Objekte 2 | Input/Output | C - Linux - Arduino - Raspberry

Inhaltsverzeichnis

Homepage

Homepage und Downloads: www.projektc.at

Allgemeine Beschreibungen

Die allgemeinen Dokumentationen findet man unter c/1_Dokus oder im Internet:

Vorwort:	www.projektc.at/programs/c/clar_vorwort.pdf	c/1_Dokus/clar_vorwort.pdf
Projekt c/ einrichten. Die ersten Schritte:	www.projektc.at/programs/c/clar_start.pdf	c/1_Dokus/clar_start.pdf
Projekthilfe und Projektmanager:	www.projektc.at/programs/c/clar_chelp.pdf	c/1_Dokus/clar_chelp.pdf
Ein neues C Programm erstellen:	www.projektc.at/programs/c/clar_projekt.pdf	c/1_Dokus/clar_projekt.pdf
Basisobjekte ohne Terminal In/Ouput:	www.projektc.at/programs/c/clar_objekte1.pdf	c/1_Dokus/clar_objekte1.pdf
Terminalsteuerung Box-Objekte für In/Ouput:	www.projektc.at/programs/c/clar_objekte2.pdf	c/1_Dokus/clar_objekte2.pdf
Datenspeicherung:	www.projektc.at/programs/c/clar_objekte3.pdf	c/1_Dokus/clar_objekte3.pdf

Dokumentation: C-Programme und Bibliotheken

- ▷ Die ersten Infos zu den C-Programmen oder Bibliotheken findet man in der Hilfedatei '1_read.me' im jeweiligen Programmordner.
- ▷ Für aufwendige Programme gibt es Beschreibungen im Format *.odt oder *.pdf im Ordner name/bin/_name/
- ▷ Die Dokumentation des Programmcodes befindet sich in den C-Headern der Programme oder Bibliotheken.
- ▷ Hilfe zu den fertigen Programmen liefert immer die Startoption '-h'.

Einstiege:

Programm chelp	Menügesteuerter Zugriff auf alle Dokus
Projekt c/ Einstieg und Übersicht	c/1_read.me
Header/Dokus für Bibliotheksfunktionen	c/lib/1_read.me
Testprogramme für Bibliotheksfunktionen	c/libtest/1_read.me

Library: iocon.a

Input-Output-CONsole für Linux Terminals.

Die Bibliothek bietet einfache und komplexe In- und Outputfunktionen und Dialoge für Consolen oder XTerminals.

Input

Die Input-Funktionen befinden sich in Modulen der Library iocon.a.

Beschreibungen chelp:	Stichworte | Lib iocon.h	: Eingabefunktionen
Modul:	c/lib/include/iocon.h	
Testprogramme:	c/libtest/testlibiocon	Einfache IO-Funktionen

Eingabefunktionen

```
// ============================================================================
// ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
// |
// | Library  : libiocon.a | Input-Output-CONsole für Consolen oder Terminals.
// | Modul    : iocon.h
// | Test/Doku: testlibiocon, testlibiocon_select, testkeys
// |
// | Library: Input-Output-CONsole für Linux Consolen oder virtuelle Terminals.
```

Alle Eingabefunktionen basieren auf der privaten Basisfunktion readKbd(). Die gelesenen Bytefolgen werden in eine uint16_t Taste codiert.

```
// ============================================================================
// Keyboard Tastenfunktionen  ||||||||||||||||||||||||||||||||||||||||||||||||
// ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
//
// Keyboard Basisfunktionen liefern einen Tastencode vom Typ (uint16_t).
// Alle weiteren Eingabefunktionen verwenden diese Basisfunktionen.
//
// Jeder Input-Bytefolge mit 1 bis 6 Bytes ist ein Tastencodes (uint16_t)
// zugeordnet.
//
// Die Tastencodes sind nachfolgend unter 'Tastencodes' definiert. //
```

Tastencodes

Die Tastencodes sind in der Tabelle Tastencodes zusammengefasst.

Alle 1 Byte Codes sind die ASCII-Codes.

```
Beispiel: uint16_t Taste='a';
```

Für Steuerzeichen sind Bezeichner definiert:

```
// ..............................................................
// Keyboard-Tasten mit 1 Byte liefern ASCII-Codes
//
// Nicht druckbare Steuerzeichen: 0x0000 bis 0x001f //
#define taste_nil      0x0000
#define taste_ctl_a    0x0001
#define taste_ctl_b    0x0002
...
```

Tasten abfragen

- Die Programme befinden sich normalerweise in einer der folgenden Funktionen. Beim Warten wird das System nicht blockiert!.
- Mit den Select()-Funktionen können eigene Eventhandler realisiert werden.

Die Verfügbarkeit einer Taste prüfen:

```
bool peekTaste(long nsec);
  // Verfügbarkeit von Daten auf stdin und Select-Inputs prüfen.
  // Daten von stdin nicht lesen. Für Select-Inputs Handlerfunktionen aufrufen.
  // nsec    : Timeout von 0 bis 999.999.999 nanosec. -1 kein Timeout
  // Rückgabe: true, Taste auf stdin verfügbar. Fehler im Err-Objekt.

bool peekTasteSec(long sec); // Funktioniert wie peekTaste(). Timeout in Sekunden.
```

Tasten lesen:

```
uint16_t chkTaste(long nsec);
  // stdin nicht blockierend abfragen und Select-Inputs prüfen.
  // Für Select-Inputs Handlerfunktionen aufrufen.
  // nsec    : Wartezeit 0 bis 999.999.999 nanosec. -1 kein Timeout
  // Rückgabe: Ein Tastencode oder taste_nil. Fehler im Err-Objekt.

uint16_t chkTasteSec(long sec); // Funktioniert wie chkTaste(). Timeout in Sekunden.
```

Auf Taste warten und lesen:

```
#define BEFEHL            "Befehl ? "               // Befehlsprompt
#define RETURNEsc         "RETURN | ESC ? "         // Befehlsprompt
#define BEFEHLEsc         "Befehl | ESC ? "         // Befehlsprompt
#define BEFEHLRetEsc      "Befehl | RETURN | ESC ?"  // Befehlsprompt
#define LESSEndText       "Ende mit q\n"            // Befehlsprompt für less

uint16_t getTaste(const char* Prompt);
  // Auf Tastencode von stdin warten und Select-Inputs prüfen.
  // Für Select-Inputs Handlerfunktionen aufrufen.
  // if (Prompt!=NULL) Prompt-Text anzeigen
  // Rückgabe: Ein Tastencode oder taste_nil . ESC kann danach mit readESC() abgefragt werden

uint16_t low(uint16_t Taste);  // Taste nach lowercase
```

Auf Taste warten und nicht lesen:

```
void WeiterMitTaste();
  // getTaste mit Prompt "Weiter mit Taste" aufrufen.
  // Für Select-Inputs Handlerfunktionen aufrufen.
  // Bei Umleitung von stdout mit lessPrint() überspringen!

void     WeiterMitTasteCon();                                    // WeiterMitTaste in X-Terminals überspringen.
uint16_t WeiterMitTasteSec(uint16_t WarteSec, const char *Prompt);  // Prompt: NULL oder Promptext anzeigen.
void     WeiterInSec(uint16_t WarteSec, const char *Prompt);        // Prompt: NULL oder Promptext anzeigen.
                                                                    // WarteSec: Wartezeit. Abbruch mit Taste nicht möglich.
```

Ja/Nein Abfrage

```
bool isAntwortJa(const char *Frage);
  // Prompt 'Frage' ? j/n anzeigen und mit getTaste() abfragen.
  // Für Select-Inputs Handlerfunktionen aufrufen.
  // Rückgabe: true bei Antwort 'j' oder 'J'
  // Nicht in Umleitungen von stdout in Datei verwenden!
```

Taste setzen. Zum Beispiel nächster Menubefehl.

```
void setTaste(uint16_t TastenCode);
  // Tastencode für die nächste Keyboardabfrage setzen.
  // Beispiel: Taste für den nächsten Menupunkt.
```

System

```
void KbdRestore();
  // Option: Keyboardfunktionen zurücksetzen.
  // Wird automatisch als Exitfunktion gesetzt.
  // - tty Einstellungen
  // - Selectobjekt freigeben

void KbdSetAutorepeat(bool on);
  // Autorepeat on/off im Terminalfenster

uint16_t readKbdRaw(char **InputBuf );
  // Testfunktion für readKbd(). SizeOf(InputBuf) > MAXBYTES! Siehe: gettaste.c
  // Für alle read-Funktionen wird die zentrale readKbd() Funktion verwendet.
  // Rückgabe: Tastencode von iocon zum Tastendruck. Wie von readKbd() .
  //           Zusätzlich: Die Bytes vom Keyboard als String im InputBuf

void KbdFlush();
  // Keyboardpuffer löschen
```

Eingabefunktionen

Testprogramme: c/libtest/testkeys Test ReadLn
c/libtest/testlibiocon 5 Test: readUInt16(), readInt16(), readInt32(), readDouble()

Für die verschiedenen Datentypen gibt es Eingabefunktionen aus der readXXX(...) Familie mit einem Eingabefeld und dem üblichen Komfort.

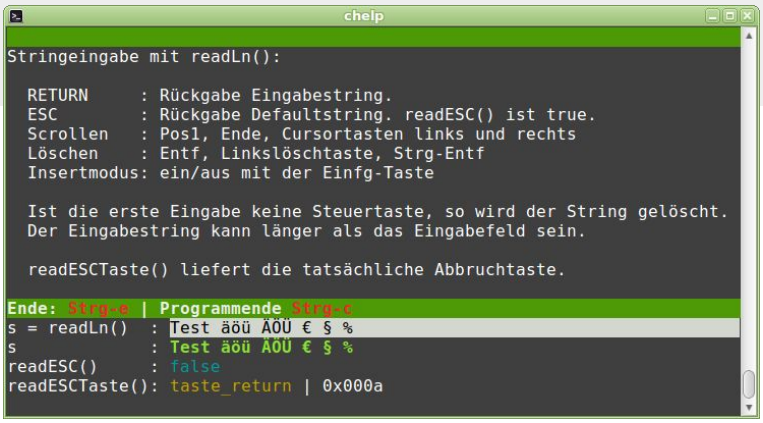
```
// ---------------------------------------------------------------------------
// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
//
// Keyboard Read-Funktionen für komfortable Eingaben:   readXXX()
//
// Eingaben erfolgen in einem Eingabefeld mit der üblicher Cursorsteuerung:
//    - Der Eingabestring kann länger als das Eingabefeld sein.
//    - Ist die erste Eingabe keine Cursortaste, so wird der String gelöscht.
//
//    RETURN     : Rückgabe Eingabestring
//    ESC        : Rückgabe Defaultstring. readESC() ist true
//    Scrollen   : Pos1, Ende, Cursortasten links und rechts
//    Löschen    : Entf, Linkslöschtaste, Strg-Entf
//    Insertmodus: ein/aus mit der Einfg-Taste
//    Sonst      : wie ESC
// ...........................................................................
// Rückgaben:
//    bei ESC    : Rückgabe ist der Default-Wert. readESC() liefert true.
//    bei RETURN : Rückgabe ist der Eingabewert. Werte im vorgegeben Bereich.
//                 Die Funktion readESC() liefert false.
//
// Alle Read-Funktionen verwenden nur die Keyboard Basisfunktionen!
```

Nach der Eingabe kann die Endtaste geprüft werden:

```
bool     readESC();      // ESC-Abfrage nach readXXX() oder getTaste().
                         // true: Letztes readXXX() wurde mit ESC abgebrochen

uint16_t readESCTaste(); // Abfrage nach readXXX() oder getTaste().
                         // Rückgabe der tatsächlichen Endtaste von readXXX().
```

Stringeingabe

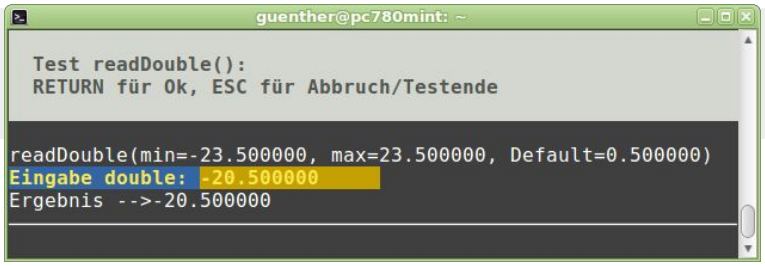
```
// Stringeingabe .................................................
const char   *readLn     // String einlesen. Länge beliebig
 (const char *Prompt,    // Anzeige-Prompt
  uint16_t    Breite,    // Breite des Eingabefelds
  const char *Default,   // Defaultwert bei ESC
  const char *Farbe      // Farbe der Anzeige. NULL FarbeBlackGray
 );
 // Rückgabe ist ein temporärer String in tmpStr.
 //
 // Ende mit RETURN:
 //   Rückgabe ist der eingegebene Wert.
 //   "" liefert NULL und readESC()==false
 // Ende mit ESC   :
 //   Rückgabe ist Defaultwert und readESC()==true
```



