devtest | Testprogramm für Steuerungen

Inhaltsverzeichnis

Einleitung

In einfachen Steuerungen werden verschiedene Funktionmodule zum Auslesen von Sensoren und Ansteuern von Aktoren innerhalb einer Endloschleife aufgerufen. Die Funktionmodule für Sensoren und Aktoren ähneln einander sind aber nicht einfach austauschbar.

Zum Beispiel gibt es zum Messen von Temperaturen eine Fülle verschiedener Sensoren mit sehr unterschiedlichen Ansteuerungsmöglichkeiten. Für die Funktion einer bestimmte Steuerung ist es aber egal, welcher Sensor tatsächlich verwendet wird.

Mit dem Konzept Loop, Events und Devices und können Steuerungen aus verschieden Funktionsmodulen einfach zusammengestellt werden. Die passenden Module werden dabei über Konfiguratonsdateien festgelegt.

Die notwendigen Programme zum Aufbau einer Steuerung liegen in Linkbibliotheken bereit. Für neue Hardware kann eine vorhandene Steuersoftware durch ein Device-Interface einfach integriert werden.

Loop

Die Endlosschleife Loop ist für alle Steuerungen gleich aufgebaut. In dieser Schleife werden die konfigurierten Devices zeit- oder programmgesteuert aufgerufen. Die Implementierung von Loop ermöglicht die sichere Fensteuerung und bietet umfangreiche Debugmöglichkeiten.

Devices

Devices sind Programmmodule die direkt auf die Hardware oder auf andere Devices zugreifen.
Aufgaben für Devices: Sensoren lesen, Aktoren steuern, Programme für Sensoren/Aktoren ausführen, Infos anzeigen, Logdatei schreiben, Termine verwalten usw.

Die Device Struktur verfügt über eine Device-Definition und Laufzeitinformationen. Die Device-Definitionen werden beim Start der Steuerung aus einer Konfigurationsdatei gelesen. Siehe Devices testen.

Alle Devices haben eine einheitliche Kommunikations-Schnittstelle DevCtl zum Aufrufen der verschiedenen Device-Funktionen. Damit können sowohl Events als auch Devices miteinander kommunizieren. Siehe Device Ctl Funktion.

Events

Zum Aufruf von Devicefunktionen werden Events verwendet. Events beschreiben, welche Devices wann und wie in der Steuerungs-Loop() aufgerufen werden. Jedes Event ist dazu mit genau einem Device verlinkt.

Die Events werden von Loop() fortlaufend alle WHSec>0 geprüft. Events mit mit WHSec=0 können von anderen Events/Devices aufgerufen werden. Für termingesteuerte Events gibt es ein programmierbares TIMER Device.

Die Event Struktur verfügt über eine Event-Definition und Laufzeitinformationen. Die Device-Definitionen werden beim Start der Steuerung aus einer Konfigurationsdatei gelesen. Siehe Events testen.

Das Device beschreibt die eigentliche Funktion eines Events. In einer Steuerung bilden die Events die fixen Elemente. Die Devices können sofort durch ähnliche Typen ersetzt werden.

devtest

Mit Programm devtest können alle Steuerung mit dem Konzept Loop, Events und Devices getestet werden. Beim Aufruf können die Konfigurationsdateien ins Testprogramm übernommen werden.

Aufruf von devtest:

```
pc780mint:~$ devtest -h         // Help für devtest

Aufruf: devtest [OPTIONS]
OPTIONS:
  -m         mtrace on, Speicherüberwachung aktivieren
  -h         Help
  -E EVENTS  Path für Eventkonfiguration
  -D DEVICES Path für Devicekonfiguration
  -n         no Wait, keine Anzeige beim Start
  -d         Debug
```

Siehe auch: Devices testen , Events testen , Loop testen , Schnittstellen testen

Beispiel: Heizung für ein Glashaus

Das Konzept Loop, Events und Devices soll am Aufbau einer einfachen Heizung für ein Frühbeet erklärt werden. Die genaue Beschreibung der Stukturen folgt in den Kapiteln Loop, Event und Device .

Vorgaben

Aufgabestellung: Ein Temperatursensor im Glashaus.
Ein Heizelement. Gesteuert mit einem Relais.
Ab einer wählbaren Frostgrenze soll das Heizelement eingeschaltet werden.

Erweiterung: Bewässerung an bestimmten Wochentagen.

Lösungsweg

Im Folgenden wird beschrieben, wie die Steuerung aus Devices und Events zusammengestellt werden kann. Die Devices und Events werden mit dem Programm devtest einzeln und mit der bereits vordefinierten Loop() getestet.

Für bereits programmierten Devices aus der Linkbibliothek werden die Aufrufparameter in die Konfigurationsdatei eingetragen. Neue Hard- oder Softwaredevices müssen eventuell programmiert werden. Alle definierten Devices und GPIO-Schnittstellen können dann mit devtest einzeln ohne Steuerung getestet werden.

Zum Aufruf der definierten Devices werden danach passende Events in der Konfigurationsdatei definiert. Mit devtest kann das Zusammenspiel von Events, Devices und Loop wieder ausführlich getestet werden.

Die Funktion der Steuerung wird dann durch die zwei Konfigrationdateien für Devices und Events vollständig beschrieben. Im Beispiel: test.events und test.devices.

Ein eigenständiges Programm für diese Steuerung erfordert dann noch Startfunktionen für die gewünschte Laufzeitumgebung. Es folgt eine Übersicht der erforderlichen Schritte am Programmbeispiel gardenctl.

Beispiel: gardenctl im Ordner c/pi/bin/gardenctl

Die Steuerung gardenctl soll autonom laufen und kann daher manuell, automatisch oder über Fernbedienung gestartet werden.
Die Namen der Sourcdateien folgen dem fixen Richtlinen von C-Projekt.

Die ersten Schritte:
- ▷ Das Programmgerüst von gardenctl mit Programm newprg in c/pi/bin/ anlegen
- ▷ Linkbiliotheken einbinden: Die benötigten relativen symbolischen Links von devtest in den Ordner c/pi/bin/gardenctl kopieren
- ▷ makefile anpassen

gardenctl.h
- ▷ Definition der globalen Konstanten
- ▷ Deklaration der Var-Bezeichner
- ▷ Deklaration der gewünschten run Funktionen

gardenctl.c
- ▷ StartCheck() anpassen
- ▷ Init() Funktion für das Programms
- ▷ LoopRun() Loop starten
- ▷ Exit() Funktion für das Programm
- ▷ Wartungsmenu

run.c
- ▷ Implementierung der run Funktionen
- ▷ garden.config

Loop

Das Objekt loop.h/loop.c findet man in der Linkbiliothek c/pi/bininc/events/loop.*

Das Steuerungsprogramm befindet sich normalerweise immer in der zentralen Verabeitungsschleife Loop(). Beim Programmstart wird zuerst immer über LoopRun(...) die Loop() und nicht das Wartungsmenu gestartet. Der Benutzer kann danach das Terminal mit laufender Steuerung beenden oder die Steuerung stoppen und ins Wartungsmenu wechseln.

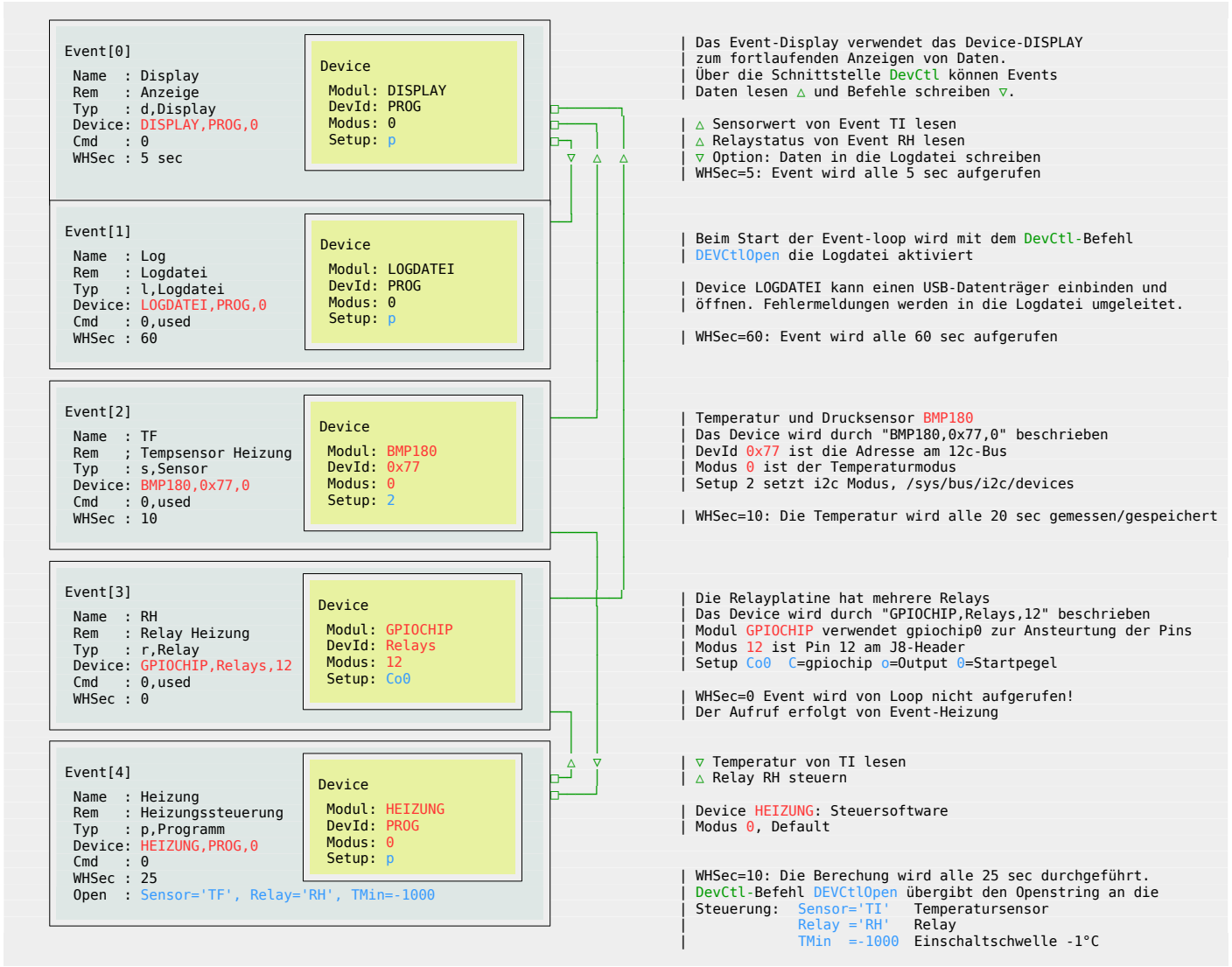
In LoopRun() werden die Konfigurationsdateien für Events und Devices geladen und initialisiert. Danach wird Loop() gestartet.

