgeführt.

```
JIN RG            000 1 01 xx      N=1: IC <- Reg xx      Jump If Negative
```
Springt bei gesetztem N-Flag zur Adresse im Register.

```
JIZ RG            000 1 10 xx      Z=1: IC <- Reg xx      Jump If Zero
```
Springt bei gesetztem Z-Flag zur Adresse im Register.

```
JMP RG            000 1 11 xx           IC <- Reg xx      JuMP
```
Springt zur Adresse im Register.

```
INP RG (IO)   NZ    001 0 00 xx      Reg xx         <- INPut
```
Das Programm hält an und wartet auf die Eingabe eines Tastaturzeichens, einer Dezimalzahl -128 .. 127 oder einer Hexadezimalzahl. Die Eingabe wird binär und alternativ in den nicht eingegebenen Formen dargestellt. Der Binärwert wird im Register abgelegt und im RAM im Tastaturbyte `hF6` gespeichert. Die Eingabe erscheint im Ein-Ausgabeprotokoll in der durch die `IO` Bits des Statusregisters definierten Form.

```
OUT RG (IO)       001 0 01 xx      OUTput         <- Reg xx
```
Der Registerinhalt wird im Ein- Ausgabesystem binär, hexadezimal, dezimal und als Zeichen dargestellt und im Ein-Ausgabeprotokoll in der durch die `IO` Bits des Statusregisters definierten Form notiert.

```
PSH RG        NZ    001 0 10 xx      PuSH  Stack    <- Reg xx
```
Legt das Register oben auf den Stack und verringert den `SP` um 1.

```
POP RG        NZ    001 0 11 xx      POP  Reg xx    <- Stack
```
Setzt das Register auf den Wert aus dem Stack und erhöht den `SP` um 1.

```
SSR RG        NZ    001 1 00 xx      Set Shadow Reg xx -> Sxx  (SDx, SAx)
```
Setzt das Schattenregister zum Register auf dessen Wert.

```
GSR RG        NZ    001 1 01 xx      Get Shadow Reg xx <- Sxx  (SDx, SAx)
```
Setzt das Register auf den Wert seines Schattenregisters.

```
BTS RG         Z    001 1 10 xx      BitTest Reg xx with Shadowregister
```
Testet die entsprechenden Bits des Registers gegen die gesetzten Bits im Schattenregister. Sind alle im Schattenregister gesetzten Bits auch im Register gesetzt d.h. die maskierte Differenz ist 0, so wird das Zero-Flag gesetzt.
Das N-Flag wird gesetzt, wenn das Schattenregister negativ ist, aber das Register positiv.

```
SWN RG        NZ    001 1 11 xx      SWap Nibble Reg xx
```
Vertauscht das linke und rechte Nibble im Register. Entspricht dem Zifferntausch der Hexadezimalzahl.

```
SHL RG        NZ C  010 0 00 xx      SHift Left   Reg xx
```
Verschiebt die Bits im Register um eine Stelle nach links. Das rechte Bit wird 0.
Entspricht einer Multiplikation mit 2. Das herausgeschobene Bit gelangt ins Carry-Flag.

```
SHR RG         Z C  010 0 01 xx      SHift Right  Reg xx
```
Verschiebt die Bits im Register um eine Stelle nach rechts. Das linke Bit wird 0.
Entspricht der Ganzzahldivision \2. Das herausgeschobene Bit (entspricht Rest) gelangt ins Carry-Flag.

```
ROL RG        NZ C  010 0 10 xx      ROtate Left  Reg xx
```
Schiebt die Bits im Register eine Stelle nach links. Das linke Bit wird in das rechte Bit und auch in das C-Flag gesetzt.

```
ROR RG        NZ C  010 0 11 xx      ROtate Right Reg xx
```
Schiebt die Bits im Register eine Stelle nach rechts. Das rechte Bit wird in das linke Bit und auch in das C-Flag gesetzt.

```
CLR RG         Z    010 1 00 xx      CLear Register Reg xx <- 0
```
Löscht das Register, setzt also alle Bits des Registers auf 0. Das Z-Flag wird damit immer gesetzt.

```
INC RG        NZVC  010 1 01 xx      INCrement Reg xx
```
Erhöht den Registerinhalt um Eins.

```
DEC RG        NZVC  010 1 10 xx      DECrement Reg xx
```
Verringert den Registerinhalt um Eins.

```
NOT RG        NZ    010 1 11 xx      Reg xx <- NOT Reg xx
```
Negiert die einzelnen Bits im Register. (Wird anschließend INC ausgeführt, wird das Vorzeichen der Dezimalzahl gewechselt.)