Zahleneingaben

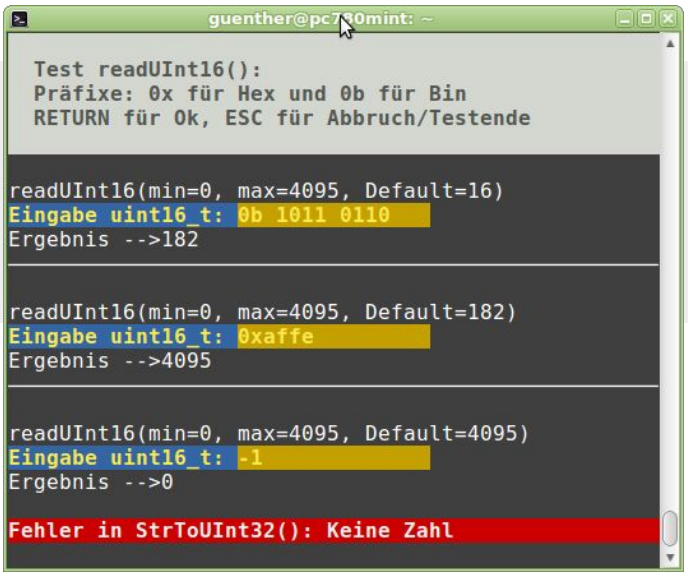
```
// Zahleneingaben ................................................
//
// Ende mit RETURN: Rückgabe ist der eingegebene Wert.
//                  Im Fehlerfall der Defaultwert. Fehler im Err-Objekt
//                  Hexzahlen mit Präfix 0x: z.B. 0xf3
//                  Für Binärwerte Funktion readUInt16() verwenden.
// Ende mit ESC   : Rückgabe Defaultwert und readESC()==true
```

```
double        readDouble  // Double einlesen
 (const char *Prompt,     // Anzeige-Prompt
  double      min,        // Minimum
  double      max,        // Maximum
  double      Default     // Defaultwert bei ESC
 );
```



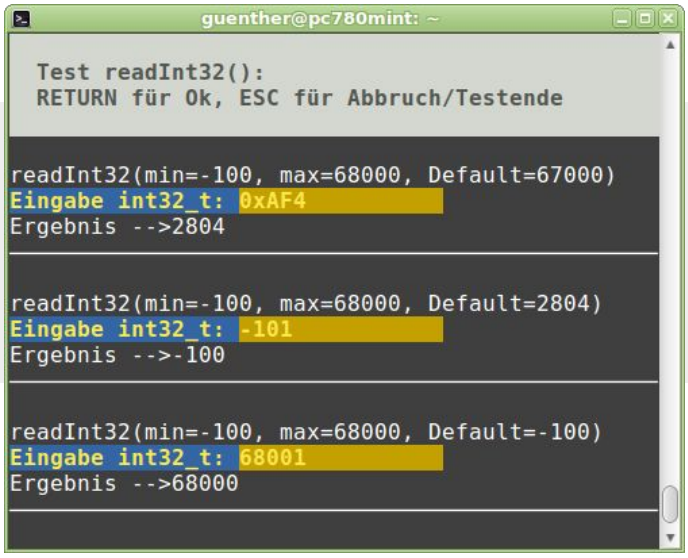
readUint16() liest auch:
Binäreingaben mit und ohne Blanks z.B. 0b 1011 1111
Hexeingaben z.B. 0xFA12

```
uint16_t      readUInt16  // Unsigned Integer oder
                          // Binär 0b oder hex 0x
 (const char *Prompt,     // Anzeige-Prompt
  uint16_t    min,        // Minimum
  uint16_t    max,        // Maximum
  uint16_t    Default     // Defaultwert bei ESC
 );

int16_t       readInt16   // Integer einlesen
 (const char *Prompt,     // Anzeige-Prompt
  int16_t     min,        // Minimum
  int16_t     max,        // Maximum, INT16_MAX=32767
  int16_t     Default     // Defaultwert bei ESC
 );
```



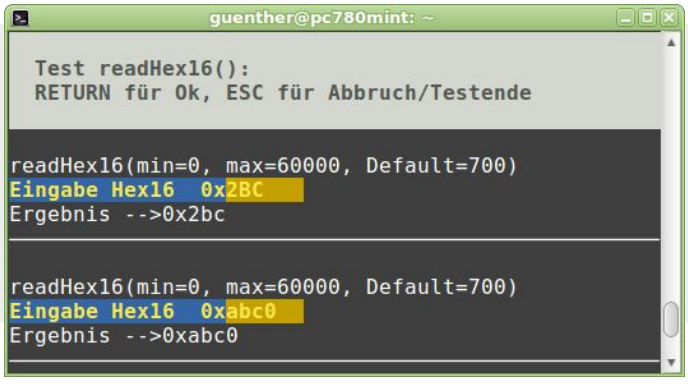
readInt32() und readUInt32() lesen auch
Hexeingaben z.B. 0xFA12

```
uint32_t      readUInt32  // Unsigned Integer einlesen
 (const char *Prompt,     // Anzeige-Prompt
  uint32_t    min,        // Minimum
  uint32_t    max,        // Maximum INT32_MAX
  uint32_t    Default     // Defaultwert bei ESC
 );

int32_t       readInt32   // Integer einlesen
 (const char *Prompt,     // Anzeige-Prompt
  int32_t     min,        // Minimum
  int32_t     max,        // Maximum, INT32_MAX=2147483647
  int32_t     Default     // Defaultwert bei ESC
 );
```



Anzeige und Eingabe in Hex.

```
uint16_t      readHex16   // Unsigned Integer.
 (const char *Prompt,     // Anzeige-Prompt
  uint16_t    min,        // Minimum
  uint16_t    max,        // Maximum
  uint16_t    Default,    // Defaultwert bei ESC
  const char *Farbe       // NULL oder Farbe
);

uint32_t      readHex32   // Unsigned Integer. Anzeige und
Eingabe in Hex
 (const char *Prompt,     // Anzeige-Prompt
  uint32_t    min,        // Minimum
  uint32_t    max,        // Maximum
  uint32_t    Default,    // Defaultwert bei ESC
  const char *Farbe       // NULL oder Farbe
);
```



Output

Terminal

Beschreibungen chelp: Stichworte |Lib iocon.h: Ausgabefunktionen, Farben
|h| Terminalfunktionen |iocon.h| Terminal
Modul: c/lib/include/iocon.h
Testprogramme: c/libtest/testlibiocon Einfache IO-Funktionen

Für die Text-Ausgaben in Consolen oder Xterminals stehen drei Möglichkeiten gleichwertig zur Verfügung:

- Die fortlaufende Ausgabe mit printf(...) ohne clipping
- Ausgabe von Text-Balken mit printBlock() mit clipping
- Box-Ausgabe in absolut positionierten Rechtecken mit clipping

Terminalsteuerung

Für die Gestaltung des Bildschirmlayouts stehen folgende Steuerfunktionen zur Verfügung.
Die Initialisierung erfolgt über einen einmaligen Aufruf von:

```
clrScr();      // Clear Screen und die aktuelle Zeilen- und Spaltenanzahl bestimmen. setMaxYX()
```
oder
```
setMaxYX();    // Die aktuelle Zeilen- und Spaltenanzahl bestimmen. setMaxYX()
```

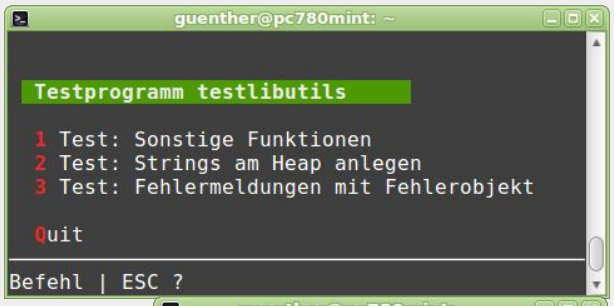
Eine Bildschirmposition wird mit Zeile y und Spalte x festgelegt.
Die sichtbaren Koordinaten (y,x) laufen von (1,1) links/oben bis (getMaxY(), getMaxX()) rechts/unten.

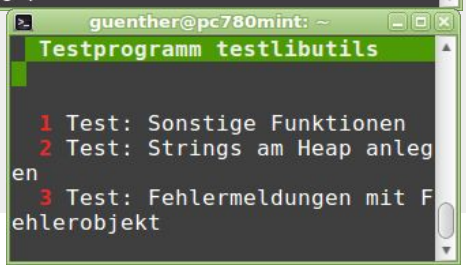
```
// ============================================================================
// Terminal Steuerfunktionen ||||||||||||||||||||||||||||||||||||||||||||||||||
//
// Beim Programmstart oder vor dem Neuzeichnen sollte immer clrScr() oder
// setMaxYX() aufgerufen werden. Damit wird die aktuelle Fenstergröße ermittelt.
//
void     clrScr();      // Clear Screen und setMaxYX().
void     clrScrColor(const char *Farbe); // Clear Screen mit Farbe,setMaxYX().
bool     setMaxYX();    // Die aktuelle Zeilen- und Spaltenanzahl für
  //                       die getMax-Funktionen bestimmen und setzen.
  //                       false: Werte nicht gelesen. Ersatzwerte gesetzt

uint16_t getMaxY();     // aktuelle Zeilenanzahl im Terminal
uint16_t getMaxX();     // aktuelle Spaltenanzahl im Terminal
  //                       Einmaliger Aufruf von clrScr() notwendig

void     clrEos();      // Clear End of Screen
void     clrEol();      // Clear End of Line
void     hideCursor();  // Cursor verstecken
void     showCursor();  // Cursor zeigen
void     moveLeft();    // Cursor left
void     moveRight();   // Cursoe right

void     gotoYX         // Cursorposition setzen
 ( uint16_t y,          // Zeile  y von 1 bis getMaxY()
   uint16_t x           // Spalte x von 1 bis getMaxX()
 );

uint16_t getLastY();    // Schnelle Funktion. Wird von clrScr(), gotoYX(),
 //                       und printBox() gesetzt.

void     getYX          // Langsame Funktion für die aktuelle Cursorposition
 ( uint16_t *y,         // Zeile
   uint16_t *x          // Spalte
 );
uint16_t getY();        // Langsame Funktion für die aktuelle Cursorposition
//                      // verwendet getYX()

void     savePos();     // Aktuelle Cursoposition und Attribute speichern
void     restPos();     // restore gespeicherte Cursoposition

char    *TermName();    // Terminalname abfragen
void     restore();     // Option: Terminalfenster zurücksetzen. Erfolgt automatisch!

void     setXtermSize(uint16_t MaxY, uint16_t MaxX); // Size vom Xterm setzen

// ============================================================================
```

printf

Ein Beispiel für die fortlaufende Ausgabe mit printf(...).

```
015 ▷ void runTest()
016 ▷ { while (true)
017 ▷   { clrScr();
018 ▷     gotoYX(3,2);
019 ▷
020 ▷     printf(FCap4 " Testprogramm testlibutils     \n" FN);
021 ▷
022 ▷     printf
023 ▷     ("\n"
024 ▷      "  "FT"1"FN" Test: Sonstige Funktionen\n"
025 ▷      "  "FT"2"FN" Test: Strings am Heap anlegen\n"
026 ▷      "  "FT"3"FN" Test: Fehlermeldungen mit Fehlerobjekt\n"
027 ▷      "\n"
028 ▷      "  "FT"Q"FN"uit\n"
029 ▷     );
030 ▷     printLine(0);
031 ▷
032 ▷     switch(low(getTaste( BEFEHLEsc )))
033 ▷     { case '1': Test1(); break;
034 ▷       case '2': Test2(); break;
035 ▷       case '3': Test3(); break;
036 ▷
037 ▷       case taste_esc:
038 ▷       case 'q': printLn(); exit(EXIT_SUCCESS);
039 ▷
040 ▷       default: ;
041 ▷     }
042 ▷   }
043 ▷ }
```





Bemerkungen:

Zeile 017: clrScr() // Bildsschirm löschen und gotoYX() initialisieren
Zeile 018: gotoYX(3,2) // Schreibposition auf Zeile 3, Spalte2

Die Farbausgabe wird im Terminal mit ANSI ESC-Farbstrings gesteuert.
Farben und Farbkonstanten können mit chelp | Farben und Tasten abgefragt werden. Siehe Farben und Tasten.

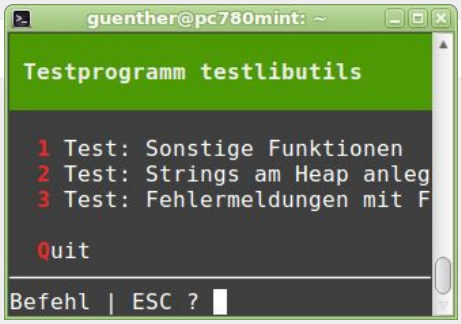
Zeile 020: FCap4 Farbe Überschrift 4 FN Farbe normal
Zeile 024: FT Farbe Funktionstaste FN Farbe normal

Zeile 030: printLine(0); // Trennlinie
Zeile 032: low(...); // Taste in lowercase umwandeln
Zeile 030: printLn(); // Leerzeile

printBlock

Mit printBlock() können farbige Textzeilen in Terminalbreite fortlaufend angezeigt werden.
Lange Textzeilen werden am linken Rand abgeschnitten. Das Layout bleibt erhalten.

```
void printBlock(const char *Text, const char *FarbStr);
  // Farbige Textbalken an der aktuellen Cursorposition.
```



Beispiel runTest() mit printBlock():

```
019 ▷     ...
020 ▷     printBlock
021 ▷     ("\n"
022 ▷      " Testprogramm testlibutils\n"
023 ▷      "\n", FCap4
024 ▷     );
025 ▷
026 ▷     printBlock
027 ▷     ("\n"
028 ▷      "  "FK"1 Test: Sonstige Funktionen\n"
029 ▷      "  "FK"2 Test: Strings am Heap anlegen\n"
030 ▷      "  "FK"3 Test: Fehlermeldungen mit Fehlerobjekt\n"
031 ▷      "\n"
032 ▷      "  "FK"Quit\n"
033 ▷      , FN
034 ▷     );
035 ▷     ...
```

Änderungen gegenüber Beispiel runTest():

Zeile 020: printf(...) wurde durch printBlock (",,,", GrundFarbe) ersetzt.
Die GrundFarbe des Blocks kann im Text beliebig geändert werden.