Loop:	while(true)	
	▷ LoopSetTime()	Aktuelle Uhrzeit oder Simulationszeit bestimmen
	▷ EventExecAndSetCheck()	Device-Exec Funktionen ausführen und die nächste Werte für CheckTime bestimmen
	▷ WaitSec=EventGetWaitSec()	WaitSec ist die kleinste CheckTime der wartenden Events
	▷ Taste=chkTasteSec(WaitSec)	Maximal WaitSec auf eine Taste warten
		Taste==taste_nil keine Taste in WaitSec, continue, Event behandeln
		Taste==taste_select Device-Interrupt behandeln. Siehe Programm picamclt
		Taste==Befehl Befehl ausführen
	▷ switch(Taste)	Taste auswerten

Events und Devices

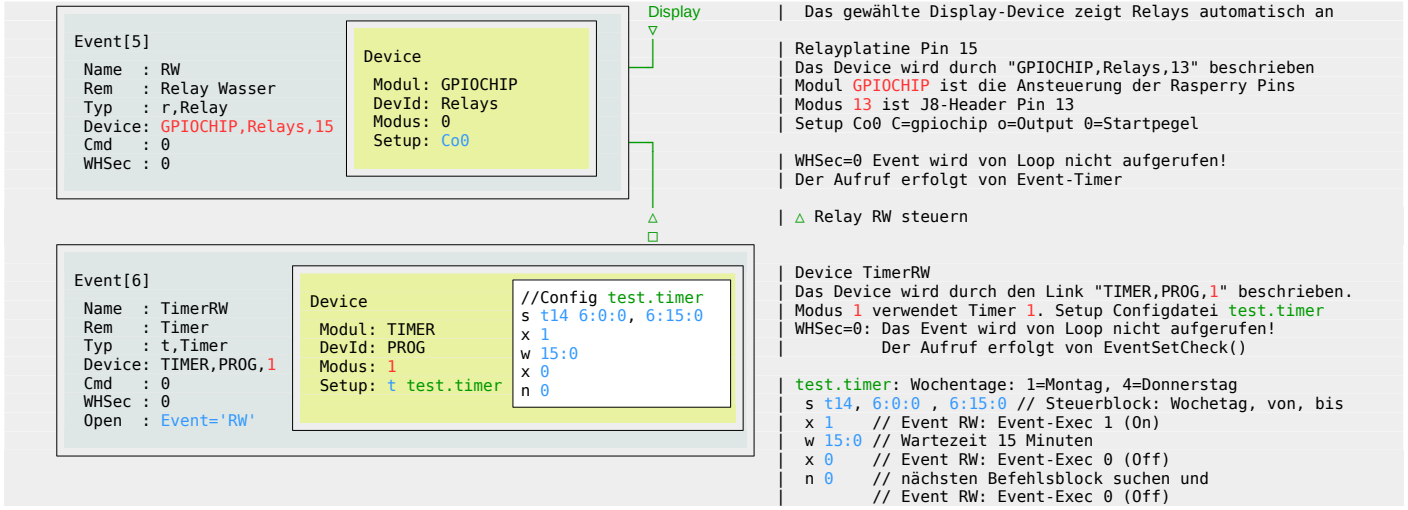
Das Objekt events.h/events.c findet man in der Linkbibliothek c/pi/bininc/events/events.*
Die genaue Beschreibung der verwendeten Stukturen folgt in den Kapiteln Event und Device .

Die konfigurierten Events werden ins Array EventLst[ ] und die Devices ins Array DevLst[ ] geladen. Die Zuordnung vom Event zum Device erfolgt über den String Event.Device: z.B. "GPIOCHIP,Relays,12" Bedeutung: Software gpiochip0, Modul Relay, Pin 12



Erweiterung

Bewässerung mit Timer steuern.



Steuerung testen

Mit dem Hilfsprogramm devtest können Konfigurationsdateien für Devices und Events von Steuerungen eingelesen und getestet werden.
Siehe Devices testen , Events testen , Loop testen , Schnittstellen testen

Devices

Device-Module beschreiben die einzelne Aktionen der Steuerung:

- Sensorwert lesen
- Relay schalten
- Interrupt auslösen
- Steuerungsprogramm ausführen
- Timer
- Anzeige aktualisieren
- usw.

Devices für die Heizung

Es werden im Beispiel folgende Device-Module verwendet:

```
Modul="BMP180",   // Sensor BMP180 , Druck und/oder Temperatur
                  // Diese Modul kann durch einen anderen Sensor ersetzt werden

Modul="HEIZUNG",  // Modul Steuerung: Temperatur mit Frostgrenze vergleichen
                  //                  Im Frostfall Relay einschalten

Modul="GPIOCHIP", // Modul: Heizungs Relay mit gpio schalten

Modul="DISPLAY",  // Modul: Ergebnisse im Terminal anzeigen
```

Diese Device-Module werden im Beispiel in der Konfigurationsdatei test.devices vereinbart. Details siehe Konfiguration der Devices.

Dev Objekt

Das Objekt devices.h/devices.c findet man in der Linkbibliothek c/pi/bininc/events/devices.*

Mit dem Dev-Objekt werden alle Devices angelegt. Beim Anlegen mit DevNew(DevConf) wird die Konfigurationsdatei gelesen und ausgewertet.

Im Beispiel werden die gewünschten Devices werden aus der Konfigurationsdatei test.devices in das Array DevLst[ ] vom Typ tDev eingelesen. Geräte mit Modus<0 werden übersprungen.

```
typedef struct tDev       // Devicedefinition für Objekt-Dev
{
  //readonly:             // Startparameter aus der Konfiguration
  char       *Modul;      // eindeutiger Modulname aus ModulLst[]
  char       *Rem;        // Beschreibung des Moduls
  char       *DevId;      // eindeutige spezielle Device-Id
  //                      // Bsp: Chip-ID oder 12c-Device-Id
  char       *Setup;      // Setupstring des Geräts                          | Das erste Zeichen ist der Interface-Typ
  int16_t     Modus;      // Wunschmodus für das Device:
  //                      //   -1  Device nicht verwenden
  //                      //    m  Pin m oder Deviceoption m verwenden
  // ------------------------------------------------------------------------
  // Programm-Parameter
  //
  // readonly: Steuerfunktion für das Modul. Wird mit dem Modulnamen
  tFnkDevCtl  DevCtl;     // aus der ModulLst[] in devicesm.c bestimmt       | Befehlsfunktion für das Device
  //
  // Read/write:          // Rückgabeparameter von Device
  int16_t     Status;     // tatsächlicher Device-Status zum Wunschmodus
  //                      // Rückgabe von DevCtl( DEVCtlOpen, Dev)
  //                      //    -1: Device nicht in Verwendung
  //                      //     m: tatsächlicher Device-Status
  //                      //        wird in Device-open bestimmt
  // ------------------------------------------------------------------------
  // Read/write:          // Parameteraustausch mit dem Device
  int32_t     ComInt;     // In/Out Deviceparameter. Ungültig: DEVCtlNil     | Com-Schnittstelle
  char       *ComStr;     // In/Out String. Nur temporäre Strings!           | Com-Schnittstelle
}tDev;
```

Jedes Device hat eine einheitliche DevCtl-Schnittstelle. Dadurch könne alle Devices der Steuerung problemlos ausgetauscht werden.

DevCtl Schnittstelle

Das Objekt Dev bietet für jedes Devices einheitliche Befehlsaufrufe DEVCtlxxx.
Folgende DEVCtlxxx Befehle können immer verwendet werden:

```
#define DEVCtlHeader 'h'  // Header        | Dateiname des Device-Headers
#define DEVCtlRem    'R'  // Remark        | Beschreibung zum Device
#define DEVCtlOpen   'o'  // Open   Device | Open  : Device in Betrieb nehmen/initialisieren
#define DEVCtlClose  'c'  // Close  Device | Close : Device ausschalten
#define DEVCtlRead   'r'  // Read   Device | Read  : Daten vom Device nach ComInt und/oder ComStr lesen
#define DEVCtlWrite  'w'  // Write  Device | Write : Daten von ComInt oder ComStr zum Device schreiben
#define DEVCtlExec   'x'  // Exec   Device | Devicefunktion mit Parameter ComInt aufrufen
#define DEVCtlInfo   '?'  // Infos         | Deviceinfos
```

```
#define DEVComIntNil INT32_MIN          //| Ungültiger int32_t Wert für ComInt
```

Alle Devices können die oben gelisteten DEVCtlxxx Befehle mit mit der Funktion DevNCtl(...) aufrufen. Die Parameter ComInt und ComStr werden beim Aufruf übergeben. Rückgaben können mit DevNComInt() und DevNComStr() abgefragt werden. Erfolgreiche Aufrufe geben immer den Device-Index für DevLst[ ] zurück. Im Fehlerfall NIL.

```
int32_t DevNCtl(uint32_t DeviceN , char Ctl, int32_t ComInt, const char *ComStr);
   // Für Device DevLst[DeviceN] die Steuerfunktion mit Befehl Ctl mit den Parametern ComInt und ComStr aufrufen.
   // Rückgabe: DeviceN bei Erfolg oder NIL für Fehler
   //           ComInt    mit DevNComInt() abfragen
   //           ComStr    mit DevNComStr() abfragen
```

Mit diesem Konzept können in einem Steuerprogramm die Devices einfach getauscht werden. Zum Beispiel kann ein Temperatursensor durch einen ganz anderes Modell ersetz werden.

Die DevCtl Schnittstelle für den Druck- und Temperatursensor BMP180 wird in Kapitel Beispiel:Device BMP180 beschrieben.

Konfiguration von Devices

Aufbau der Konfiguration test.devices :

```
// ---------------------------------------------
// devtest : Konfiguration für Devices
// Beispiel: Heizung für ein Glashaus
// Datum 2025-01-07

Devices[]=
{{ Modul="DISPLAY",         // Terminalanzeige                  | Modulname aus Modulliste
   DevId="PROG",            // Programm-Modul                   | Device-Id: Software
                            //                                  | Optionen: Chip-ID, 12c-Device-Id oder PROG
   Rem  ="Anzeige",         //                                  | Beschreibung des Moduls
   Setup="p l",             // Setupstring, Programm, l Led     | Interface-Typ: p Software, Option: l Pi Led schalten
   Modus=0,                 // Wunschmodus                      | Gewünschter Devicemodus
                            // -1                               | Device nicht verwenden
                            //  n                               | Device n, Devicemodus n oder Pin n verwenden
 },

 { Modul="BMP180",          // Sensor BMP180                    | Der Sensor BMP180 hat zwei Modi (Druck und Temperatur) und
                            //                                  | kann über verschiedene GPIO-Interfaces angesteuert werden.
   DevId="0x77",            // Id für /dev/i2c                  | i2c Adresse
   Rem  ="Temp Heizung",    // Remark: Beschreibung
   Setup="2",               // Interface /dev/i2c-1 mit ioctl() | 1.Zeichen '2', i2c Ansteuerung über /dev/i2c-1 mit ioctl()
   Modus=1,                 // 1 Temperatur, 2 Druck            | Temperaturmodus
 },

 { Modul="HEIZUNG",         // Modul: Steuerung                 | Software: Liest die Temperatur von Modul="BMP180" und steuert
                            //                                  | das Relay mit Modul="GPIOCHIP". Die Terminalanzeige erfolgt
                            //                                  | mit Modul="DISPLAY"
   DevId="PROG",            //                                  | Device-Id:Software
   Rem  ="Prog Heizung"
   Setup="p",               //                                  | Interface-Typ oder p Software, t Timer
   Modus=0,                 // verwenden
 },

 { Modul="GPIOCHIP",        // Modul: I/O-Steuerung             | GPIO Interface gpiochip0 für Raspberry Pi
   DevId="Relay",           //                                  | Device-Id: Relay
   Rem  ="Relay Heizung"
   Setup="Co0",             // 'C' Chip, 'o' Output, '0' Startwert | 1.Zeichen 'C' Interface gpiochip0, "o0" Setupstring
   Modus=12,                // Pinnummer vom J8-Header          | J8 Pinnnummer für gpiochip0
 },
}
```

Modul

Der Modulname steht für eine Deviceklasse und ist daher nicht eindeutig. Ein Device wird durch die drei Felder Modul, DevId und Modus umkehrbar eindeutig definiert!

Beispiel: Das Modul GPIOCHIP bedient den J8-Header vom Rasperry Pi. Das Device "GPIOCHIP, Relay, 12" ist dann das Relay an Pin 12.

DevId

Die DevId's sind spezielle Angaben zum identifizieren eines Devices.

Beispiele:		
	"PROG"	für Steuer-Programme
	"Relay"	Device Typ
	"10-000802e444d1"	1-Wire Id von /sys/bus/w1/devices, Chip-Kennung
	"0x77"	I2C Adresse

Setupstring

Beispiel: Setup für Device Relay:

```
Modul : GPIOCHIP             | Modulname beschreibt den Geraetetyp. Nicht eindeutig!
DevId : Relays               | Device-Id: PROG, 1-Wire-ID, Chip-ID oder 12c-Device-Id
Setup : "Co0               " | Setupstring für Gerät. Setup[0] ist der Interface-Typ
Rem   : Relay Heizung        | Kurzinfo zum Modul
Modus : 12                   | Gewünschter Gerätemodus, tatsächlicher Modus 'Status'
```

Setup:	Co0	C Interface gpiochip0, o Output-Pin, 0 Startpegel 0
Modus:	12	Pin 12 verwenden

Aufbau des Setupstrings für GPIO-Devices.