Die Befehle `AND/OR/ADD/SUB/MOV RG RG`
existieren nicht für identische Register. Bei `AND/OR` *wäre das Ergebnis identisch zum Operanden, bei* `ADD` *entspricht es der Verdoppelung, die durch* `SHL` *besser abgedeckt wird. Bei* `SUB` *ist* `CLR` *ebenfalls effizienter. MOV macht keinen Sinn. Die dadurch freien Binärcodes werden für die Transportbefehle* `MOV` *mit dem Status Register und dem Stack Pointer genutzt und die Datenart* `IO` *kann gesetzt werden.*

`AND RG RG NZ 011 0 yy xx Reg yy <- Reg yy AND Reg xx (xx ≠ yy)`
Verknüpft die einzelnen Bits mit logischem UND. Ergebnis ist nur 1, wenn beide Bits 1 sind, sonst 0.

`MOV RG SR NZ 011 0 yy yy Reg yy <- SR`
Bewegt/Kopiert den Inhalt des Status Registers in das links stehende Register.

`OR RG RG NZ 011 1 yy xx Reg yy <- Reg yy OR Reg xx (xx ≠ yy)`
Verknüpft die Bits mit logischem ODER. Ergibt 0 nur, wenn beide Bits 0 sind, sonst 1.

`MOV RG SP NZ 011 1 yy yy Reg yy <- SP`
Bewegt/Kopiert den Inhalt des Stack Pointers in das links stehende Register.

`ADD RG RG NZVC 100 0 yy xx Reg yy <- Reg yy + Reg xx (xx ≠ yy)`
Addiert die beiden Register und legt das Ergebnis im links stehenden Register ab.

`MOV SR RG NZ 100 0 xx xx SR <- Reg xx`
Bewegt/Kopiert den Inhalt des rechts stehenden Registers in das Status Register und setzt damit die Flags.

`ADD RG [RG] NZVC 100 1 yy xx Reg yy <- Reg yy + Adr(Reg xx)`
Addiert das linke Register mit dem Byte, dessen Adresse im rechten Register steht. Ergebnis dann im linken Register.

`SUB RG RG NZVC 101 0 yy xx Reg yy <- Reg yy - Reg xx (xx ≠ yy)`
Subtrahiert das rechts stehende Register vom linken Register und legt das Ergebnis im linken Register ab.

`MOV SP RG NZ 101 0 xx xx SP <- Reg xx`
Bewegt/Kopiert den Inhalt des rechts stehenden Registers in den Stack Pointer und setzt ihn damit neu.

`SUB RG [RG] NZVC 101 1 yy xx Reg yy <- Reg yy - Adr(Reg xx)`
Subtrahiert vom linken Register das Byte, dessen Adresse im rechten Register steht. Ergebnis dann im linken Register.

`MOV [RG] RG NZ 110 0 yy xx MOVe Adr(Reg yy) <- Reg xx`
Bewegt/Kopiert den Inhalt des rechten Registers an die Speicheradresse im linken Register.

`MOV RG [RG] NZ 110 1 yy xx MOVe Reg yy <- Adr(Reg xx)` *)
Bewegt/Kopiert den Inhalt der Speicheraresse im rechten Register in das linke Register.

`MOV RG RG NZ 111 0 yy xx MOVe Reg yy <- Reg xx (xx ≠ yy)`
Bewegt/Kopiert das rechts stehende Register in das links stehende Register.