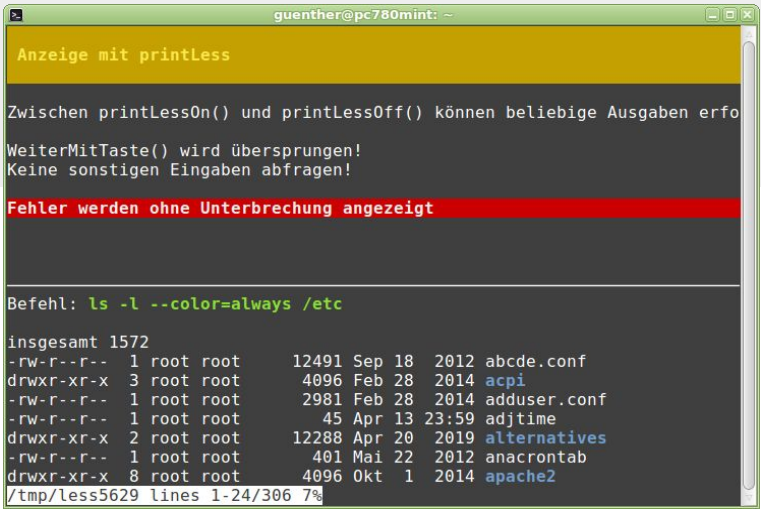
Zeile 028: FK (Farbe Key) ändert nur die Farbe des folgenden Zeichens in printBlock().

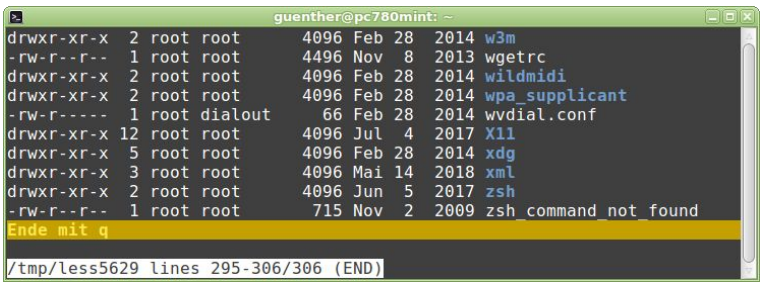
printLess

Längere Terminalausgaben können mit printLess() in den Pager less umgeleitet werden.

Beispiel:

```
017 ▷
018 ▷   printLessOn("\n" " Anzeige mit printLess\n" "\n", NULL,0);
019 ▷
020 ▷     printLn();
021 ▷     printf("Zwischen printLessOn() und printLessOff() können beliebige Ausgaben erfolgen\n");
022 ▷     printLn();
023 ▷     printf("WeiterMitTaste() wird übersprungen!\n");
024 ▷     printf("Keine sonstigen Eingaben abfragen!\n");
025 ▷     ErrPrint(Err,"Fehler werden ohne Unterbrechung angezeigt",true);
026 ▷     printLn();
027 ▷     printLine(0);
028 ▷     WeiterMitTaste();
029 ▷
030 ▷     char *Befehl="ls -l --color=always /etc";
031 ▷
032 ▷     printf("Befehl: "FG"%s\n\n"FN, Befehl);
033 ▷
034 ▷     callSystem(Befehl);
035 ▷
036 ▷   printLessOff(NULL, NULL, NULL, 100);
037 ▷
```





Zeile 018: printLessOn() startet die Umleitung. Die Ausgaben werden unter /tmp gespeichert.

Zeile 021: Lange Zeilen. Betrachten mit den Pfeiltasten links/rechts

Zeile 023: WeiterMitTaste() wird übersprungen.

Zeile 025: Err-Ausgaben halten nicht an.

Zeile 028: WeiterMitTaste() wird übersprungen.

Zeile 034: Befehl ausführen und anzeigen.

Zeile 036: printLessOff() beendet die Umleitung. Ausgabendatei mit less anzeigen.
100: Ausgabendatei nicht löschen.
0: Ausgabendatei autom. löschen.

In den meisten Fällen sind die Defaulteinstellungen von printLess() passend.

```
bool printLessOn(const char *Caption, const char *FarbStr, uint16_t Debug);
  // Ausgabe von stdout in Datei getTmpPath("less") für die Anzeige mit
  // less umleiten. printLessOff() beendet die Umleitung und zeigt
  // die Umleitungsdatei mit less an.
  //
  // Caption: NULL oder Blockzeilen mit Überschrift.
  // FarbStr: NULL für FCap1. Sonst Farbstring.
  // Debug  : >0    Debuginfos ausgeben
  //          100   Keine Umleitung, nur Caption ausgeben
```

```
bool printLessOff(const char *LessOpt, const char *EndCaption, const char *FarbStr, uint16_t Debug);
  // Nach der Ausgabe des Anzeigeblocks EndCaption wird die Umleitung beendet und
  // stdout wieder aktiviert.
  //
  // LessOpt   : String mit less Options. Siehe man less.
  //             NULL  Default für "-MRS"
  //             -M    ausfühliches less Prompt
  //             -R    Farbsteuerung mit ANSI "color" escape sequences
  //             -N    Zeilennummern anzeigen
  //             -S    Zeilen am Bildrand abschneiden. links/rechts scrollen mit Tasten
  //
  // EndCaption: Null oder Anzeigeblock mit FarbStr ausgeben.
  // FarbStr   : NULL für FCap1. Sonst Farbstring.
  // Debug     : =0          Umleitungsdatei löschen.
  //             0<Debug<100  Debuginfos (Debug>0) ausgeben
  //             =100         Umleitungsdatei nicht löschen.
  //             =100+Debug   Umleitungsdatei nicht löschen und
  //                          Debuginfos (Debug>0) ausgeben.
```

printBox

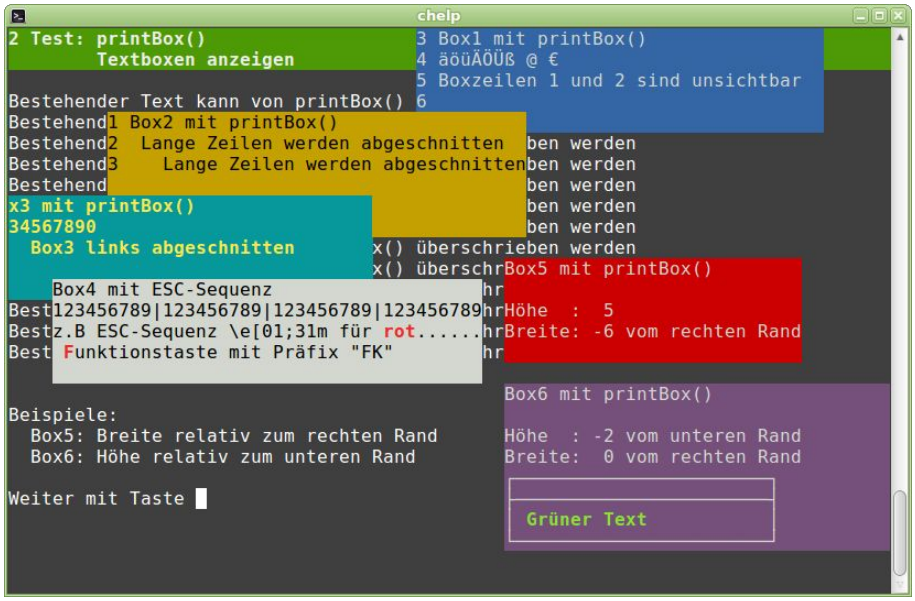
Die Ausgabefunktion printBox() ist die Basis für alle Ausgabeobjekte.

Die Funktion kann rechteckige Textblöcke an beliebigen Bildschirmpositionen ausgeben. Die Rechtecke werden an den Bildschirmrändern abgeschnitten. Der Text wird zeilenweise ausgegeben und an den Rändern der Box geclippt. UTF8 Zeichen und ESC-Sequenzen sind erlaubt.

```
void printBox(
  int16_t     y,     // 1. Zeile, negative Werte möglich
  int16_t     x,     // 1. Spalte, negative Werte möglich
  int16_t     h,     // Höhe,   <1 vom Bildschirmende oder Zeilenanz
  #define             ZeileAnz 0x7FFF   // // Höhe ist Zeilenanzahl
  int16_t     b,     // Breite, <1 vom rechten Rand
  const char *Farbe, // NULL oder ESC-Sequenz
  const char *Text   // Text NULL oder 1 bis n Zeilen mit \n getrennt
  );
  // UTF8 Text zeilenweise in einer rechteckigen farbigen Box anzeigen.
  // Die Box wird am Terminalfenster geklippt. Der Text wird an der Box abgeschnitten.
  // ESC-Sequenzen im Text:
  //   Farben: \e[...m oder \e[m werden ausgegeben
  //   \eKm   wird ausgefiltert
  //   \etX   wird als Funktionstaste X in rot ausgegeben
```

Beschreibungen chelp: Stichworte | Lib iocon.h : Ausgabefunktionen, Farben
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testlibiocon 2 Test: Textboxen mit printBox() anzeigen

---




```
336 ▷   printBox
337 ▷   ( getLastY()-1, -1, HISZEILEAnz, 35,
338 ▷     FarbeYellowCyanB,
339 ▷     "Box3 mit printBox()\n"
340 ▷     "1234567890\n"
341 ▷     "    Box3 links abgeschnitten\n"
342 ▷     "\n"
343 ▷   );
```

Zeile 337: HISZEILEAnz : Höhe der Box entspricht der Zeilenanzahl

```
345 ▷   printBox
346 ▷   ( 13, 5, 5, 39,
347 ▷     FarbeBlackGray,
348 ▷     "Box4 mit ESC-Sequenz\n"
349 ▷     "123456789|123456789|123456789|123456789|123456789\n"
350 ▷     "z.B ESC-Sequenz \\e[01;31m für "FR"rot"FN".....\n"
351 ▷     " "FK"Funktionstaste mit Präfix \"FK\"\n"
352 ▷   );
```

Zeile 351: Funktionstaste FK.

```
370 ▷   printBox
371 ▷   ( 18, 46, -2,  0,
372 ▷     FarbeWhiteViolet,
373 ▷     "Box6 mit printBox()\n"
374 ▷     "\n"
375 ▷     "Höhe  : -2 vom unteren Rand\n"
376 ▷     "Breite:  0 vom rechten Rand\n"
377 ▷     "┌──────────────────────┐\n"
378 ▷     "│                      │\n"
379 ▷     "│  "FG"Grüner Text"FN"           │\n"
380 ▷     "└──────────────────────┘\n"
381 ▷   );
```

Zeile 371: h=-2 Höhe -2 vom unteren Rand. b=0 Breite 0 vom rechten Rand.

Box-Dialoge

Komplexe Ausgabedialoge bestehen aus zusammengesetzten Box-Objekten.
Eine Box beschreibt einen absolut positionierten rechteckigen Bildschirmbereich.
Mehrere Boxen können mit BoxInsert() zu einem Dialog zusammengefügt werden.
Es folgen nun wichtige vordefinierte Dialoge.

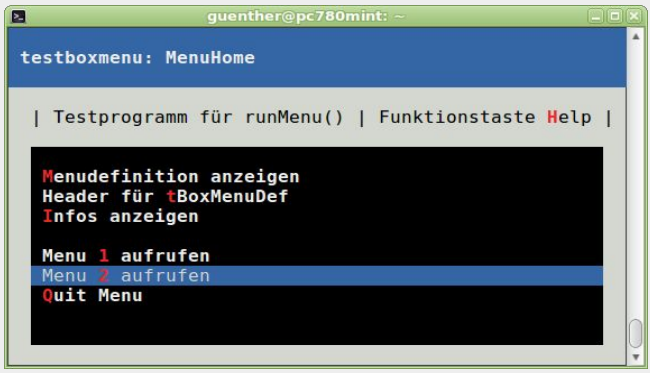
Menudialoge

Beschreibungen chelp: Stichworte | Lib boxmenu.h: Menus
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testboxmenu Dialogelement Menu

Beispiel: Testprogramm c/libtest/testboxmenu für Menuaufrufe

Die Menudefinition:

```
Code: MenuHome
021 ▷ tBoxMenuDef Menu1;
022 ▷ tBoxMenuDef Menu2;
023 ▷
024 ▷ tBoxMenuDef MenuHome=
025 ▷ { .y=1, .x=1, .h=14, .b=0,
026 ▷
027 ▷   .Titel=
028 ▷   "\n"
029 ▷   " "TESTBoxMenu": MenuHome\n",
030 ▷
031 ▷   .Info=
032 ▷   "\n"
033 ▷   "  | Testprogramm für runMenu() | Funktionstaste "FK"Help |"
034 ▷   "\n",

035 ▷   .FarbeZeile =FarbeWhiteBlackB,
036 ▷   .FarbeCursor=FarbeWhiteBlue,
037 ▷
038 ▷   .I=1,
039 ▷   .Items=(tBoxMenuItem [])
040 ▷   { {.Typ=Leer},
041 ▷     {.Typ=Run,  .Txt=" "FK"Menudefinition anzeigen", .Ptr=printMenuDefAktuell, .Key='m'},
042 ▷     {.Typ=Run,  .Txt=" Header für "FK"tBoxMenuDef",  .Ptr=printTBoxMenuDef,    .Key='t'},
043 ▷     {.Typ=Run,  .Txt=" "FK"Infos anzeigen",          .Ptr=runInfos,            .Key='i'},
044 ▷
045 ▷     {.Typ=Leer},
046 ▷     {.Typ=Menu, .Txt=" Menu "FK"1 aufrufen", .Ptr=&Menu1,    .Key='1'},
047 ▷     {.Typ=Menu, .Txt=" Menu "FK"2 aufrufen", .Ptr=&Menu2,    .Key='2'},
048 ▷     {.Typ=Menu, .Txt=" "FK"Quit Menu", .Ptr=NULL },   // immer NULL!
049 ▷     { /*Menuende*/ }
050 ▷   }
051 ▷ };
```



Bemerkungen zur Menudefinition:

Zeile 021: Forward Deklaration für Menu1.