Das erste Zeichen im Setupstrings ist der Interface-Typ aus gpioci.h:

```
Interface-Typ:

       #define GPIOIfTypChip        'C'  // Ansteuerung über /dev/chipiochip0
       #define GPIOIfTyp1Wire       'W'  // Ansteuerung 1-Wire direkt
       #define GPIOIfTyp1WireSys    'S'  // Ansteuerung über /sys/bus/w1/devices
       #define GPIOIfTypI2c         '2'  // Ansteuerung über /dev/i2c-1 mit ioctl()
       #define GPIOIfTypI2cSys      'I'  // Ansteuerung über /sys/bus/i2c/devices
       #define GPIOIfTypProg        'p'  // Interface Programm
       #define GPIOIfTypTimer       't'  // Interface für Timer-Device
```

Die folgenden Zeichen des Setupstrings beschreiben Deviceeinstellungen.

Für Pins aus gpioci.h sind folgende Mehrfachangaben dabei erlaubt:

```
       #define GPIOPinSetOut        'o' // Output-Pin
       #define GPIOPinSetIn         'i' // Input-Pin
       #define GPIOPinSetLogikP     'p' // Default, positive Logik
       #define GPIOPinSetLogikN     'n' // negative Logik
       #define GPIOPinSetOpenDrain  'd'
       #define GPIOPinSetOpenSource 's'
       #define GPIOPinSetDefOut1    '1' // Startpegel 1, Default:GPIOPinPegelErr
       #define GPIOPinSetDefOut0    '0' // Startpegel 0, Default:GPIOPinPegelErr
       #define GPIOPinSetEvent01    'r' // Event 0-1 Flanke, rising
       #define GPIOPinSetEvent10    'f' // Event 1-0 Flanke, falling
```

Device Timer Setup:

```
       Setup="t timer1.tim" //  timer1.tim Timer Konfiguration
```

Modus

Modus beschreibt den gewünschten Betriebsmodus des Devices. Die möglichen Werte hängen vom jeweiligen Gerät ab.

Beispiele:

Device BMP180:	1 für Temperatur, 2 für Druck
Device Relay:	Modus ist die Pinnummer

Modus=-1 steht für Device nicht verwenden.

Jedes Device wird durch die drei Felder Modul, DevId und Modus umkehrbar eindeutig definiert!

DevNCtl Funktion

Aus dem Modulnamen und dem Interface-Typ (das Zeichen des Setup-Strings) des Devices, wird die passende Steuerfunktion DevClt() (und DevNCtl()) ermittelt.

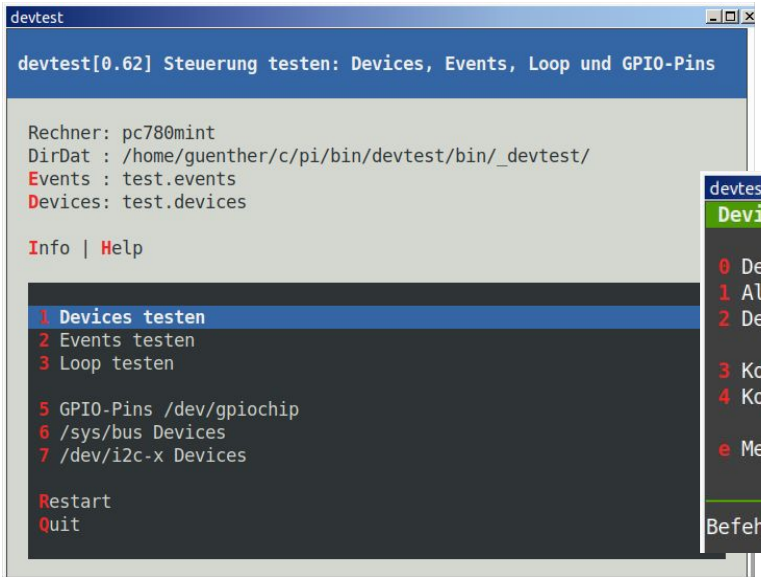
In der Programmdatei devicesm.c werden die Steuerfunktionen und C-Header in der Modulliste ModulLst[] deklariert.
Beschreibung in Kapitel Beispiel:Device BMP180

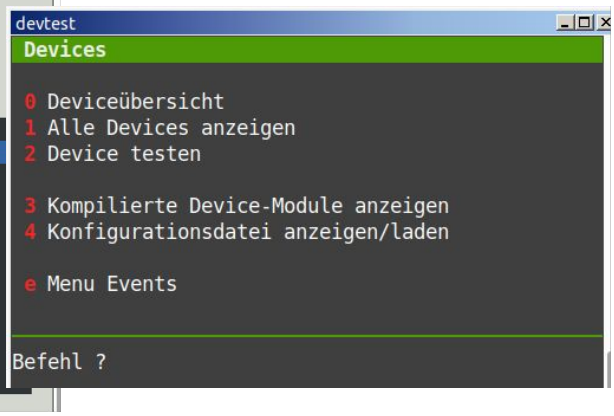
```
// ------------------------------------------------------------------------
// // C-Header der eingebunden Device-Programme
//
 #include "devdisp.h"   // Programm-Modul     | Anzeige
 #include "devlog.h"    // Programm-Modul     | Logdatei
 #include "bmp180i.h"   // i2c-Interface      | Druck- und Temperatursensor
 #include "ds1820s.h"   // SBus-Device, 1Wire | Temperatur
 #include "relayc.h"    // Goio-Chip Device   | Rasperry I/O
 #include "devprog.h"   // Steuerprogramme    | Heizung
 #include "devtimer.h"  // Programm-Modul     | Device-Timer
// ------------------------------------------------------------------------
// Modulliste
//
tModulLst ModulLst[]=
{ //        Modulname          Interface-Typ               Steuerfunktion
  { .Modul="DISPLAY",  .IfTyp= GPIOIfTypProg,     .DevCtl= DispDevCtl    },
  { .Modul="LOGDATEI", .IfTyp= GPIOIfTypProg,     .DevCtl= LogDevCtl     },
  { .Modul="BMP180",   .IfTyp= GPIOIfTypI2c,      .DevCtl= Bmp180iDevCtl },
  { .Modul="DS1820",   .IfTyp= GPIOIfTyp1WireSys, .DevCtl= DsDevCtl      },
  { .Modul="GPIOCHIP", .IfTyp= GPIOIfTypChip,     .DevCtl= RelayDevCtl   },
  { .Modul="HEIZUNG",  .IfTyp= GPIOIfTypProg,     .DevCtl= HeizungDevCtl },
  { .Modul="TIMER",    .IfTyp= GPIOIfTypTimer,    .DevCtl= TimerDevCtl   },
  { .Modul=NULL }
};
```

Siehe Testprogramm devtest: Befehl: 1 Devices testen/ 0 Deviceübersicht

Devices testen

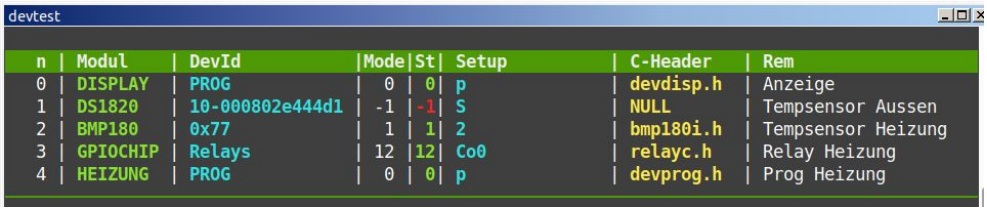
Mit Programm devtest können die Events und Devices aus events_test.conf und devices_test.conf eingelesen und mit der Com-Schnittstelle getestet werden. Events und die Konfigurationsdatei events_test.conf werden im nächsten Abschnitt erklärt.





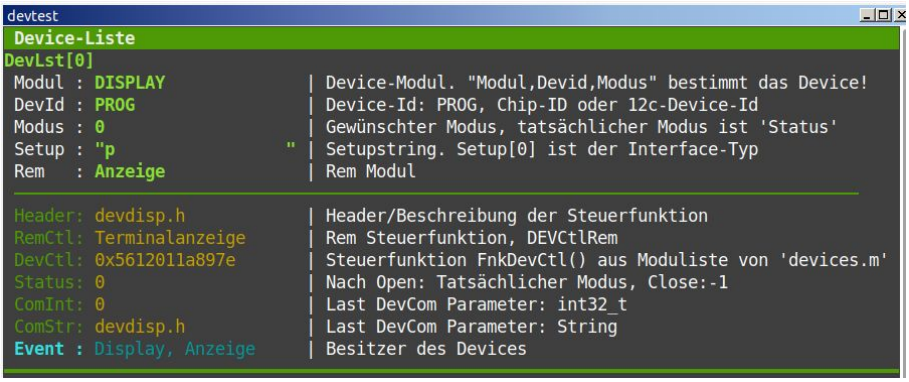
Befehl: 1 Devices testen/ 0 Deviceübersicht

Modul: Device Name
DevId: Id für das Device
Mode: Devicemodus. -1 ausgeschaltet
Status: Tatsächlicher Modus
Setup: Setupstring
1. Zeichen: Interface
Folgezeichen: Deviceeinstellungen



Befehl: 1 Devices testen/ 1 Alle Devices anzeigen

Die grünen Werte stammen aus der Konfiguration. Die gelben Werte wurden berechnet.



Befehl: 1 Devices testen/ 2 Device testen

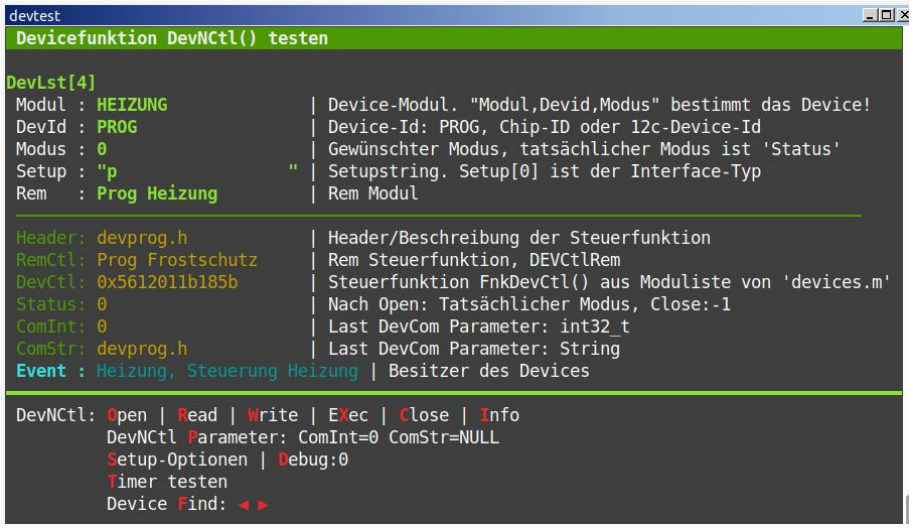
DevNCtl Befehle:

Open	Device initialisieren
Read	Daten lesen
Write	Daten schreiben
EXec	Daten berechnen
Close	Device freigeben
Info	Infos zum Device

Mit diesen Befehlen kann das Device gesteuert oder getesten werden.

Mir den Parametern ComInt und ComStr können Daten ausgetauscht werden.

Debug Detailierte Ausgaben



Events

Für den zeitgesteuerten Aufruf von Devices in einer Steuerungs-Loop werden Events definiert. Events können zeitunabhängig aber auch von anderen Events aufgerufen werden.

Event Objekt

Das Objekt events.h/events.c findet man in der Linkbiliothek c/pi/bininc/events/events.*

Objekt Event verwaltet alle Events. Beim Anlegen des Objekts mit EventDevNew() wird die Event-Konfiguration test.events und die Device-Konfigurationen test. devices gelesen und die Events und Devices angeleft. Danach werden die Events mit den passenden Devices verlinkt.