`MOV SR CC 111 0 CC CC MOVe SR <- CC.. (CC=CC)`
Die konstanten Bits CC werden in die linken zwei Bits des Status Register gesetzt. Damit wird die Datenart `IO` für die Ein- und Ausgabe festgelegt: `00` Zeichen, `01` dezimal, `10` hexadezimal, `11` binär.

`MOV RG IA NZ 111 1 00 xx MOVe Reg xx <- Adr(A0+A1+ST)`
Bewegt/Kopiert das Byte von der berechneten Indexadresse `A0+A1+ST` in das Register.
Die Adresse wird bei <0 oder >255 zyklisch bestimmt.

`MOV IA RG NZ 111 1 01 xx MOVe Adr(A0+A1+ST)<- Reg xx`
Bewegt/Kopiert den Registerwert in den RAM an die berechnete Indexadresse `A0+A1+ST`.
Die Adresse wird bei <0 oder >255 zyklisch bestimmt.

`LOD RG NZ 111 1 10 xx LOad Data Reg xx <- Adr(h80+A1+ST)`
Lädt/Kopiert das Datenbyte von der berechneten Indexadresse `h80+A1+ST` in das Register.
Basisadresse Daten ist `h80`, die Adresse wird bei <0 oder >255 zyklisch bestimmt, `A1` mit Vorzeichen.

STO (IO) Z 111 1 11 00 STOre Datei <- Adr(h80+A1+ST) D0 Byte, über D1 (Art: IO)

Schreibt D0 Byte (vorzeichenlos) ab Adresse h80+A1+ST in eine Textdatei. Die Adresse wird bei <0 oder >255 zyklisch bestimmt. Die Bytes werden dabei erst nach D1 gebracht. Pro Zeile wird ein Wert abgelegt. In Abhängigkeit von IO wird bei 00 Zeichen, 01 dezimal, 10 hexadezimal, 11 binär geschrieben. Das Z-Flag wird gesetzt, wenn keine Daten geschrieben wurden. Die Anzahl der tatsächlich geschriebenen Bytes steht anschließend vorzeichenlos in D0, das letzte Byte in D1.

RCL (IO) ZV 111 1 11 01 ReCalL Adr(h80+A1+ST) <- Datei über D1 (Art:IO) /D0 Byte gelesen

Liest aus einer Textdatei Daten in den RAM ab Adresse h80+A1+ST ein. Die Adresse wird bei <0 oder >255 zyklisch bestimmt. Die Bytes werden dabei zuerst nach D1 gebracht. Pro Zeile muss ein Wert vorhanden sein. In Abhängigkeit von IO wird bei 00 Zeichen, 01 dezimal, 10 hexadezimal, 11 binär als Typ in der Datei vorausgesetzt. Das Z-Flag wird gesetzt, wenn keine Daten gelesen wurden. Das V-Flag wird gesetzt, wenn mehr Daten in der Datei sind, als bis zum RAM-Ende gespeichert werden können. Die Anzahl der tatsächlich gelesenen Bytes steht anschließend vorzeichenlos in D0 , das letzte Byte in D1.

CPY ZV 111 1 11 10 CoPY Adr(A1)<-Adr(A0) D0 Byte über D1

Kopiert D0 Byte (vorzeichenlos) von RAM-Adresse in A0 beginnend in den Bereich ab RAM-Adresse in A1.
Das Z-Flag wird gesetzt, wenn keine Daten kopiert wurden.
Das V-Flag wird gesetzt, wenn mehr Daten kopiert werden sollen, als bis zum RAM-Ende gespeichert werden können.
Die Anzahl der tatsächlich kopierten Bytes steht anschließend vorzeichenlos in D0 , das letzte Byte in D1.

STP (Vorbef.) 111 1 11 11 StoP Programm anhalten, Flags vom Vorbefehl

Hält das Programm an. Die Flags werden vom voranstehenden Befehl übernommen. Damit kann STP neben den Breakpoints zu Testzwecken an beliebige Programmstellen eingeschoben werden.