Zeile 022: Forward Deklaration für Menu2.

Zeile 024: Definition der Menustruktur MenuHome. Bei Definitionen ausserhalb von Funktionen werden alle unbelegten Felder automatisch mit 0/NULL belegt.

Zeile 025: Die Box-Koordinaten: Ecke (1,1) links/oben, Höhe h=14, Breite b=0 .
Höhe <=0 Abstand vom unteren Rand. Breite <=0 Abstand vom rechten Rand.

Zeile 027: Die blaue Titelbox. Textkonstante TESTBoxMenu

Zeile 031: Die graue Infobox mit Funktionstaste Help.

Zeile 035: Farbe der schwarzen Listbox für die Menuitems. Default NULL.
Zeile 036: Farbe des Listbox Cursors. Default NULL.

Zeile 038: Startindex für den Listbox Cursor. Default 0.

Zeile 039: Array der Menuitems.

Zeile 040: Menuitem, Leerzeile { .Typ=Leer }

Zeile 041: Menuitem

{.Typ=Run,	Atemtyp: Ausführbare Funktion
.Txt=" "FK"Menudefinition anzeigen",	Menuzeile mit Funktionstaste: Menudefinition anzeigen
.Ptr=printMenuDefAktuell,	Funktionsname
.Key='m'	Tatsächliche Funktionstaste 'm'
},	

Zeile 046: Menuitem

{.Typ=Menu,	Itemtyp: Menu
.Txt=" Menu "FK"1 aufrufen",	Menuzeile mit Funktionstaste: Menu 1 aufrufen
.Ptr=&Menu1,	Menudefinition
.Key='1'	Tatsächliche Funktionstaste '1'
},	

Zeile 046: Menuitem

{.Typ=Menu,	Itemtyp: Menu.
.Txt=" "FK"Quit Menu",	Menuzeile mit Funktionstaste: Quit Menu
.Ptr=NULL	NULL für die Rückkehr zum letzten Menu oder Menuende.
},	Die Funktionstaste ist immer 'q' oder ESC

Zeile 049: Menuende { }. Für Menu-Definitionen am Stack muss zwingend { .Typ=MenuNil. } verwendet werden.

Beispiel Menuaufruf:

```
Code: Menu-Loop
140 ▷
141 ▷ void TestRunMenu()
142 ▷ { uint16_t Taste=taste_nil;
143 ▷
144 ▷   while(Taste != taste_return)
145 ▷   {
146 ▷     Taste=runMenu(&MenuHome, NULL, 0); // Menu anzeigen und Menu-Tasten auswerten
147 ▷
148 ▷     switch(Taste)
149 ▷     { case 'h': runHelp(); break;      // globale Funktionstaste 'h'
150 ▷
151 ▷       case taste_return:
152 ▷         BoxMenuDefDelete(&MenuHome);  // Option: Heapstrings in Menudefinition freigeben!
153 ▷       break;
154 ▷     }
155 ▷   }
156 ▷
157 ▷   printLn();
158 ▷   printf("Test runMenu() beendet!\n");
158 ▷ }
```

Bemerkungen zu TestRunMenu():

Zeile 142: Taste für die Rückgabe von runMenu()

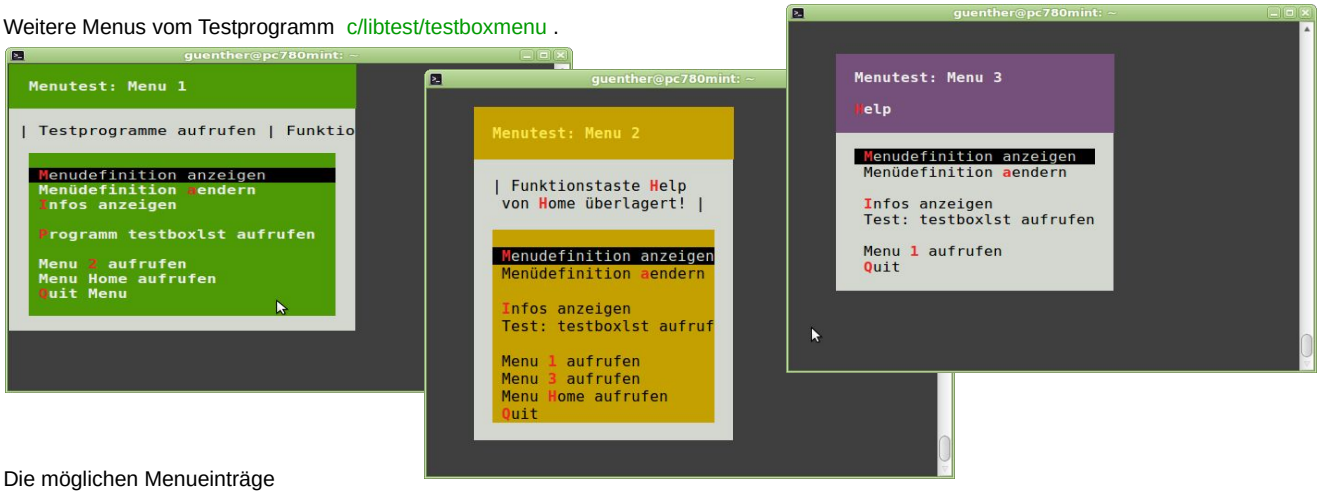
Zeile 146: runMenu() legt das Menu-Objekt an und startet die interne Menu-Loop.
Über die Keys werden Menubefehle ausgeführt oder weitere Menus gestartet.
runMenu() wird durch Taste 'q' oder ESC (Menuende) oder eine unbekannte Taste beendet.
Das Menu-Objekt wird mit Delete freigegeben.
Rückgaben: Bei Menuende taste_return . Sonst die Endetaste.

Zeile 149: Globale Funktionstaste Help behandeln.

Zeile 151: Menuende.

Zeile 152: Option: Wenn die statischen Strings der Menudefinition im Proramm durch Heapstrings ersetzt wurden, kann der Heap mit BoxMenuDefDelete(&MenuHome) bereinigt werden.

Weitere Menus vom Testprogramm c/libtest/testboxmenu .



Die möglichen Menueinträge

```
  typedef enum tBoxMenuTyp  // Typen für mögliche Menueinträge
  { MenuNil=0,     // {} Ende der Menüeinträge! Wichtig für SizeOfMenu()!!!
    Run,           // Ptr ist Funktionspointer vom Typ tBoxMenuRun.
    Bash,          // Ptr ist Bash-Befehlstring für Systemaufruf.
    Script,        // Ptr ist Name einer Script-Datei für Funktion tScriptRun.
    Menu,          // Ptr ist Menüdefinition vom Typ tBoxMenuDef.
    Leer,          // Leerzeile.
  }tBoxMenuTyp;
```

Die Menudefinition:

```
// ====================================================================
// MenuDef | Vollständige Beschreibung eines Menüs
//
typedef struct tBoxMenuDef   // Menüdefinition
{
  int16_t       y;           // erste Zeile
  int16_t       x;           // erste Spalte
  int16_t       h;           // Höhe der Box ohne Titel, <=0 vom Bildschirmende
  int16_t       b;           // Breite der Box,          <=0 vom vom Zeilenende
  char         *FarbeTitel;  // Farbe des Titels
  char         *Titel;       // Titeltext, Menükopf

  char         *FarbeInfo;   // Farbe Infotext, Menüinfo
  char         *Info;        // Menüinfotext

  tBox         *Show;        // Anzeigebox, Erweiterung, Koordinaten relativ zum Menu

  char         *FarbeZeile;  // Farbe der Menüzeilen in MenuLst
  char         *FarbeCursor; // Farbe des Cursors in MenuLst
  char         *FarbeScr;    // NULL oder Hintergrundfarbe des Terminals

  uint32_t      I;           // Startindex für den Menu-Cursor
  tBoxMenuItem *Items;       // Liste der Menüeinträge. Mit {} abschließen!

  tScriptRun    ScriptRun;   // NULL oder Funktion zum Ausführen von Script-Dateien
} tBoxMenuDef; // ------------------------------------------------------------
```

Menu-Scripte

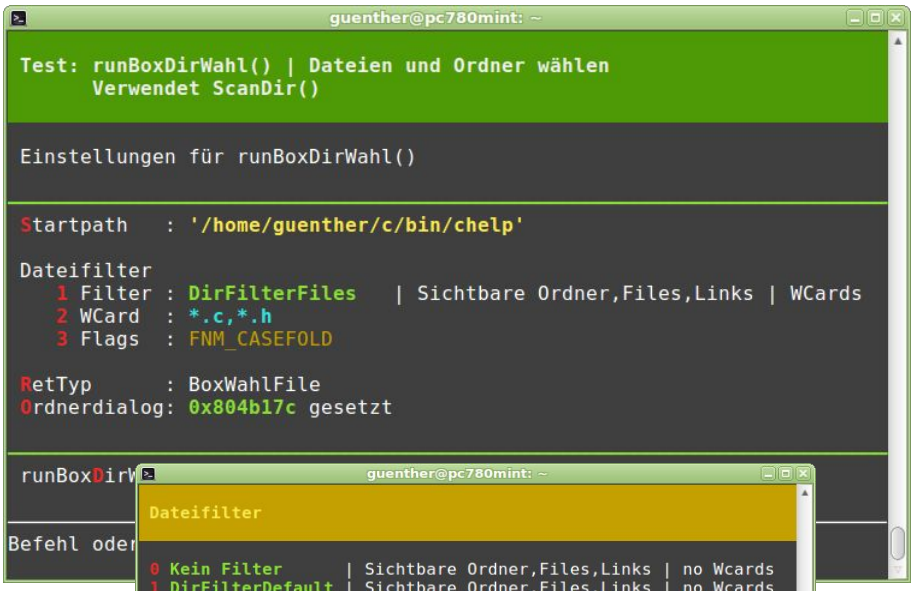
Menus können auch über Script-Dateien zur Laufzeit erzeugt werden! Siehe Programm saveit

Dateidialog

Beschreibungen chelp: Stichworte | Lib boxdirwahl.h: Dateidialog
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testboxlst d Test: runTestRunBoxDirWahl()

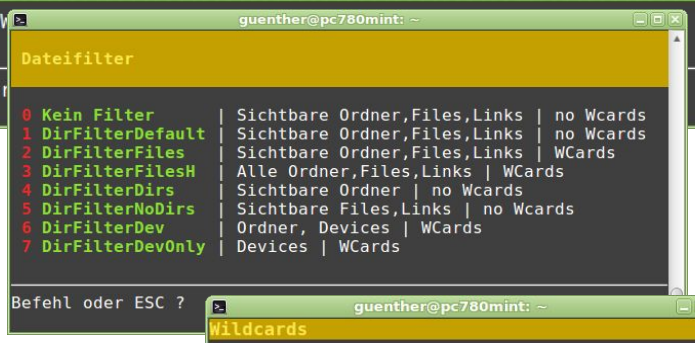
Der Dateidialog kann mit Defaulteinstellungen aufgerufen werden. Er bietet aber auch sehr umfangreiche Optionen an.
Die verschiedenen Möglichkeiten werden hier mit Hilfe des Testprogramms c/libtest/testboxlst beschrieben.

Optionen des Dateidialogs:



Startpath:

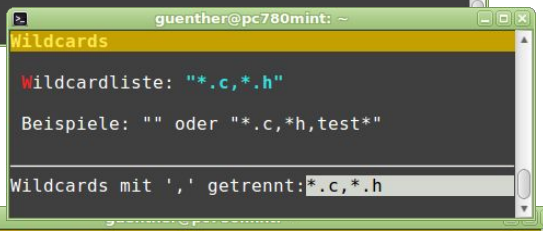
Startordner oder -datei im Dialog.



Dateifilter:

Der Filter legt die angezeigten Dateilisten fest.

Neben den vordefinierten Filtern können eigene Filter erstellt werden. Dafür stehen alle Elemente aus 'struct dirent' von scanDir() zur Verfügung.