Event Struktur:

```
typedef struct tEvent  // Private Beschreibung eines Events
{ //Die folgenden Felder werden aus der Konfigurationsdatei gelesen .........
  //Eventstruktur 'EventDef' siehe eventsedit.c

  char     *Name;     // Eindeutiger, kurzer Eventbezeichner ohne Blanks
  char     *Rem;      // Bemerkung, Kurzbeschreibung
  tEventTyp Typ;      // Event-Typ: Sensor, Relay, Programm                    | Siehe unten: tEventTyp
  char     *Device;   // NULL oder eindeutige Decice-Link: "Modul,DevId,Modus"  | Eindeutiger Link-String zum Device
                      // z.B."BMP180,0x77,0"
  char     *Open;     // NULL oder Parameterstring für Funktion DEVCtlOpen
  int16_t   Cmd;      // Event-Exec Wunschparameter. -1: Event nicht verwenden
  uint16_t  WHSec;    // Event-Exec alle WHSec Sec Wiederholen

  //Die folgenden Felder werden von Event berechnet .....................     | interne Laufzeitinformationen
  int16_t   CmdExec;  // Kopie von Cmd. -1: Gerät wird momentan nicht verwendet | tatsächlicher Event-Exec Parameter
  uint32_t  DevN;     // NIL oder Device-Index für DevList[].
  time_t    CheckTime;// Checkzeitpunkt: 0      Event-Exec nicht verwenden     | keine Event-Checks
                      //                 Time   if(CheckTime<=Time) Event-Exec | Termin des nächsten Event-Checks
  int32_t   CheckCmd; // Event-Exec Parameter. -1 Event-Exec nicht aufrufen     | Event-Exec Parameter
  uint32_t  ErrCount; // Fehlerzähler
  uint16_t  Debug;    // >0 Debugmodus nur für dieses Event

} tEvent; // ------------------------------------------------------------------
```

Event Typ:

```
typedef enum tEventTyp     // Art eines Events
 { EvSensor  ='s',         // Sensor read
   EvRelay   ='r',         // Relais ON/OFF
   EvDisp    ='d',         //  Programm, Terminal-Anzeige
   EvLog     ='l',         // Programm, Logdatei schreiben
   EvProg    ='p',         // Steuerungsprogramm
   EvTimer   ='t',         // Timerprogramm
   EvNop     ='n'          // no operation
 } tEventTyp; // ----------------------------------------------------------------
```

Device Link:

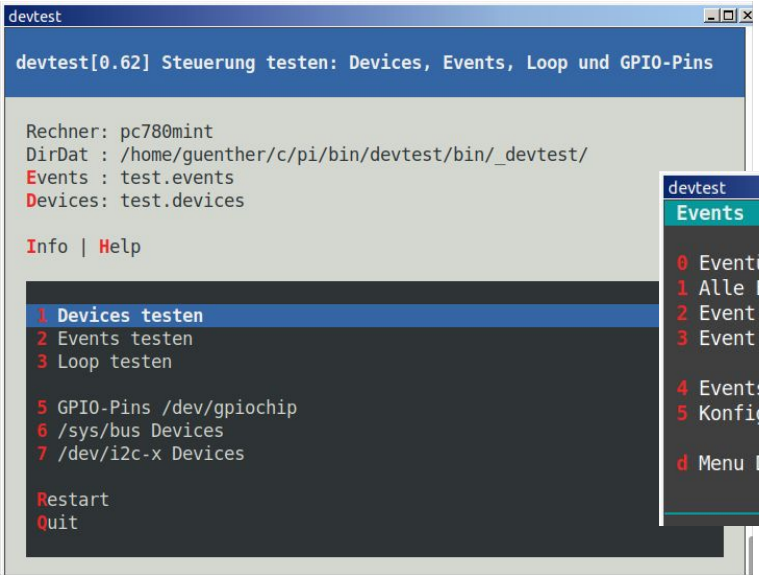
Ein Device wird durch den String "Modul, DevId,Modus" umkehrbar eindeutig definiert!
In devicesm.c werden die benötigten Modul-Header und die Modulliste deklariert:

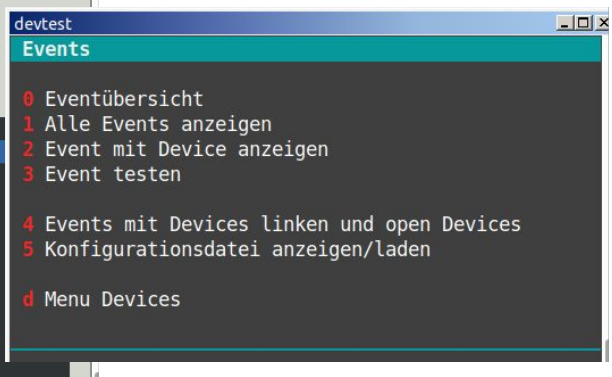
Konfiguration von Events

Beispiel: Die Konfigurationsdatei test.events für Events:

```
// devtest : Konfiguration für Events
// Beispiel: Heizung für ein Frühbeet
// Datum 2025-01-10

Events[]=
{{Name="Display",              // Event-Id, eindeutiger Eventbezeichner   | Event Display: Terminalanzeige
  Rem="Anzeige",               // Remark, Kurzbeschreibung
  Typ="d",                     // 'd' Programm, Terminal-Anzeige
  Geraet="DISPLAY,PROG,0",     // Device:"Modul,DevId,Modus"              | Devicelink Terminalanzeige
  Open="",
  Cmd=0,                       // Wunsch   0: Gerät verwenden,           | Defaultparameter für Event-Exec
                               //         -1: Gerät nicht verwenden
  WHSec=5                      // Wiederholung in Sec                     | Anzeige alle 5 Sekunden erneuern
 },

 // ---------------------------------------------------------------
 // Heizung Frühbeet steuern
 // Events: "TF"      Temperatursensor
 //         "RH"      Relay
 //         "Heizung" Steuerung

 {Name="TF",                    // Event-Id: Temperatursensor        | Event TF: Temperatur lesen
  Rem="Temp Fruehbeet",
  Typ="s",
  Device="BMP180,0x77,1",       // Device:"Modul,DevId,Modus"        | Devicelink: Temperatursensor
  Open="",
  Cmd=0,
  WHSec=10                                                         | Temperatur alle 10 Sekunden bestimmen
 },

 {Name="RH",                    // Event-Id, Relay Heizung           | Event RH: Relay ON/Off schalten
  Rem="Relay Heizung",
  Typ="r",                      // 'r',Relais ON/OFF
  Geraet="GPIOCHIP,Relays,12", // Device:"Modul,DevId,Modus"         | Devicelink: Relay
  Open="",
  Cmd=0,
  WHSec=0                                                          | Event in Steuer-Loop nicht aufrufen! Aufruf von Event Heizung
 },

 {Name="Heizung",                                                  | Event Heizung: Steuerungsprogramm ausführen
  Rem="Heizungssteuerung",
  Typ="p",                      // 'p', Steuerungsprogramm
  Geraet="HEIZUNG,PROG,0",      // Device:"Modul,DevId,Modus"        | Devicelink: Steuerung
  Open="EvSensor='TI', EvRelay='RH', TMin=-1000 ",                 | Device-Open Parameter für die Steuerung
                                                                   | EvSensor='TI': Zugriff der Steuerung auf den Sensor
                                                                   | EvRelay ='RH': Zugriff der Steuerung auf das Relay
                                                                   | TMin   =-1000: Schaltschwelle -1°C einstellen
  Cmd=0,
  WHSec=15                                                         | alle 15 Sekunden aufrufen
}
```

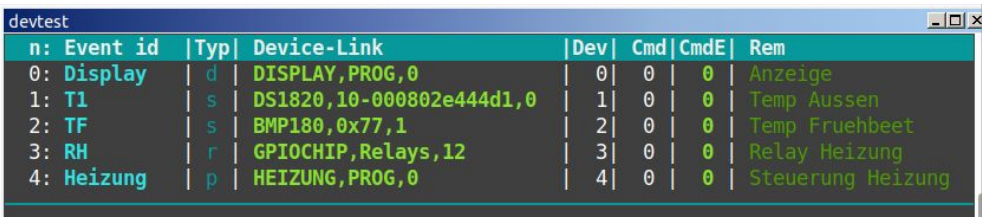
Events testen



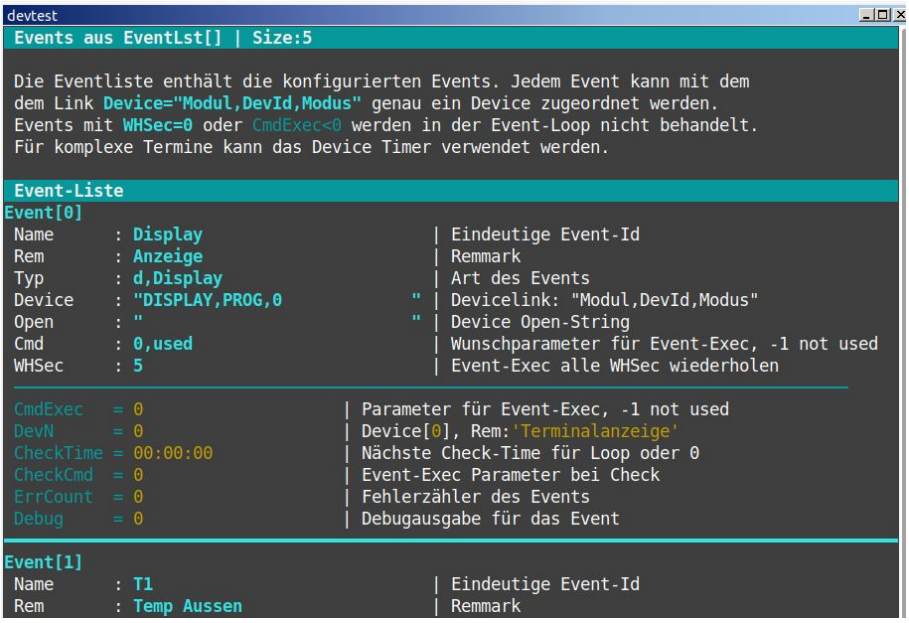


Mit Befehl 2 Events aus dem Hauptmenu von devtest können Events angezeigt und getestet werden.

Befehl: 2 Events testen/ 0 Eventübersicht

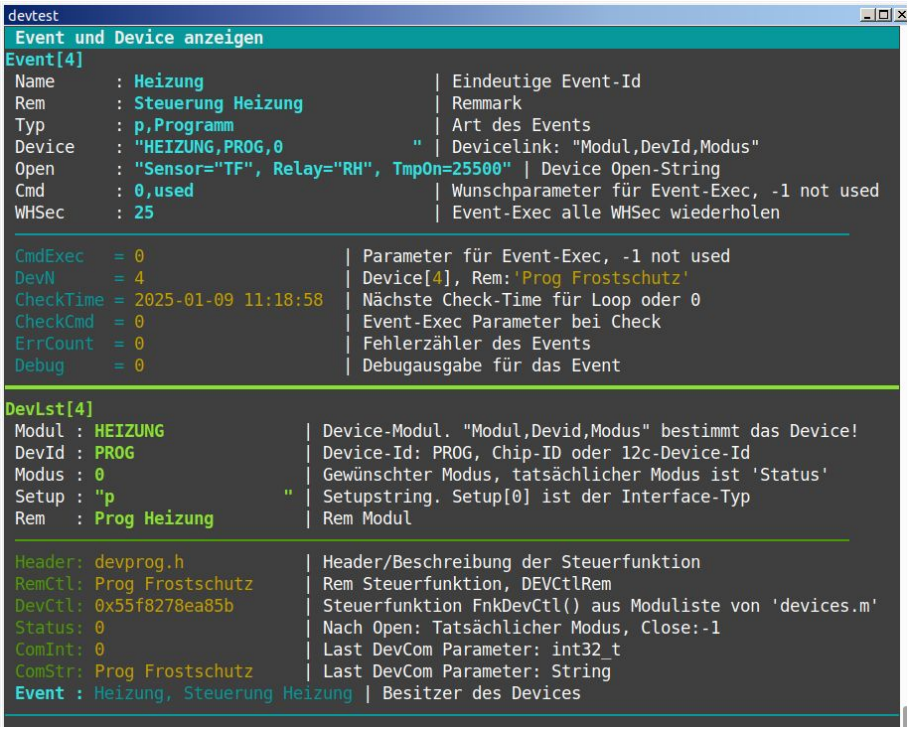


Befehl: 2 Events testen/ 1 Alle Events anzeigen

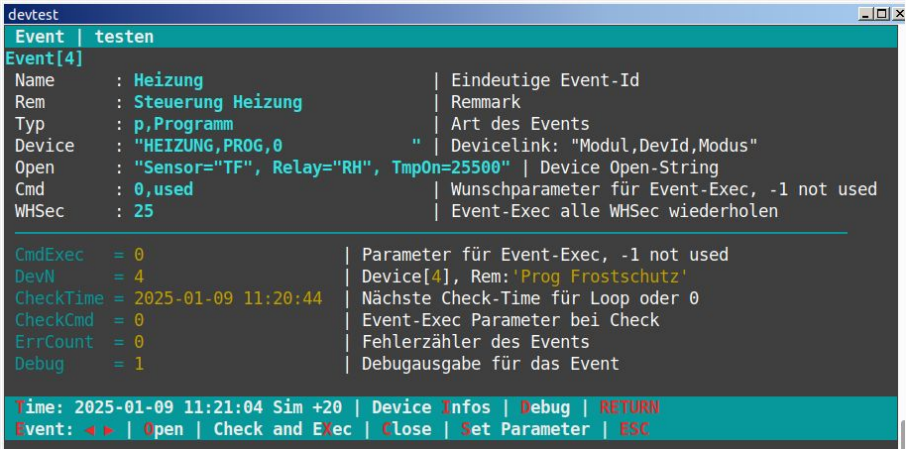


Befehl: 2 Events testen/ 2 Event mit Device anzeigen

Diese Option zeigt die Einstellungen eines Events mit dem zugeordneten Device.