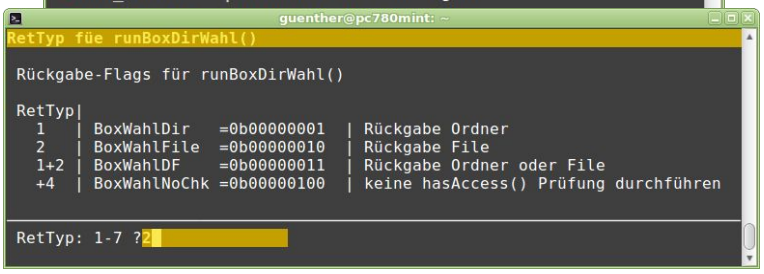


WCard: Wildcards

In manchen Filtern können zusätzlich Wildcardlisten für die Anzeige der Dateinamen verwendet werden.

Flags: Wildcard-Flags

Die Wildcard Treffer werden mit Funktion fnmatch() ermittelt.
Mit dem Flag FNM_CASEFOLD wird die Groß-/Kleinschreibung der Wildcardliste ignoriert.



RetTyp: Rückgabeflags

Sie steuern die mögliche Dialog-Rückgabe.

Normalerweise werden nur existierende Ordner, Files und Links gewählt.

Für manuell eingegebene Ordner/Dateinamen kann mit BoxWahlNoChk die Existenzprüfung abgeschaltet werden.

Ordnerdialog:

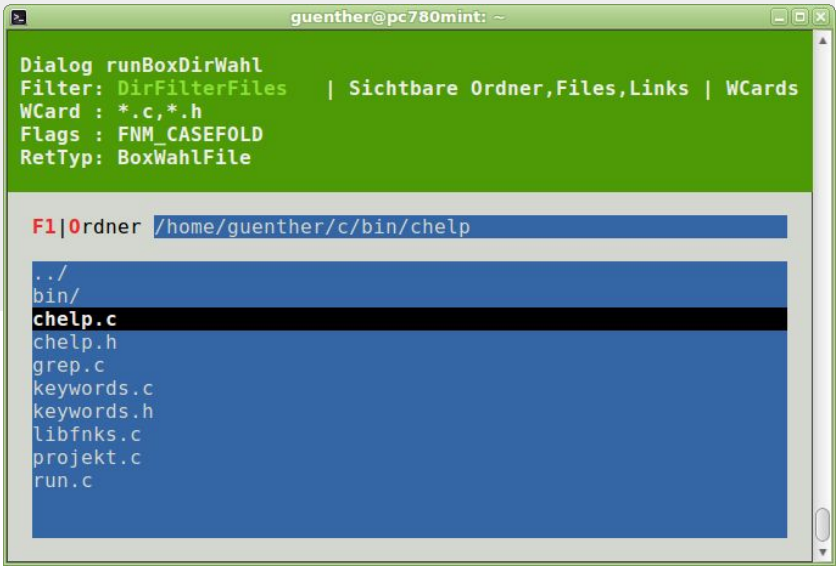
Mit optionalen Ordnerdialogen können zum Kontext passende Verzeichnisse und Funktionen im Dateidialog angeboten werden.

Beispiel: Ordnerdialog Archiv
Diese Dialog mountet USB-Laufwerke und zeigt die Archive

Der Aufruf des Dateidialogs im Testprogramm:

Die Funktion verwendet die oben ermittelten Einstellungen.

```
Code: testboxdirwahl.c
254 ▷ void TestRunBoxDirWahl()
256 ▷ {
257 ▷   DirFilterSetWildcardsStr(WildcardStr, WSep);
258 ▷   DirFilterSetFlags(FNM_Flags);
259 ▷
260 ▷   const char *erg=runBoxDirWahl
261 ▷   ( 1,1,-1,0,
262 ▷     StartPath,
263 ▷     DirFilterFiles, // mit Wildcards
264 ▷     RetTyp,
265 ▷     FCap4,
266 ▷     getTitel(),
267 ▷     OrdnerDlg
268 ▷   );
269 ▷
270 ▷   printLn();printLine(0); clrEos();
271 ▷
272 ▷   printf("Ergebnis: "Fy"%s"FN"\n",erg);
273 ▷   WeiterMitTaste();
274 ▷ }
```



Zeile 257: Option: Wildcards setzen.
Kann entfallen.

Zeile 258: Option: Wildcards-Flags setzen.
Kann entfallen.

Zeile 260: Aufruf runBoxDirWahl .
Rückgabe: NULL oder Dateipfad.

Die Funktions-Parameter:

Zeile 261: Boxkoordinaten
y=1, x=1 Ecke links/oben
h=-1 Höhe -1 vom unteren Rand
b=0 Breite 0 vom rechten Rand

Zeile 262: NULL oder String Startpfad.

Zeile 263: NULL oder Filterfunktion. Im Beispiel DirFilterFiles. Diese Funktion enthält Wildcards mit Flags.

Zeile 264: 0 oder die Art der Rückgabe. Im Beispiel: BoxWahlFile

Zeile 265: NULL oder Farbstring für die Titelbox. Im Beispiel: FCap4. Abkürzung für FarbeWhiteGreenB.

Zeile 266: NULL oder Titelstring. Im Beispiel wird der mehrzeilige Text in der Funktion getTitle() zusammengestellt.

Zeile 267: NULL oder Funktion Ordnerdialog.

Zeile 272: Ergebnis anzeigen.

F1: Fenster für Funktionstasten des Dialogs.

Ordner: Ordnerdialog

Dateidialog Default:

```
const char *erg=runBoxDirWahl
  ( 1,1,-1,0,
    NULL,
    NULL,
    0,
    NULL,
    NULL,
    NULL
  );
```

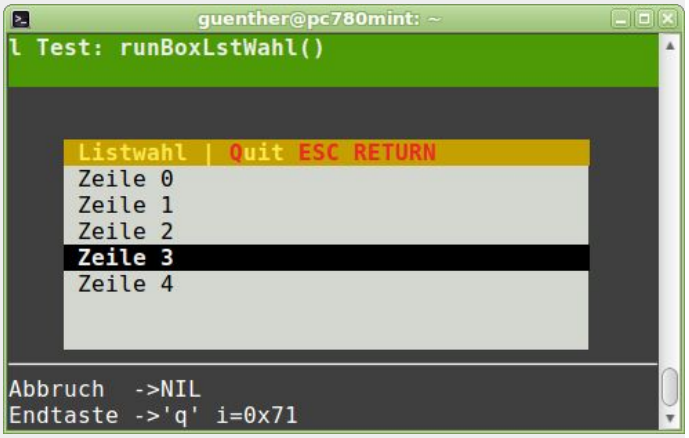
BoxLstWahl

Beschreibungen chelp: Stichworte | Lib boxlst.h : Listwahldialog
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testboxlst L Test: runBoxLstWahl()

Mit Funktion runBoxLstWahl() können Auswahllisten erzeugt werden.

Auswahlliste mit Testprogramm:

```
Code: testboxlst.c
289 ▷ void TestRunBoxLstWahl()
290 ▷ {
293 ▷   clrScr(); hideCursor();
294 ▷   printBlock("l Test: runBoxLstWahl()\n" "\n", FCap4);
295 ▷   printLn();
296 ▷
297 ▷   tArray *a=ArrayNew(5,0);
298 ▷
299 ▷   ArrayAddStr(a,
300 ▷     " Zeile 0\n"
301 ▷     " Zeile 1\n"
302 ▷     " Zeile 2\n"
303 ▷     " Zeile 3\n"
304 ▷     " Zeile 4"
305 ▷   );
306 ▷
307 ▷   uint16_t EndTaste=taste_nil;
308 ▷
309 ▷   uint32_t idx=runBoxLstWahl
310 ▷   ( 5,5, 8, -5
311 ▷     ,FarbeBlackGray,FarbeWhiteBlackB
312 ▷     ,a
313 ▷     ,3
314 ▷     ,NULL
315 ▷     ," Listwahl | "FK"Quit "FT"ESC RETURN"FN"\n"
316 ▷     ,&EndTaste
317 ▷   );
318 ▷   printLn();printLine(0);
319 ▷
320 ▷   if (idx==NIL) printf("Abbruch  ->NIL\n");
321 ▷   else          printf("Ergebnis ->%i\n", idx);
322 ▷   printf
323 ▷   ("Endtaste ->'%c' i=0x%x\n"
324 ▷    ,(EndTaste>=' ') ? EndTaste : taste_nil
325 ▷    ,EndTaste
326 ▷   );
327 ▷
328 ▷   ArrayDelete(a);
330 ▷ }
```

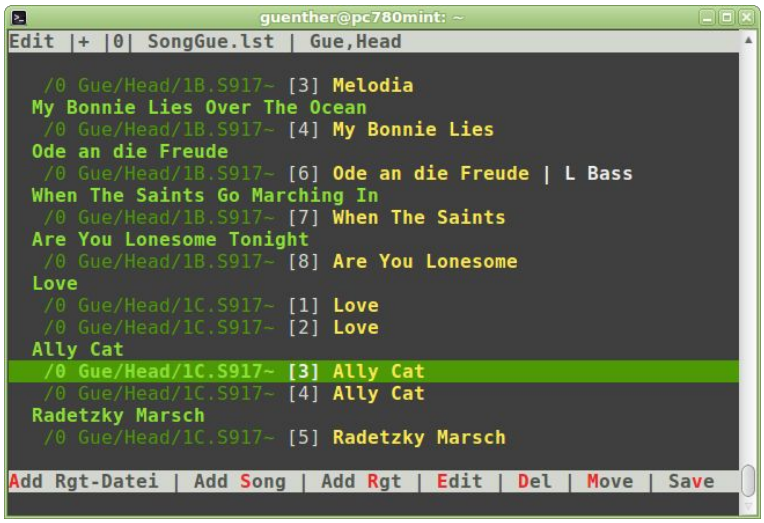


Zeile 297: bis
Zeile 304: Array mit den Daten der Liste. Beliebiges Array mit passender ArrayPrintItem() Funktion.
Zeile 307: EndTaste für die Rückgabe von runBoxLstWahl()
Zeile 309: Aufruf von runBoxLstWahl()
Zeile 310: Boxkoordinaten
y=5, x=8 Ecke links/oben
h=8 Höhe 8 Zeilen
b=-5 Breite -5 vom rechten Rand
Zeile 311: Farbe Zeile, Farbe Cursor
Zeile 312: Datenarray
Zeile 313: Startposition für den Cursor
Zeile 314: Titelfarbe, Defaultwert
Zeile 315: Titeltext mit Funktionstaste FK und Tastenfarbe FT.
Zeile 316: Referenz auf EndTaste.
Zeile 320: Rückgabe-Index idx ist Arrayindex oder NIL
Zeile 322: EndTaste auswerten.

Definition aus c/lib/include/iocon.h

```
// ============================================================================
// Dialog-Funktion für Listwahl.
// Verwendet Boxobjekt tBoxLst.
//
uint32_t runBoxLstWahl(
    int16_t     y,              // erste Zeile
    int16_t     x,              // erste Spalte
    int16_t     h,              // Höhe,   <1 vom Bildschirmende
    int16_t     b,              // Breite, <1 vom rechten Rand
    const char *FarbeZeile,     // Zeilenfarbe
    const char *FarbeCursor,    // Farbe der Cursorzeile
    tArray     *a,              // Datenarray definiertem ArrayPrintItem()
    uint32_t    Index,          // 0 oder Cursorposition in a
    const char *FarbeTitel,     // Farbe der Überschrift
    const char *Titel,          // Überschrift

    uint16_t   *EndTaste        // Einfabe : NULL oder &Taste
    //                          // Rückgabe: Die letzte Taste
  );
  // Listwahldialog mit Objekt tBoxLst.
  // Rückgabe: Array-Index oder NIL
// ============================================================================
```

BoxLst



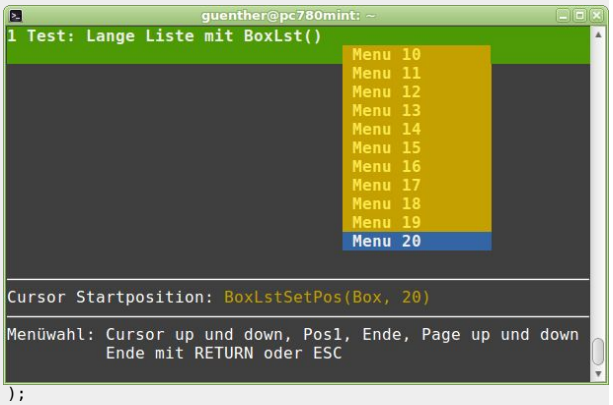
Mit der Basisfunktion BoxLstl() können komplexe Listendialoge erzeugt werden.

Beispiel: Songverwaltung von Programm kbdctl

Beschreibungen chelp: Stichworte | Lib boxlst.h : Listwahldialog
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testboxlst 1 Test: Lange Liste mit BoxLst()

Auswahlliste mit Basisobjekt BoxLst:

```
Code: testboxlst.c
181 ▷ void Test1()
182 ▷ { clrScr(); hideCursor(); printBlock("1 Test: Lange Liste mit BoxLst()\n\n", FCap4);
183 ▷
184 ▷   tArray *a=ArrayNew(39,0);
185 ▷   for(uint16_t i=0; i<38; i++) ArrayAddStr(a, tmpStrF(" Menu %i\n", i) );
186 ▷
187 ▷   tBoxLst *Box=BoxLstNew
188 ▷   ( 2,35,11,15
189 ▷     ,FarbeYellowYellowB  // Farbe Zeile
190 ▷     ,FarbeWhiteBlueB     // Farbe Cursor
191 ▷     ,a
192 ▷   );
193 ▷
194 ▷   BoxLstSetPos(Box, 20);
195 ▷
196 ▷   gotoYX(14,1); printLine(0); savePos();
197 ▷   printf("Cursor Startposition: "Fy"BoxLstSetPos(Box, 20)\n"FN);
198 ▷   printLine(0);
199 ▷   PrintCursorTasten();
200 ▷
201 ▷   BoxPrintAll(Box);
202 ▷   uint16_t Taste   =BoxControlLoop(Box);
203 ▷   uint16_t EndTaste=BoxEndTaste(Box);
204 ▷   uint32_t Index   =BoxLstIndex(Box);
205 ▷
206 ▷   restPos(); clrEos();
207 ▷   printf("Ergebnis: "FY"Taste=0x%x, EndTaste=0x%x, ",Taste ,EndTaste );
208 ▷   if (Index!=NIL) printf("Index=%u, Text=\"%s\"\n"FN, Index, ArrayPrintItem(a, Index) );
209 ▷   else            printf("Index=NIL\n"FN);
210 ▷   printLine(0);
211 ▷
212 ▷   BoxLstPrintInfo(Box,false);
213 ▷   BoxDelete(Box); ArrayDelete(a);
214 ▷
215 ▷   WeiterMitTaste();
217 ▷ }
```



Zeile 187: BoxLstNew(...) Objekt BoxLst anlegen. In diese Box können weitere Boxen eingefügt werden.
Zeile 194: Cursorposition setzen.
Zeile 201: BoxPrintAll(Box) zeigt die BoxLst und alle eingefügten Box an.
Zeile 202: BoxControlLoop(Box): Die Funktion ermittelt Events und steuert die Box. Rückgabe: taste_esc oder taste_return .
Zeile 203: BoxEndTaste(Box) liefert die tatsächliche End-Taste.
Zeile 204: BoxLstIndex(Box): NIL oder Cursorposition.
Zeile 213: BoxDelete(Box) gibt das Objekt BoxLst wieder frei.

Box Basisobjekt

Basisobjekt tBox

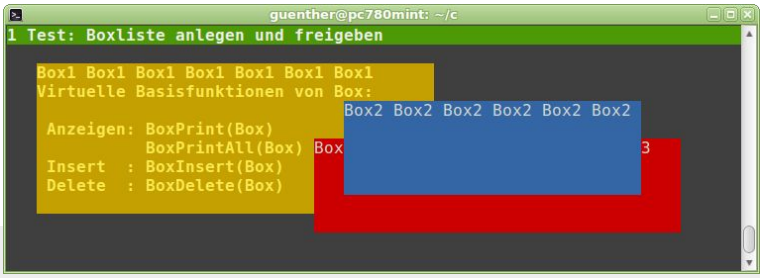
tBox ist das Basisobjekt für alle Dialogelemente.

```
// ========================================================================
// | Library  : iocon.a  |||||||||||||||||||||||||||||||||||||||||||||||||
// | Modul    : box.h    |||||||||||||||||||||||||||||||||||||||||||||||||
// | Objekt   : tBox     |||||||||||||||||||||||||||||||||||||||||||||||||
// | Test/Doku: testbox  |||||||||||||||||||||||||||||||||||||||||||||||||
// |
// | Eine Box verwaltet einen rechteckigen Bildschirmbereich.
// | Mit BoxInsert() können Box-Listen gebildet werden.
// | Eine Box kann mit BoxInsertItem() weitere Box-Items verwalten.
// |
// | tBox ist das Basisobjekt für alle abgeleiteten Boxen.
// |
// | Boxfunktionen für alle Boxen:
// |   Delete;   // virtueller Destruktor
// |   Print;    // virtuelle Printfunktion
// |   Control;  // virtuelle Control-Funktion zur Steuerung der Box//
// |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
// ========================================================================
```

Beschreibungen chelp: Stichworte | Lib box.h : Box Basisobjekt
Modul: c/lib/include/iocon.h
Testprogramm: c/libtest/testboxlst

Komplexe Dialogelemente bestehen aus Box-Objekten in einer doppelt verketteten Liste.

Beispiel: 1 Test: Boxliste anlegen und freigeben



Box2 und Box3 in Box1 einfügen.

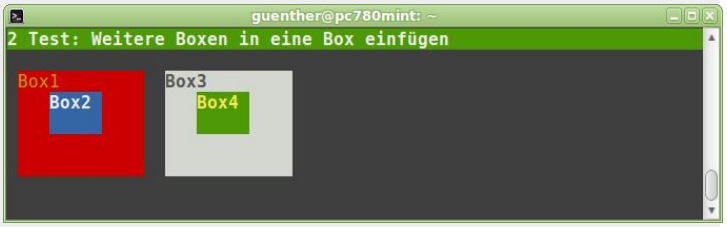
```
Code: testbox.c
425 ▷
426 ▷   tBox *Box1=BoxNew
427 ▷   ( 3,  4,                    // Zeile, Spalte
428 ▷     8, 40,                    // Höhe, Breite
429 ▷     FarbeYellowYellowB,       // Boxfarbe
430 ▷     "Box1 Box1 Box1 Box1 Box1 Box1 Box1"   // Textzeilen
431 ▷     "\n"
432 ▷     "Virtuelle Basisfunktionen von Box:\n"
433 ▷     "\n"
434 ▷     " Anzeigen: BoxPrint(Box)\n"
435 ▷     "           BoxPrintAll(Box)\n"
436 ▷     " Insert  : BoxInsert(Box)\n"
437 ▷     " Delete  : BoxDelete(Box)"
438 ▷   );
439 ▷
440 ▷   tBox *Box2=BoxNew
441 ▷   ( 5,  35,                              // erste Zeile, erste Spalte
442 ▷     5, -10,                             // Höhe, Breite
443 ▷     FarbeWhiteBlue,                     // Farbe
444 ▷     "Box2 Box2 Box2 Box2 Box2 Box2 Box2"   // Textzeilen
445 ▷   );
446 ▷
447 ▷   tBox *Box3=BoxNew
448 ▷   (  7, 32,                             // erste Zeile, erste Spalte
449 ▷      5, -6,                             // Höhe, Breite
450 ▷     FarbeWhiteRed,                      // Farbe
451 ▷     "Box3 Box3 Box3 Box3 Box3 Box3 Box3"   // Textzeilen
452 ▷   );
453 ▷
454 ▷   BoxInsert(Box1,Box2);   // Box2 am Ende der Boxliste einfügen
455 ▷   BoxInsert(Box1,Box3);   // Box3 am Ende der Boxliste einfügen
456 ▷   BoxPrintAll(Box1);      // Boxliste anzeigen
457 ▷
```

Zeile 454: BoxInsert(Box1, Box2) fügt Box2 am Ende in die Boxliste von Box1 ein.

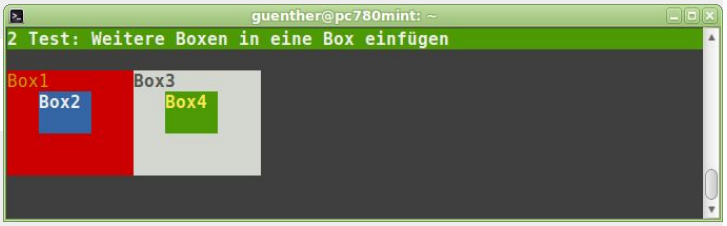

```
Code: testbox.c
469 ▷
470 ▷   BoxPrint(Box2);      // Box2 anzeigen
471 ▷   BoxPrint(Box3);      // Box3 anzeigen
472 ▷   BoxPrint(Box1);      // Box1 anzeigen
473 ▷
474 ▷   BoxDeleteAll(Box1);  // Boxliste freigeben
475 ▷
```

Beispiel: 2 Test: Weitere Boxen in eine Box einfügen

```
Code: testbox.c
490 ▷   tBox *Box1=BoxNew
491 ▷   ( 3,  2,  5,  12,
492 ▷     FarbeYellowRed,
493 ▷     "Box1"
494 ▷   );
495 ▷
496 ▷   tBox *Box2=BoxNew
497 ▷   ( 1,  3, -2, -4, // Koordinaten relativ zu Box1
498 ▷     FarbeWhiteBlueB,
499 ▷     "Box2"
500 ▷   );
501 ▷
502 ▷   tBox *Box3=BoxNew
503 ▷   ( 3, 16, 5, 12,
504 ▷     FarbeBlackGrayB,
505 ▷     "Box3"
506 ▷   );
507 ▷
508 ▷   tBox *Box4=BoxNew
509 ▷   ( 1,   3, -2,  -4, // Koordinaten relativ zu Box2
510 ▷     FarbeYellowGreenB,
511 ▷     "Box4"
512 ▷   );
513 ▷
514 ▷   BoxInsertItem(Box1,Box2); // Box2 als einfaches Item in Box1 einfügen
515 ▷   BoxInsertItem(Box3,Box4); // Box4 als einfaches Item in Box3 einfügen
516 ▷   BoxPrint(Box1);
517 ▷   BoxPrint(Box3);
518 ▷
```

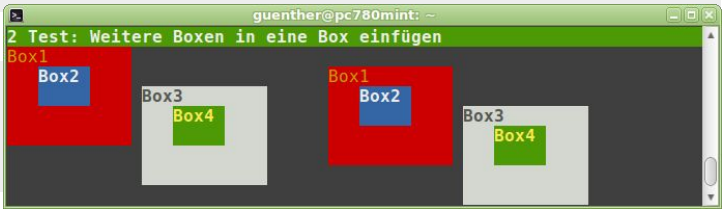


Zeile 454: BoxInsertItem(Box1, Box2) fügt Box2 als Item in Box1 ein. Die Boxliste bleibt unverändert.

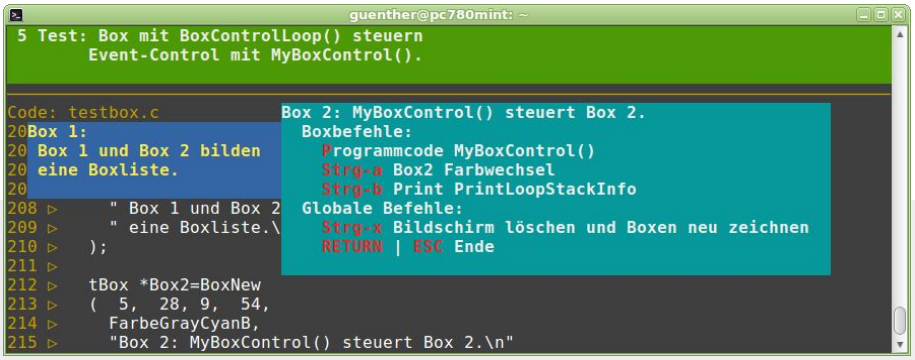



```
Code: testbox.c
523 ▷
524 ▷   BoxMoveToYX(Box1, 2, 1); // move Box1 nach (2, 1)
525 ▷   BoxMoveToYX(Box3, 2,13); // move Box3 nach (2,13)
526 ▷   BoxPrint(Box1);
527 ▷   BoxPrint(Box3);
528 ▷
```

```
Code: testbox.c
534 ▷
535 ▷   BoxInsertItem(Box1, Box3);  // Box3 in Box1 einfügen
536 ▷   BoxPrintAll(Box1);
537 ▷   BoxMoveToYXAll(Box1, 3,32); // Box1 verschieben
538 ▷   BoxPrintAll(Box1);
539 ▷
```



Beispiel: 5 Test: Box mit BoxControlLoop() steuern




```
204 ▷   tBox *Box1=BoxNew
205 ▷   (  6,  3, 4, 30,
206 ▷     FarbeYellowBlue,
207 ▷     "Box 1:\n"
208 ▷     " Box 1 und Box 2 bilden\n"
209 ▷     " eine Boxliste.\n"
210 ▷   );
211 ▷
212 ▷   tBox *Box2=BoxNew
213 ▷   (  5,  28, 9,  54,
214 ▷     FarbeGrayCyanB,
215 ▷     "Box 2: MyBoxControl() steuert Box 2.\n"
216 ▷     "  Boxbefehle:\n"
217 ▷     "    "FK"Programmcode MyBoxControl()\n"
218 ▷     FT"    Strg-a"FN" Box2 Farbwechsel\n"
219 ▷     FT"    Strg-b"FN" Print PrintLoopStackInfo\n"
220 ▷     "  Globale Befehle:\n"
221 ▷     FT"    Strg-x"FN" Bildschirm löschen und Boxen neu zeichnen\n"
222 ▷     FT"    RETURN"FN" | "FT"ESC"FN" Ende\n"
223 ▷   );
224 ▷
225 ▷   BoxSetControl(Box2,MyBoxControl);    // Event-Control für Box2 setzen
226 ▷   BoxInsert(Box1, Box2);               // Boxliste bilden
227 ▷   BoxPrintAll(Box1); fflush(stdout);   // Boxliste anzeigen
228 ▷
229 ▷   uint16_t Taste=BoxControlLoop(Box2); // Rückgabe
230 ▷
272 ▷   BoxDeleteAll(Box1);
```

Zeile 225: BoxSetControl(...) setzt die Event-Control Funktion.

Zeile 227: BoxPrintAll(Box1): Boxliste anzeigen. fflush(stdout) zeigt Boxliste sofort an. Normalerweise wird die Anzeige mit fflush() durch einen Zeilenwechsel "\n" ausgelöst.