Befehl: 2 Events testen/ 2 Event testen

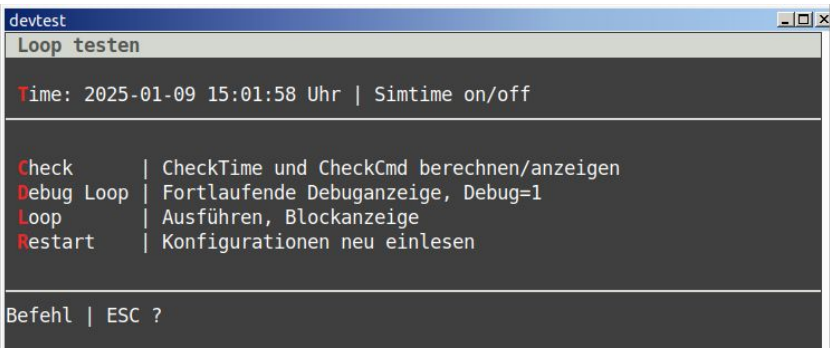


Loop

Loop testen

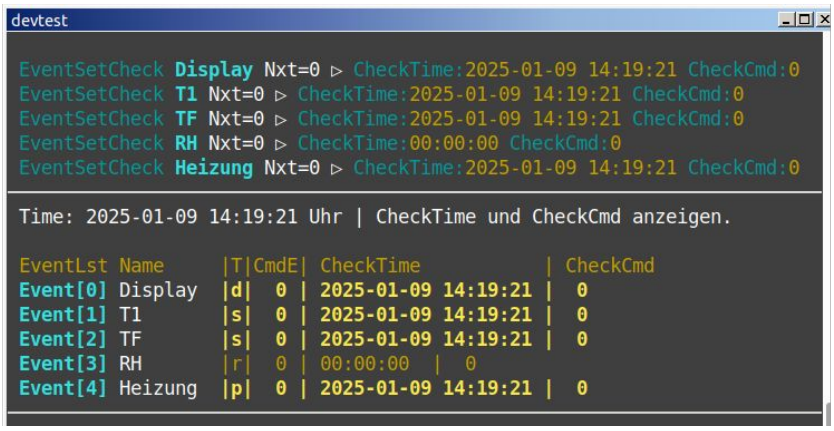
Befehle:

Time Time oder SimTime einstellen



Befehl: Check

Die aktuelle Werte für CheckTime und CheckCmd mit Funktion EventSetCheck() bestimmen.



Warteschlange der Event-Loop.

Die Events-Exec Funktionen werden bei
Time >= CheckTime
mit dem Parameter CheckCmd aufgerufen.

CheckTime=0 : kein Aufruf von Event-Exec!

Danach werden CheckTime und CheckCmd mit Funktion EventSetCheck() neu bestimmt.

Befehl: Debug Loop

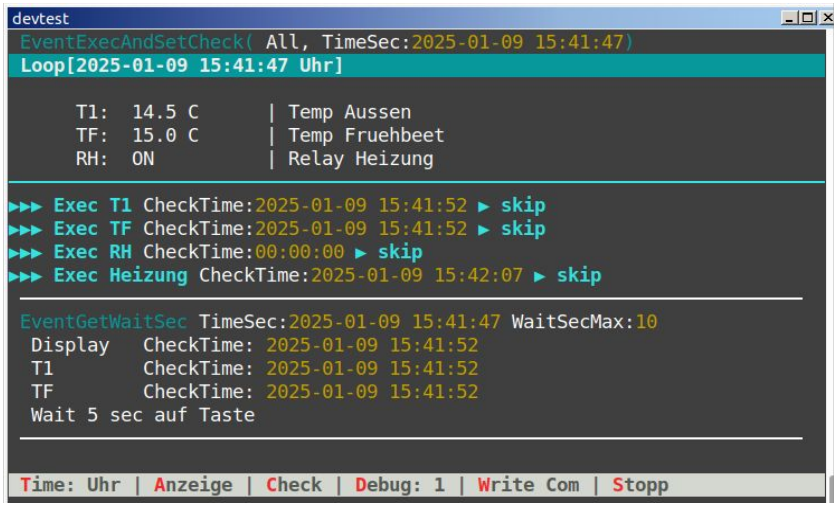
Die Loop() wird mit fortlaufender Anzeige im Modus Debug=1 in Echtzeit gestartet. In diesem Modus werden die Funktionsaufrufe laufend angezeigt.

Loop():
1. EventExecAndCheck() prüft der Reihe nach die CheckTime aller Events.

   Ist die CheckTime erreicht so wird Event-Exec mit Parameter CheckCmd ausgeführt.

   Danach wird mit EventSetCheck() die nächste CheckTime des Events ermittelt.

2. Mit EventGetWaitSec() wird die kleinste Wartezeit bis zum nächsten Aufruf von EventExecAndCheck() bestimmt.



Befehlsoptionen im Debugmodus:

Time	Uhrzeit oder Simulationszeit wählen.
Anzeige	Blockanzeige oder fortlaufende Anzeige
Check	EventSetCheck() und Warteschlange
Debug	Weiterschalten: 0, 1 oder 2
Write Com	Testwert mit Device-Write setzten
Stopp	Loop anhalten

Befehl: Loop Run
Loop im Simulations-Modus mit fortlaufender Anzeige aufrufen.

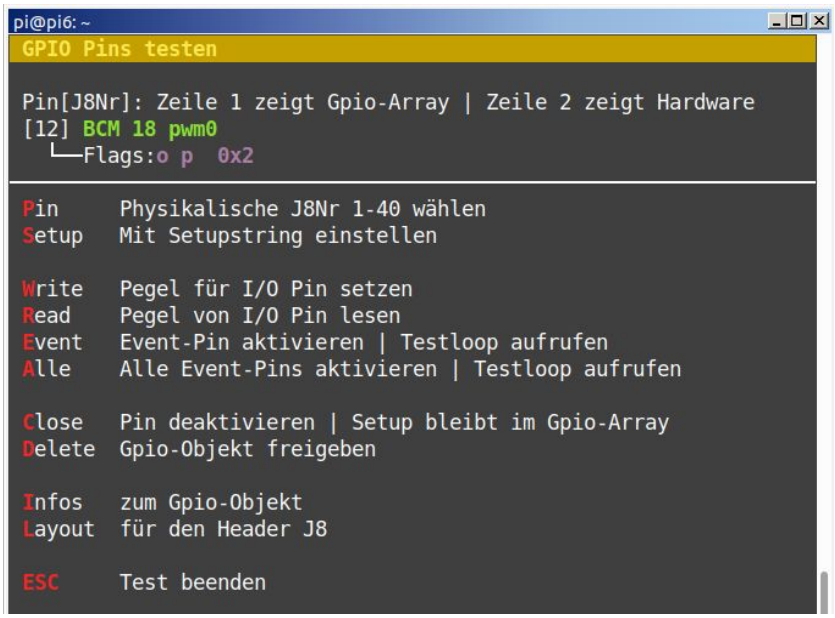
Schnittstellen testen

GPIO-Pins

Menubefehl: 5 GPIO-Pins /dev/gpiochip

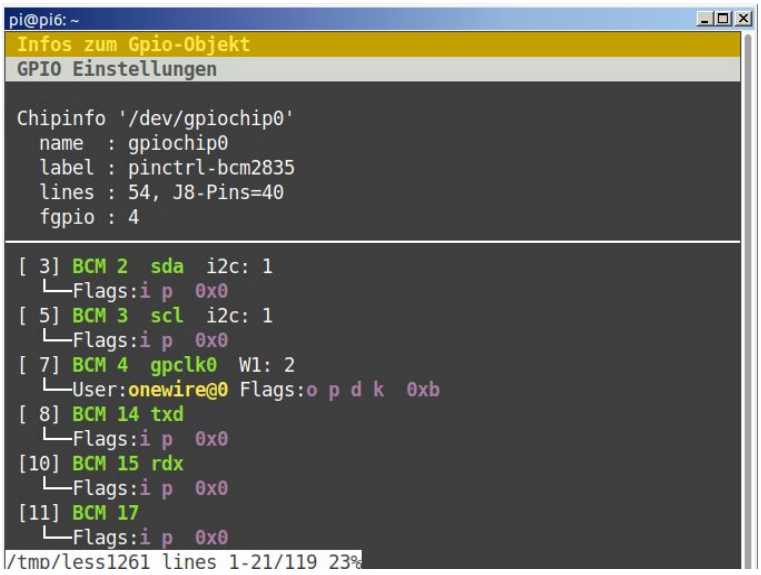
Der Dialog bietet die Möglichkeit die Pins des J8 Headers mit einzustellen und zu testet.

Es wird /dev/gpiochip0 mit dem Opjekt Gpio verwendet. Siehe: c/pi/bininc/gpioci.h .