Zeile 229: BoxControlLoop (...) Event-Control aufrufen.

Die Event-Control Funktion MyBoxControl:

```
151 ▷ uint16_t MyBoxControl(void *Box, uint16_t Control)
152 ▷ { if (!Box) return taste_nil;
153 ▷   tBox *p=(tBox *) Box;
154 ▷
155 ▷   switch (Control)
156 ▷   { case taste_ctl_input: return taste_ctl_true;
157 ▷     // Frage an Box: Control-Fokus möglich?
158 ▷     // Antwort     : Box bearbeitet Control-Eingaben
159 ▷
160 ▷     case taste_ctl_a:
161 ▷       gotoYX(13,1); clrEos();
162 ▷       BoxSetTxt(p, " Weiter mit ESC\n" " Strg-a: Farbwechsel\n" " Strg-b: Print Info\n");
163 ▷
164 ▷       if (0==strcmp(FCap3, BoxGetFarbe(p))) BoxSetFarbe(p, FCap1);
165 ▷       else  BoxSetFarbe(p, FCap3);
166 ▷
167 ▷       BoxPrint(p); fflush(stdout); // Box neu anzeigen
168 ▷     return taste_nil;              // Tasten-Event erledigt
169 ▷
170 ▷     case taste_ctl_b:
171 ▷       gotoYX(13,1); clrEos();
172 ▷       printLine(0);
173 ▷       printf("Info: Taste Strg-b gedrückt\n");
174 ▷       printLine(0);
175 ▷       PrintLoopStackInfo();
176 ▷     return taste_nil;              // Tasten-Event erledigt
177 ▷
178 ▷     case 'p':
179 ▷       clrScr();                   // Programmcode anzeigen
180 ▷
181 ▷      printCode(__LINE__-32,__FILE__); printCode(__LINE__+8,NULL); WeiterMitTaste();
182 ▷       gotoYX(6,3);
183 ▷       BoxPrint(p); fflush(stdout); // Box neu anzeigen
184 ▷     return taste_nil;              // Tasten-Event erledigt
185 ▷
186 ▷     case taste_return: Control=taste_esc;
187 ▷
188 ▷     default: return Control;
189 ▷   }
190 ▷ }
```

Zeile 151: Die eigenen Event-Control Funktionen werden von BoxControlLoop() mit den Parametern Box und Control-Taste gerufen. Folgende Rückgaben sind möglich.

Zeile 156: Anfrage taste_ctl_input: Antwort taste_ctl_true ,wenn die Box bereit ist Tasten auszuwerteten.

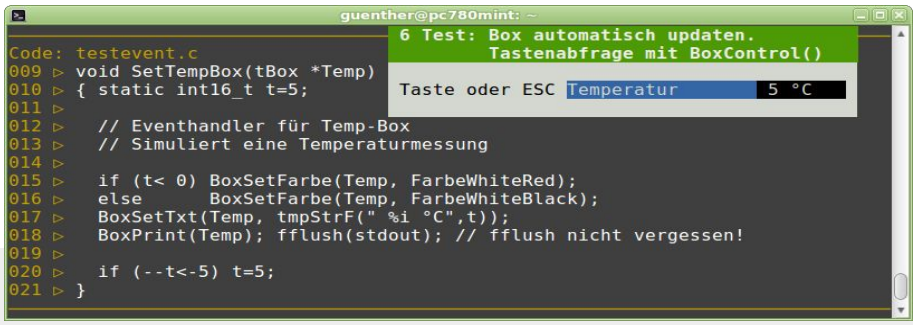
Zeile 168: Antwort taste_nil: Tasten-Event wurde vollständig ausgewertet.

Zeile 188: Antwort Control: Auswertung fortsetzen.

Beispiel: 6 Test: Box automatisch updaten. Tastenabfrage mit BoxControl()

Das abgebildete Objekt wird aus vier Boxen zusammengesetzt:

1. Grüne Titelbox: " 6 Test: ..."
2. Graue Textbox: "Taste oder ESC"
3. Blauer Text: "Temperatur"
4. Schwarze Temp-Box: " 5 °C"




```
Code: testevent.c
032 ▷   tBox *Box= BoxNew
033 ▷   (1,35,2,42, FarbeWhiteGreenB,
034 ▷    " 6 Test: Box automatisch updaten.\n"
035 ▷    "         Tastenabfrage mit BoxControl()\n"
036 ▷   );
037 ▷   BoxInsertItem(Box, BoxNew( 2,0,3,42, FarbeBlackGray, "\n Taste oder ESC\n"));
038 ▷   BoxInsertItem(Box, BoxNew( 3,16,1,17, FarbeWhiteBlue, "Temperatur"));
039 ▷
040 ▷   tBox* Temp=BoxInsertItem(Box, BoxNew( 3,16+17,1,8, FarbeWhiteBlack, NULL));
041 ▷
042 ▷   BoxSetEventHandler(Temp, SetTempBox);
043 ▷   BoxPrintAll(Box);
044 ▷
045 ▷   uint16_t Taste=taste_nil;
046 ▷
047 ▷   while (Taste!=taste_esc)
048 ▷   { BoxEventHandler(Temp);                    // Temp lesen/anzeigen
049 ▷     Taste=BoxControl(Box, chkTaste(99999999)); // Auf Taste warten
050 ▷
051 ▷     switch(low(Taste))
052 ▷     { case taste_nil:
053 ▷       case taste_esc: break;
054 ▷       default:
055 ▷         gotoYX(9,1); clrEos();printLine(0);    // Taste anzeigen
056 ▷         printf("Taste: "Fy"0x%x\n"FN,Taste);
057 ▷     }
058 ▷   }
059 ▷
060 ▷   BoxDeleteAll(Box);
```

Das Anzeigeobjekt erzeugen:

Zeile 032: Die grüne Titelbox Box bildet die Objektbasis.

Zeile 037: Mit BoxInsertItem() wird die graue Textbox hinzugefügt. BoxControl bleibt damit bei Box.
Zeile 038: Mit BoxInsertItem() wird der blaue Text hinzugefügt. BoxControl bleibt bei Box.
Zeile 040: Mit BoxInsertItem() wird die scharze Temp hinzugefügt. Mit Temp kann auf diese Box zugegriffen werden..

Zeile 042: Die Event-Funktion SetTempBox() wird mit Box-Temp verbunden.
Der Sourcecode von SetTempBox() ist im obigen Bild ersichtlich.

Zeile 043: BoxPrintAll(Box) zeigt alle Boxen an.

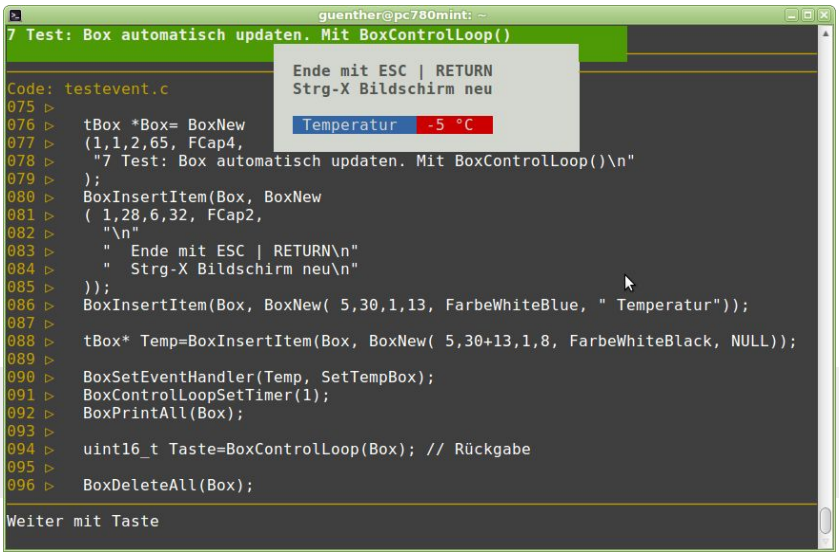
Zeile 047: Event-Loop
BoxEventHandler(Temp) ruft Funktion SetTempBox() auf. Die Temp-Box wird aktualisiert und angezeigt.
BoxControl() wird mit chkTaste(s nsec) aufgerufen. chkTaste wartet nicht blockierend s nanosec.
Danach liefert chkTaste Taste taste_nil. Bei Unterbrechung mit Tastendruck liefert chkTaste diese Taste.

Zeile 051: Tasten-Rückgabe von BoxControl() auswerten.

Beispiel: 7 Test: Box automatisch updaten. Mit BoxControlLoop()

Beispiel ohne Tastenabfrage.
Rückgabe ESC oder RETURN.

Tastenabfragen können mit BoxSetControl() implementiert werden. Siehe Beispiel 5

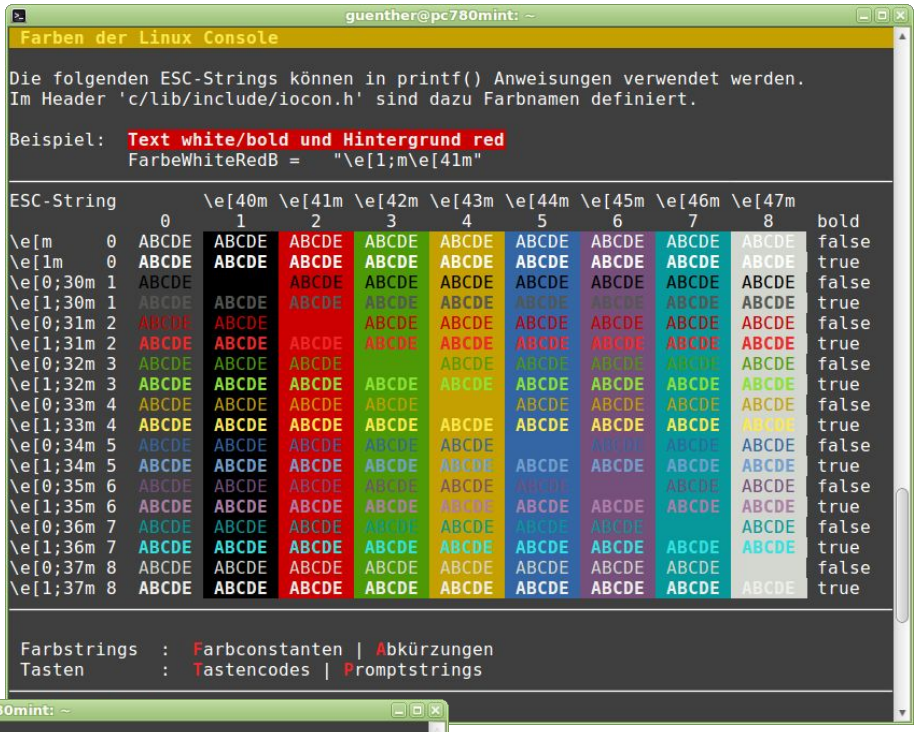



```
090 ▷   BoxSetEventHandler(Temp, SetTempBox);
091 ▷   BoxControlLoopSetTimer(1);
092 ▷   BoxPrintAll(Box);
093 ▷
094 ▷   Taste=BoxControlLoop(Box);
095 ▷
096 ▷   BoxDeleteAll(Box);
```

```
Code: testevent.c
075 ▷
076 ▷   tBox *Box= BoxNew
077 ▷   (1,1,2,65, FCap4,
078 ▷    "7 Test: Box automatisch updaten. Mit BoxControlLoop()\n"
079 ▷   );
080 ▷   BoxInsertItem(Box, BoxNew
081 ▷   ( 1,28,6,32, FCap2,
082 ▷     "\n"
083 ▷     "  Ende mit ESC | RETURN\n"
084 ▷     "  Strg-X Bildschirm neu\n"
085 ▷   ));
086 ▷   BoxInsertItem(Box, BoxNew( 5,30,1,13, FarbeWhiteBlue, " Temperatur"));
087 ▷
088 ▷   tBox* Temp=BoxInsertItem(Box, BoxNew( 5,30+13,1,8, FarbeWhiteBlack, NULL));
089 ▷
090 ▷   BoxSetEventHandler(Temp, SetTempBox);
091 ▷   BoxControlLoopSetTimer(1);
092 ▷   BoxPrintAll(Box);
093 ▷
094 ▷   uint16_t Taste=BoxControlLoop(Box); // Rückgabe
095 ▷
096 ▷   BoxDeleteAll(Box);
```

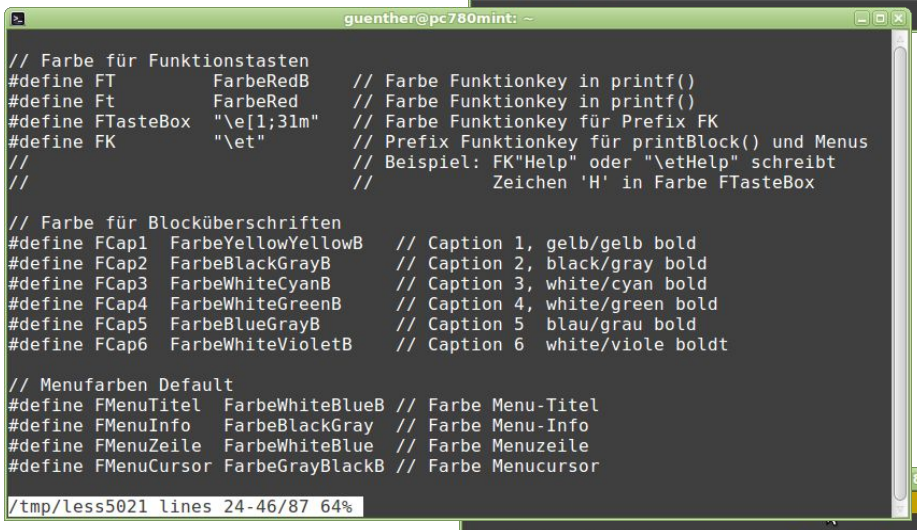
Farben und Tasten

Siehe Programm chelp | Farben und Tasten.