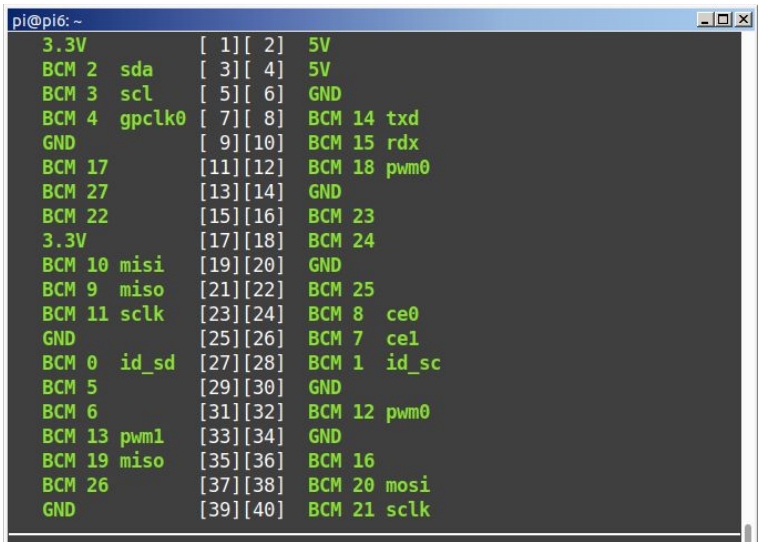


Option: Infos zum Gpio-Objekt

Es werden alle aktuellen Einstellungen des J8 Headers angezeigt,



Option: Layout für Header J8

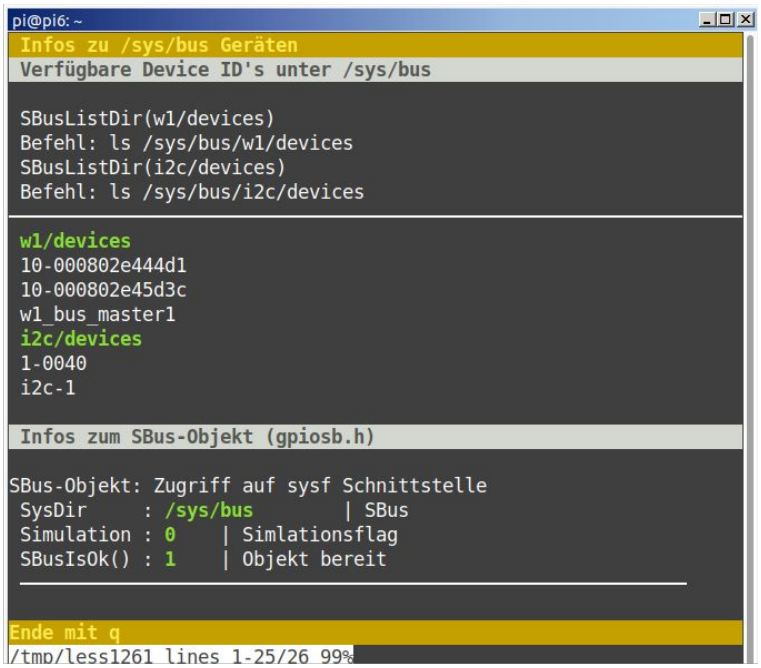


Devices /sys/bus

Menubefehl: 6 /sys/bus Devices

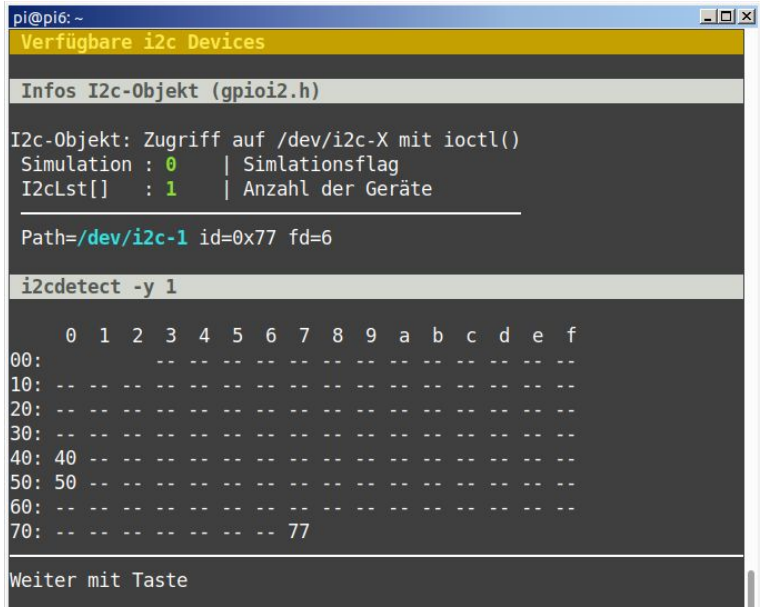
Infos 1-Wire und i2c Devices auf dem /sys/bus.

w1/devices zeigt die aktuellen 1-Wire Geräte.



Devices i2c

Menubefehl: 7 /dev/i2c-x Devices



Datei- und Ordnerübersicht

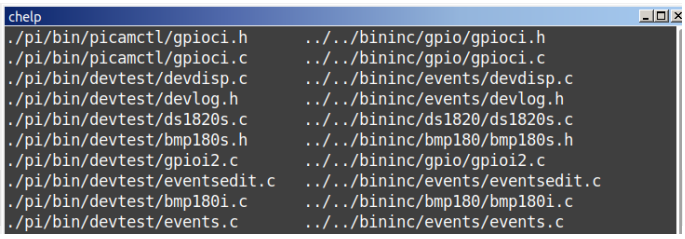
Linkbibliotheken

Das Programm devtest verwendet Funktionen aus der Linkbibliothek c/pi/bininc/. Die Header und Sourcedateien der Funktionen werden dabei immer über relative symbolische Links eingebunden.

Die Linkbilothek bietet beim Entwickeln von unterschiedlichen Steuermodulen Vorteile gegenüber einer C-Bibliothek. Das Testen der einzelnen Module kann in einem beliebigen Steuerprogramm erfolgen. Das Ergebnis steht dann sofort allen anderen Steuerprogrammen zur Verfügung.

Für transportable Ergebnisse müssen die Links unbedingt realtiv und symbolisch sein. Die Bibliotheklinks können mit dem Programm chelp überprüft werden. Befehl: Einstellungen / Projektlinks prüfen / Pfade der symbolischen Links in c/ prüfen

Das Bild zeigt einen Auszug der geprüften Links.
Relative Links sind am Pfad '../../' zu erkennen.
Fehlerhafte links werden rot angezeigt.



Beispiel: Einen relativen symbolischen Link für die Linkbibliothek sicher loop.c anlegen.

```
cd c/pi/bin/devtest                in den Programmordner wechseln
ls ../../bininc/events/loop.c      Linkpfad testen
ln -si ../../bininc/events/loop.c  symb. Link interaktiv anlegen
```

Dateien

Die aktuellste Dateiübersicht findet man in 1_read.me

```
/pi/bin/
    ├── devtest/                      Projektverzeichnis
    │   │
    │   ├── bin/                      Ordner Programm
    │   │   ├── _devtest/             Ordner Konfiguration
    │   │   │   ├── 1_read.me         Diese Hilfedatei
    │   │   │   ├── 2_devtest.odt     Dokumentation
    │   │   │   ├── 2_devtest.pdf
    │   │   │   │
    │   │   │   ├── 20220329.log      Logdateien, Option
    │   │   │   │
    │   │   │   ├── devtest.conf      Konfiguration für devtest
    │   │   │   ├── devtest.devices   Konfiguration für Devices
    │   │   │   ├── devtest.events    Konfiguration für Events
    │   │   │   └── ...
    │   │   │
    │   │   └── devtest               fertiges Programm
    │   │
    │   ├── devtest.h     Globale Definitionen
    │   ├── devtest.c     init(), main(), Exit()
    │   │
    │   ├── run.c           Dialog- uns Hilfsfunktionen
    │   │
    │   ├── devicesm.c      Modulverzeichnis. Liste der verfügbaren
    │   │                   C-Programme für Devices
    │   │
    │   ├── makefile        makefile
    │   │
    │   │   ____________________________________________________________
    │   │   Linkbibliotheken aus 'c/pi/bininc/...' Die Dateien dieser
    │   │   Bibliothek werden über relative Links in die Programme eingebunden.
    │   │
    │   ├── @loop.h         Zentrale Steuerungsschleife für Events
    │   ├── @loop.c
    │   │
    │   ├── @events.h       Verwaltungsfunktionen für Events
    │   ├── @events.c
    │   ├── @eventsedit.c   Dialogfunktionen für Events
    │   │
    │   ├── @devices.h      Verwaltungsfunktionen für Devices
    │   ├── @devices.c
    │   │
    │   ├── @devdisp.h      Device DISPLAY: Anzeige im Terminal
    │   ├── @devdisp.c
    │   │
    │   ├── @devlog.h       Device LOGDATEI: Logdatei schreiben
    │   ├── @devlog.c
    │   │
    │   ├── @devprog.h      Device HEIZUNG: Steuerprogramm für Heizungen
    │   ├── @devprog.c
    │   │
    │   ├── @devtimer.h     Device TIMER: Verwaltet Timerprogamme für Events
    │   ├── @devtimer.c
    │   │
    │   │
    │   │   ____________________________________________________________
    │   │   Linkbibliotheken aus 'c/pi/bin/bininc/' zur Ansteuerung der
    │   │   Raspberry Pi Hardware. Diese Interfaces werden von den Aktoren
    │   │   Sensoren verwendet.
    │   │
    │   ├── @gpioci.h       GPIO-Interface für Paspberry I/O-Pins. Verwendet
    │   ├── @gpioci.c       das Chip-Interface /dev/gpiochip0 zum Steuern.
    │   ├── @gpioci_pi.h    Kopie von <linux/gpio.h> vom Raspberry Pi
    │   │                   für die Simulation am PC
    │   │
    │   ├── @gpiosb.h       GPIO-Interfaces im sysfs unter /sys/bus
    │   ├── @gpiosb.c       - i-Wire Interface mit SBUS /sys/bus/w1 steuern
    │   │                   - i2c-Interface mit SBUS /sys/bus/i2c steuern
    │   │
    │   ├── @gpioi2.h       GPIO-Interface: i2c-Interface /dev/i2c-X mit ioctl()
    │   ├── @gpioi2.c
    │   │
```

Fortsetzung

```
    │   │   ──────────────────────────────────────────────────────────
    │   │   Linkbibliotheken aus 'c/pi/bin/bininc/' für Aktoren und Sensoren.
    │   │   Diese Devices verwenden die GPIO-Interfaces zur Ansteuerung.
    │   │
    │   ├── @relayc.h       Device: Relais mit Rasperry GPIO steuern
    │   ├── @relayc.c
    │   │
    │   │
    │   ├── @ds1820s.h      Device: 1-Wire Temperatursensore vom Typ DS1820
    │   ├── @ds1820s.c
    │   │
    │   │
    │   ├── @bmp180i.h      Device: Sensor BMP180. i2c Feuchte und Temperatursensor
    │   ├── @bmp180i.c              i2c-Interface /dev/i2c-X mit ioctl()
    │   │
    │   │
    │   ├── @bmp180s.h      Device: Sensor BMP180. i2c Feuchte und Temperatursensor
    │   ├── @bmp180s.c              SBus i2c-Interface /sys/bus/i2c
    │   │
    │   │
    │   └── ...
    │
    ├── devtest_con.geany       Starter für IDE geany
    │
```

Beispiel: Device BMP180

Beispiel: Device für Druck- und Temperatursensor BMP180.

GPIO-Schnittstellen

Sensoren oder Aktoren können über verschiedene GPIO-Schnittstellen gesteuert werden:

- ▷ Chip-Interface /dev/gpiocihip0
- ▷ Interface /sys/bus
- ▷ i2c-Interface /dev/i2c-X mit ioctl() steuern

Die entsprechenden Programm findet man im Linkordner c/pi/bininc/gpio/.

Linkordner c/pi/bininc/

Für BMP180 gibt es zum Beispiel die Interface-Module bmp180i und bmp180s. Im Beispiel wird Modul bmp180i verwendet.

```
c/pi/bininc
 │
 ├── 1_read.me
 │
 ├── bmp180/
 │    │
 │    ├── 1_Dokus
 │    │    ├── ...
 │    │    └── ...
 │    ├── 1_read.me
 │    │
 │    ├── bmp180i.h   Objekt Bmp180i: i2c=0x77 Temperatur, Luftdruck
 │    ├── bmp180i.c   Schnittstelle : gpioi2.c, i2c-Interface /dev/i2c-X mit ioctl()
 │    │
 │    ├── bmp180s.h   Objekt Bmp180s: i2c=0x77 Temperatur, Luftdruck
 │    └── bmp180s.c   Schnittstelle : gpioib.c, SBus i2c-Interface /sys/bus/i2c
 │
 .    ___________________________________________________________________
 │
 ├── gpio/                 GPIO-Module für die GPIO-Hardware vom Pi
 │    │
 │    ├── gpioci.h         Objekt Gpio: I/O Steuerung der J8 Pins mit dem
 │    ├── gpioci.c         Chip-Interface /dev/gpiocihip0
 │    ├── gpioci_pi.c      Simulation am PC: Kopie von <linux/gpio.h> vom Pi
 │    │
 │    │
 │    ├── gpiosb.h         Objekt SBus: Steuerung der J8 Pins mit
 │    ├── gpiosb.c         SBus Interface /sys/bus
 │    │                    i-Wire Interface: /sys/bus/w1/devices   dtoverlay=w1-gpio
 │    │                    i2c-Interface:    /sys/bus/i2c/devices
 │    │
 │    │
 │    ├── gpioi2.h         Objekt I2c: i/O Steuerung  der J8 Pins mit dem
 │    ├── gpioi2.c         i2c-Interface /dev/i2c-X mit ioctl() steuern
```

DevCtl-Schnittstelle

Alle Devices haben eine DevCtl-Schnittstelle mit einheitlichen DEVCtlXxxx Befehlen. Diese Schnittstelle kann einfach in eine bestehendes Treiberprogramm eingefügt werden.
Beispiel: Die Funktion DevOpen() versucht alle konfigurierten Sensoren zu initialisieren. Im Fehlerfall wird der Status auf -1 gesetzt.