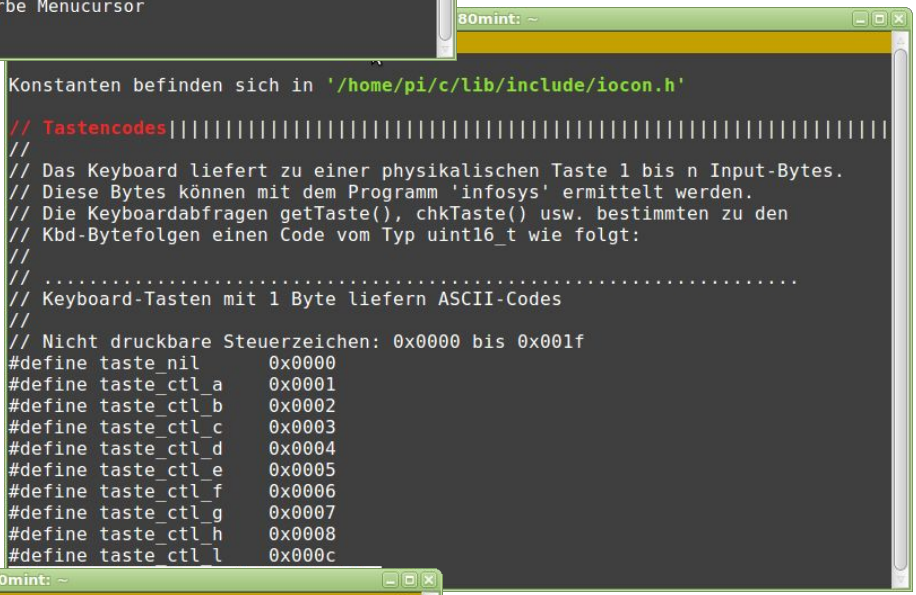


Farbkonstanten

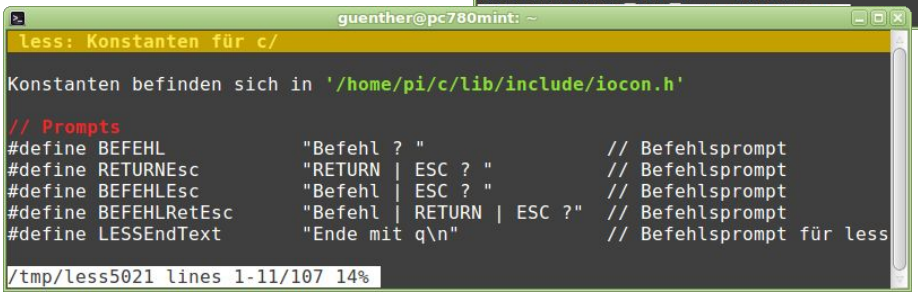
Abkürzungen



Tastencodes



Promptstrings



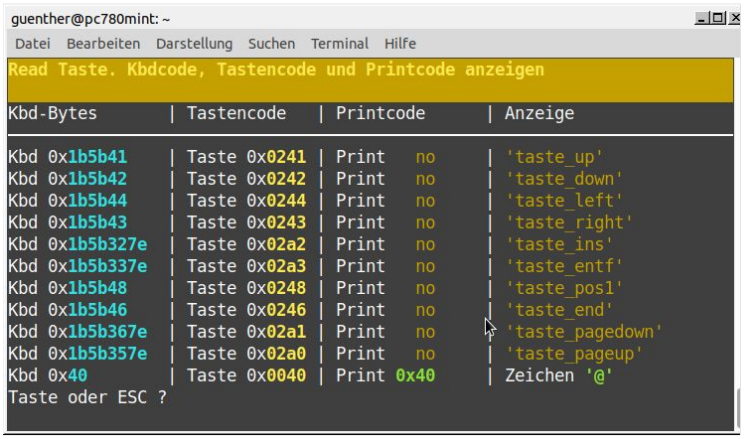
Die Tastencodes

Der Tastecode ist vom Datentyp uint16_t.
Die Tastencodes erfassen ASCII-Codes, UTF8-Codes, Funktionstasten, Control-Befehle und frei definierte Codes.
Sie werden für Tastatureingaben und Steueraufgaben verwendet.

Alle Codes werden in der Tabelle Tastencodes mit aufsteigenden Nummern von taste_nil bis taste_not_def festgelegt.
Nur verwendete Codes wurden in der Tabelle bereits definiert. Weiteren Keyboardtasten mit mehreren Bytes kann in der Tabelle Key3codes eine uint16_t Taste zugewiesen werden.

Tabelle Tastencodes

Testprogramm: infosys | Keyboard
Tasten-Eingaben testen



Programm: chelp | Farben und Tasten Zeigt die definierten Tastencodes.

```
// Tabelle Tastencodes ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
//
// Das Keyboard liefert zu einer physikalischen Taste 1 bis n Input-Bytes.
// Diese Bytes können mit dem Programm 'infosys' ermittelt werden.
// Die Keyboardabfragen getTaste(), chkTaste() usw. bestimmten zu den
// Kbd-Bytefolgen einen Code vom Typ uint16_t wie folgt:
//
// ..............................................................
// Keyboard-Tasten mit 1 Byte liefern ASCII-Codes
//
// Beispiel: uint16_t Taste='a';
//
```

Für Steuerzeichen sind Bezeichner definiert:

```
// ..............................................................
// Keyboard-Tasten mit 1 Byte liefern ASCII-Codes
//
// Nicht druckbare Steuerzeichen: 0x0000 bis 0x001f //
#define taste_nil      0x0000
#define taste_ctl_a    0x0001
#define taste_ctl_b    0x0002
...
#define taste_ctl_z    0x001a

#define taste_esc      0x001b
#define taste_return   0x000a
#define taste_tab      0x0009
//                     ...        weitere Bezeichner für 1 Byte-Tasten
//                     ...        hier definieren
//                     ...        bis 0x001f
//
#define taste_plus     0x002B
#define taste_del      0x007f     // Steuertaste, nicht druckbar
```

```
// ..............................................................
// Druckbare ASCII-Codes:           von 0x0020 bis 0x01ff
//
#define taste_blank    0x0020     // erste druckbare Taste
//                     ...        // ASCII-Zeichen von 0x0020 bis 0x007e
                                  // Taste=ASCII-Code
#define taste_0        0x0030     //'0' und taste_0 gleichwertig
...                               // ASCII
#define taste_ja       0x006a     //'j'
#define taste_nein     0x006e     //'n'
#define taste_q        0x0071     //'q'
```

```
// ..............................................................
// Keyboard-Tasten mit mehr als 2 Bytes werden in der
// Tabelle Key3Codes[] im Modul gettaste.c in frei definiert.
// ----------------------------------------------------------------
// Die Definition neuer Tastencodes erfordert zwei Schritte:
//    1. Tastencode (uint16_t) Wert in der Tabelle Tastencodes definieren.
//       z.B. #define taste_xxx  0x0101
//    2. Input-Bytefolge in der Tabelle Key3Codes[] in gettaste.c definieren.
//       z.B. { "\x1b\x4f\x51",     taste_xxx },
// Mit dem Programm 'infosys' kann die Definition überprüft werden.
// ----------------------------------------------------------------

// Druckbare Tasten mit mehr als 2 Bytes
#define taste_key3tab  0x0100     // von 0x0100 bis 0x01ff frei definiert
#define taste_euro     0x0100     // Bezeichner für weitere druckbare
//                     ...           n-Byte (n>)Tasten hier und in
//                     ...           Tabelle Key3Codes[] definieren.
//                     ...        // frei
#define taste_lastp    0x01ff     // letzte druckbare Taste
```

```
// ..............................................................
// Nicht druckbare Steuertasten mit mehr als 2 Bytes
//                                //  von 0x0200 bis 0x02ff frei definiert
#define taste_f1       0x0201
#define taste_f2       0x0202
#define taste_f3       0x0203
#define taste_f4       0x0204
//#define taste_f5       0x0205
#define taste_up       0x0241
#define taste_down     0x0242
#define taste_right    0x0243
#define taste_left     0x0244
#define taste_end      0x0246
#define taste_pos1     0x0248
//                     ...
#define taste_pageup   0x02a0
#define taste_pagedown 0x02a1
#define taste_ins      0x02a2
#define taste_entf     0x02a3
#define taste_ctl_entf 0x02a4
//                     ...            Bezeichner für weitere n-Byte Steuertasten
//                     ...            hier und in Key3Codes[] definieren.
#define taste_key3End  0x2fff     //  bis 0x2fff
// ..............................................................
```

```
// ..............................................................
// Keyboard-Tasten mit 2 Bytes
// Alt-Tasten:                       von 0x1b30 bis 0x1c7a
//                                   ESC+Taste
//
// Druckbare Sonderzeichen
// zwei Bytes UTF8-Code:             110xxxxx 10xxxxxx
#define taste_utfanf   0xc080     // UTF8-Code, 2 Bytes: Anfang
#define taste_para     0xc1a7     // '§'
#define taste_utfend   0xd7bf     // UTF8-Code, 2 Bytes: Ende
```

```
// ..............................................................
// Keine realen Tastencodes
// Control-Befehle für Boxen.
#define taste_ctl_true   0x1001   // ja-Antwort von Box
#define taste_ctl_input  0xf002   // Frage an Box:
//                                   Werden Control-Eingaben bearbeitet?
//                                   Antworten: taste_ctl_true oder taste_nil
#define taste_menu_nxt   0xf003
```

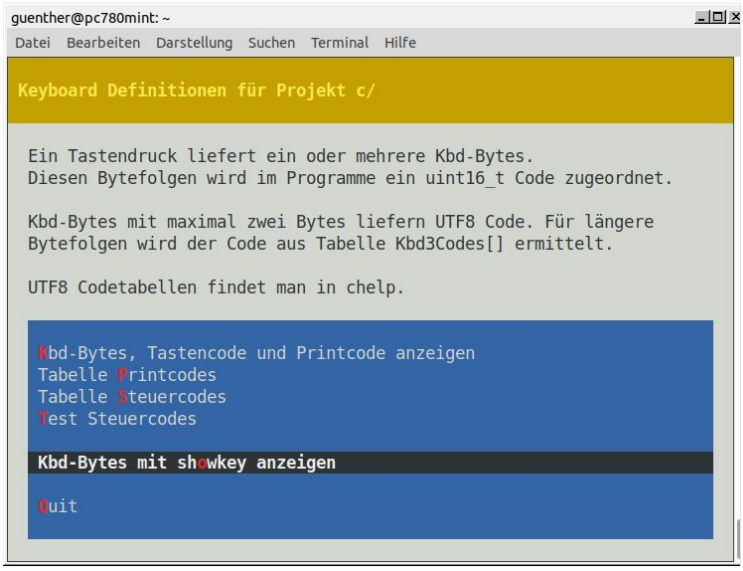
```
// ..............................................................
// Keine realen Tastencodes
// Codes zur freien Verwendung

#define taste_user0      0xfff0   // freier Tastecode
#define taste_user1      0xfff1   // freier Tastecode
#define taste_user2      0xfff2   // freier Tastecode
#define taste_user3      0xfff3   // freier Tastecode
#define taste_user4      0xfff4   // freier Tastecode
#define taste_user5      0xfff5   // freier Tastecode
#define taste_user6      0xfff6   // freier Tastecode
#define taste_user7      0xfff7   // freier Tastecode
#define taste_user8      0xfff8   // freier Tastecode
#define taste_user9      0xfff9   // freier Tastecode

#define taste_userA      0xfffa   // freier Tastecode
#define taste_userB      0xfffb   // freier Tastecode
#define taste_userC      0xfffc   // freier Tastecode
#define taste_userD      0xfffd   // freier Tastecode
#define taste_userE      0xfffe   // freier Tastecode
```

```
#define taste_not_def    0xffff   // nicht definierte Taste/Tastenfolge
```

Neue Tastencodes

```
// ..............................................................
// Keyboard-Tasten mit mehr als 2 Bytes werden in der
// Tabelle Key3Codes[] im Modul gettaste.c frei definiert.
// --------------------------------------------------------------
// Die Definition neuer Tastencodes erfordert folgende Schritte:
//    1. Tastencode (uint16_t) Wert in der Tabelle Tastencodes definieren.
//       z.B. #define taste_xxx  0x0101
//    2. Input-Bytefolge in der Tabelle Key3Codes[] in gettaste.c definieren.
//       z.B. { "\x1b\x4f\x51",     taste_xxx },
//    3. Tabelle printTaste() ergänzen
// Mit dem Programm 'infosys' kann die Definition überprüft werden.
// --------------------------------------------------------------
```

Testprogramm: infosys | Keyboard



GNU General Public License

```
/*
 *
 * Copyright 2022 Günther Schardinger <schardinger@projektc.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
*/
```