Für Modul bmp180i Die findet man die Schnittstelle in bmp180i-h:

```
// ▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀
// ░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░
// Device Sensor BMP180: i2c Feuchte und Temperatursensor
//
// GPIO-Schnittstelle: i2c-Interface /dev/i2c-X mit ioctl()
//                     gpioi2.h, gpioi2.c
//
// ▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀
//
// DevCtl-Schnittstelle für Device-Verwaltung: devices.h, devices.c
//
bool Bmp180iDevCtl(char Ctl, tDev *Dev);

// Ctl Befehl
//   DEVCtlHeader     Rückgabe:     C-Header
//   DEVCtlRem        Rückgabe:     Gerätebeschreibung
//   DEVCtlOpen       Openparameter:
//                    Dev->DevId    i2c Adresse
//                    Dev->Modus    1 Temperatur, 2 Druck
//                    Rückgabe:     Status=Modus, im Fehlerfall Status=-1
//   DEVCtlClose      Close Bmp180
//
//   DEVCtlExec       Rohwerte für Temperatur/Druck bestimmen und speichern
//   DEVCtlRead       Gepeicherte Rohwerte in Sensorwert umrechnen:
//                    Dev->Status==1: Temperatur: Dev->ComInt °C mal 1000
//                    Dev->Status==2: Druck:      Dev->ComInt Pa mal 1000
//                    Dev->ComStr Anzeigestring
//
//   DEVCtlWrite      Nur Debug:  Sensorwert Rohwert Dev->ComInt schreiben
//
//   DEVCtlInfo       ComInt=0: Infos zum Objekt Bmp180
//                    ComInt=1: Testkalibrierung anzeigen
//
// Ende DevCtl Schnittstelle
```

DevCtl Funktionen

Das Treiberprogramm für Sensor BMP180 stammt vom Hersteller. Die DevCtl-Schnittstelle kann durch folgenden Programmcode implementiert werden.

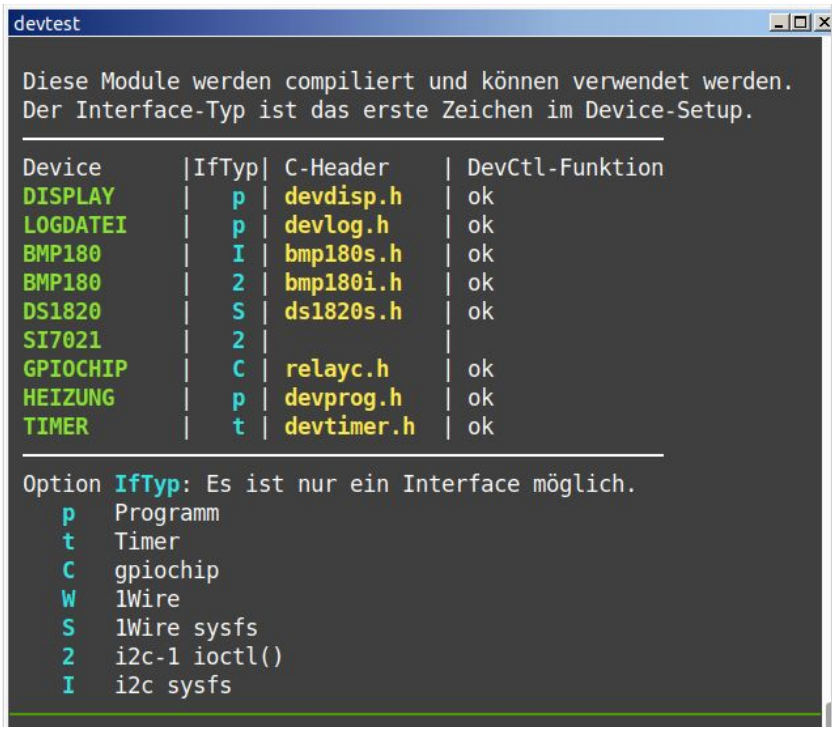
```
// ▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
// ░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░

// DevCtl Schnittstelle für Modul BMP180
//
#define HEADER "bmp180i.h"
#define REM_P  "Druck i2c"
#define REM_T  "Temp i2c"

static int32_t Roh_T=DEVComIntNil;                    | Rohwert Temperatur
static int32_t Roh_P=DEVComIntNil;                    | Rohwert Druck

bool Bmp180iDevCtl(char Ctl, tDev *Dev)               | Dev-Com Schnittstelle für Treiber BMP180
{ if (!Dev) return false;

  switch(Ctl)
  { case DEVCtlHeader: Dev->ComStr=HEADER;
    return true;

    case DEVCtlRem:
      Dev->ComStr=NULL;
      if (Dev->Modus==1) Dev->ComStr=REM_T;
      if (Dev->Modus==2) Dev->ComStr=REM_P;
    return true;

    case DEVCtlOpen:                                  | /dev/i2c: Keine spezielle open-Funktion notwendig
                                                      | DevId="0x77" i2c-Adresse
    return Bmp180iOpen(Dev);                          | Dev->Status=Dev->Modus

    case DEVCtlRead:                                  | Daten vom Chip lesen
                                                      | ComInt auf Temperatur oder Druck setzen
                                                      | ComStr auf Anzeigestring setzen
    return Bmp180iRead(Dev);

    case DEVCtlWrite:
    return Bmp180iWrite(Dev);

    case DEVCtlExec:                                  | Rohwerte  für Temperatur und Druck vom Chip lesen
    return Bmp180iExec(Dev);

    case DEVCtlClose:                                 | Close-Funktion des Treibers aufrufen
    return Bmp180iClose(Dev);

    case DEVCtlInfo:                                  | Infos zum Device anzeigen
    return Bmp180iPrintInfo(Dev);


    default: Dev->ComStr=NULL; Dev->ComInt=DEVComIntNil; return false;
  }
  return false;
} // -------------------------------------------------------------------------
```

Modulliste für die Devices

Das Programm-Module wird Sensor BMP180 wird in die Modulliste devicesm.c des betreffenden Steuerprogramms eingetragen.

```
//
//
//
// Device-Module
// Liste der für ein Programm verfügbaren Devices und Steuerfunktionen.
//
// Beim Anlegen Objekt Dev wird die Konfigurationsdateifür Devices gelesen.
// Für jedes Device wird der Modul-Name und Setup-String ermittelt.
// Setup[0] ergibt Interface-Typ aus 'gpio.h'. Setup==NULL ist GPIOTypNIL.
//
// Mit Modulnamen und Interface-Typ wird aus der Modulliste die passende
// Steuerfunktion XxxxCtl für ein Device ermittelt.
//
// Neue Geräte einbinden:
//  1. Header des C-Programms includen
//  2. Modulname, den Interface-Typ aus 'gpio.h'
//     und die Steuerfunktion ModulLst[] eintragen
//
  // ───────────────────────────────────────────────────────────────────────

  // ───────────────────────────────────────────────────────────────────────
  // // C-Header der eingebunden Device-Programme
  //
   #include "devdisp.h"   // Programm-Modul     | Anzeige
   #include "devlog.h"    // Programm-Modul     | Logdatei
   #include "bmp180s.h"   // SBus-Device, i2c   | Druck- und Temperatursensor
   #include "bmp180i.h"   // i2c-Interface      | Druck- und Temperatursensor
   #include "ds1820s.h"   // SBus-Device, 1Wire | Temperatur
   #include "relayc.h"    // Goio-Chip Device   | Rasperry I/O
   #include "devprog.h"   // Steuerprogramme    | Heizung
   #include "devtimer.h"  // Programm-Modul     | Device-Timer
  // ───────────────────────────────────────────────────────────────────────
  // Modulliste
  //
  tModulLst ModulLst[]=
  { //        Modulname          Interface-Typ              Steuerfunktion
    { .Modul="DISPLAY",  .IfTyp= GPIOIfTypProg,      .DevCtl= DispDevCtl     },
    { .Modul="LOGDATEI", .IfTyp= GPIOIfTypProg,      .DevCtl= LogDevCtl      },
    { .Modul="BMP180",   .IfTyp= GPIOIfTypI2cSys,    .DevCtl= Bmp180DevCtl  },
    { .Modul="BMP180",   .IfTyp= GPIOIfTypI2c,       .DevCtl= Bmp180iDevCtl },
    { .Modul="DS1820",   .IfTyp= GPIOIfTyp1WireSys, .DevCtl= DsDevCtl       },
    { .Modul="GPIOCHIP", .IfTyp= GPIOIfTypChip,      .DevCtl= RelayDevCtl    },
    { .Modul="HEIZUNG",  .IfTyp= GPIOIfTypProg,      .DevCtl= HeizungDevCtl },
    { .Modul="TIMER",    .IfTyp= GPIOIfTypTimer,     .DevCtl= TimerDevCtl    },
    { .Modul=NULL }
  };
```

Menubefehl von devtest: 1 Devices testen / 3 Kompilierte Device-Module anzeigen

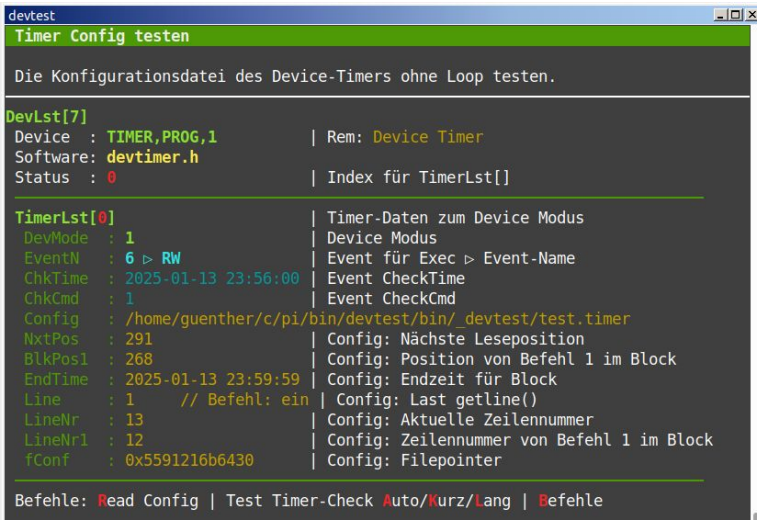


Beispiel: Device TIMER

Device Timer kann zum termingesteuerten Aufruf von Events verwendet werden. Das Modul Device-Timer kann mehrere Timer verwalten. Die Timer-Nummer gehen von 0 bis TIMERMax. Im Timer-Event wird mit dem Devicelink Device="TIMER,PROG,1" der Device Modus 1 und damit Timer 1 gewählt (Beispiel).

Die folgenden Testfunktionen für Timer-Devices findet man in devtest unter: 1 Devices testen / 2 Device testen / Timer testen

Die Daten von Timer 1 werden in einer Struktur TimerLst[0] gespeichert. Der Index 0 wird dabei automatisch vergeben und unter Status gespeichert.



Die Laufzeitdaten von Timer 1:

DevMode:	Nummer des Timers
EventX:	Event-Index und Name für Exec-Funktion
ChkTime:	Event CheckTime
ChkCmd:	Parameter für Exec-Funktion, -1 kein Befehl
Config:	Pfad der Config-Datei
NxtPos:	Pfad Nächste Leseposition in Config
BlkPos1:	Position von Befehl 1 im Befehlsblock
EndTime:	Endzeit für den Befehlsblock
Line:	Last getLine() aus Config
LineNr:	Zeilennummer von Line
LineNr1:	Zeilennummer von Befehl 1 im Block
fConf:	Filepointer für timer1.conf

Timer Config

Aufbau

Die Schaltfunktionen des Timer-Devices werden mit einer Config-Datei beschrieben. Der Dateiname wird im Device-Setup z.B. mit Setup="t test.timer" definiert. Die Config-Datei wird dann im Konfigurationsverzeichnis des Programms erwartet.

Diese Datei wird täglich zeilenweise neu gelesen. In jeder Zeile ist das erste Zeichen ein Befehlszeichen. Remarks und Leerzeilen werden übersprungen. Die Event-Befehle sind Blockweise organisiert. Jeder Befehlsblock beginnt mit Befehl 's ...' und endet mit Befehl 'l' für Loop oder 'n' für nächster Block. Die Böcke müssen mach Blockzeiten aufsteigend definiert werden.

```
// Beispiel 1
s t14, 6:0:0, 6:18:0
x 1    // Exec 1
w 18:0 // Warte 18 min
n 0    // Exec 0 bei Blockende
```

```
s Blockanfang | An den Wochentagen 1=Montag und 4=Donnerstag von 6:0:0 bis 6:15:0
x 1    | Exec 1 (On) aufrufen. Das Event wird mit Open="Event='Name'" festgelegt.
w 18:0 | Wartezeit bis zum nächsten Event | Wunsch 18 Minuten | Maximal Blockende
n nächster Block | Zeitpunkt 6:18:0 | Event Exec 0 (Off) aufrufen
```

```
// Beispiel 1

s 6:0:0, 6:5:0
x 1  // Exec 1
w 20 // Warte 20 sec
x 0  // Exec 0
w 30
l 0
```

```
s Blockanfang | Zeiten täglich von 6:0:0 bis 6:5:0
x 1  | Exec 1 | EventN | Beispiel Relay On
w 20 | Wartezeit 20 sec
x 0  | Exec 0 | EventN | Beispiel Relay Off
w 20 | 30 sec Wartezeit bis zum nächsten Event
l Loop | Befehle bis 6:5:0 wiederholen | Am Ende immer Event Exec 0 (Relay Off) ausführen
```

```
// Beispiel 2
s t06, 6:0:0
x 3   // Sensor abfragen
w 0
n
```

```
s Blockanfang | t06 nur am Sonntag 0 und Freitag 6 | Zeitpunkt 6:0:0
x 3 | Exec 3 | Beispiel Sensorabfragew 0 | keine Wartezeit
weiter mit dem nächsten Block
```

```
// Beispiel 3
s 2024-12-16 7:0:0, 2024-12-17 7:5:0
x 1
w 20
x 2
w 0
n
```

```
s Blockanfang | Zeitraum 2024-12-16 bis 2024-12-17 | Zeit von 7:0:0 bis 7:5:0
x 1  | Event-Exec 1
w 20 | 20 sec Wartezeit
x 2  | Event-Exec 2
w 0  | keine Wartezeit
weiter mit dem nächsten Block
```

```
// Beispiel 4
s 2024-11-30 7:0:0
x 3
w 0
n
```

```
s Blockanfang | Zeitraum ab 2024-11-30 | Täglich um 7:0:0
x 3 | Exec 3
w 0 |keine Wartezeit
weiter mit dem nächsten Block
```

Config Befehle

Befehl: 1 Devices testen / 2 Device testen /
Timer testen / Befehle

Der Befehl listet die möglichen Timerbefehle für die Config-Datei. Timerbefehle werden in Befehlsblöcken programmiert. Ein Befehl pro Zeile. Jeder Block startet mit "s ... " und endet mit "n ... " oder "l ..."

s Blockanfang. Die Parameter:

[t[0-6],]	Wochentag, optional t0 steht für Sonntag t15 steht für Montag und Freitag
Startdatum	Datum mit Zeit oder nur Zeit für den aktuellen Tag
Enddatum	Datum mit Zeit oder nur Zeit für den aktuellen Tag oder Defaultwert Startdatum

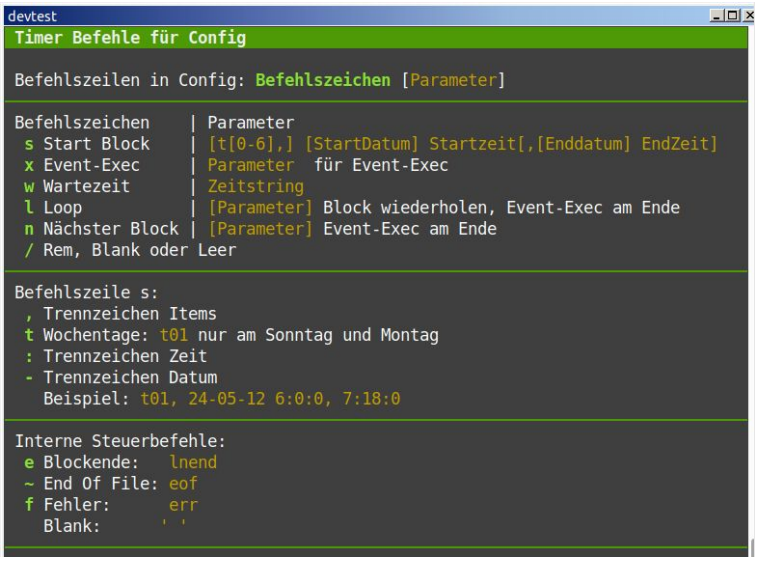
x Parameter für Event-Exec Befehl
Wird nur innerhalb der Blockzeit ausgeführt.

w Wartezeit aud den nächsten Befehl.
Format: Stunden:Minuten:Sekunden
Wird nur innerhalb der Blockzeit ausgeführt.

l Block wiederholen. Beim Beenden des Blocks kann ein Event-Exec Parameter ausgeführt werden.

n Nächsten Block suchen Beim Beenden des Blocks kann ein Event-Exec Parameter ausgeführt werden.

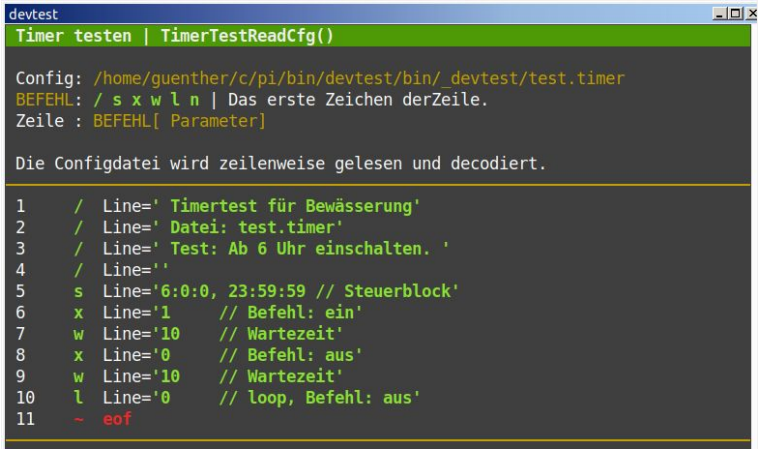
/ Remark, Blank oder Leerzeile



Config anzeigen

.. / Timer testen / Read Config

Decodiert alle Zeilen der Configdatei



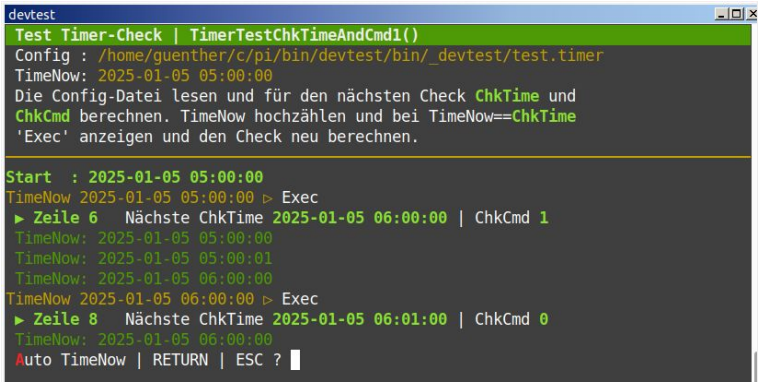
Config testen

.. / Timer testen / Test Timer-Check Auto

Mit dem Befehl können die programmierten Schaltpunkte des Timers manuell überprüft werden.

Nach Eingabe einer Startzeit werden die nächsten Checkwerte ChkTime und ChkCmd berechnet.

RETURN TimeNow wird um eine Sekunde erhöht.
Auto TimeNow läuft bis zu nächsten ChkTime.



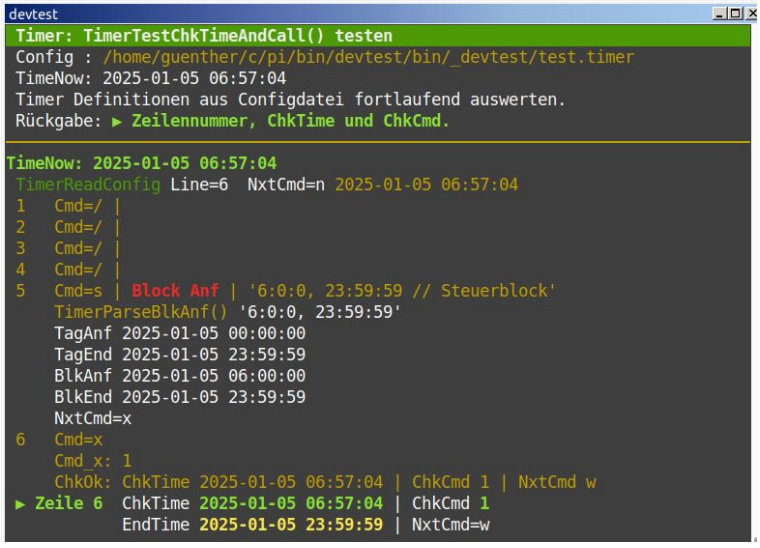
Config debuggen

.. / Timer testen / Test Timer-Check Lang

Mit dem Befehl kann das Verhalten des Timers im Debugmodus geprüft werden.

Nach Eingabe einer Startzeit werden die nächsten Checkwerte ChkTime und ChkCmd berechnet.

In diesem Modus kann das Parsen der Config-Datei genau mit Zeilenummern verfolgt werden.



Beispiel: Bewässerung

Erweiterung: Bewässerung an bestimmten Wochentagen.

Hardware: Zum Einschalten der Bewässerung wird ein Relais an Pin 15 des Rasperry Pi verwendet.

Software: Das Relais wird mit einem Device="GPIOCHIP,Relays,15" geschaltet.
Die Zeitsteuerung verwendet einen Timer Device="TIMER,PROG,1"

Device Konfiguration: test.devices

```
// -------------------------------------------------
// Bewässerung
// Devices: "GPIOCHIP"  Relay Wasser
//          "TIMER"     Zeitsteuerung

{ Modul="GPIOCHIP",        // Rasperry GPIO Steuerung
  DevId="Relays",          //
  Rem  ="Relay Wasser",    // Wasser on/off
  Setup="Co0",             // Chip, Output, Start 0
  Modus=15,                // Pin 15 GPIO J8-Nummer
},

{ Modul="TIMER",           // Timerprogramm verwendet
  DevId="PROG",            //
  Rem  ="Timer Wasser",    // Zeitsteuerung
  Setup="t wasser.timer",  // Timer, Zeiteinstellungen
  Modus=1,                 // Timerindex 1
},
```

Die Ansteuerung erfolgt mit folgenen Events:
Event Konfiguration: test.events

```
// -----------------------------------------------------
// Timer Bewässerung
// Events: "RW"       Relay Wasser
//         "TimerRW"  Timer

{Name="RW",                    // Event Relay Wasser
 Rem="Relay Wasser",
 Typ="r",                      // Relay
 Device="GPIOCHIP,Relays,15",  // Id: Treiber,Typ,Pin
 Open="",
 Cmd=0,                        // verwenden
 WHSec=0,                      // kein Aufruf durch Loop()
},

{Name="TimerRW",               // Event Timer
 Rem="Timer Wasser",
 Typ="t",                      // Timer
 Device="TIMER,PROG,1",        // TIMER,PROG,Timer-Nr
 Open="Event='RW'",            // Timer steuert Event RW
 Cmd=0,                        // verwenden
 WHSec=0,                      // kein Aufruf durch Loop()
},
```

Timer Einstellungen: wasser.timer

Siehe: Timer Configdatei

```
// Timer für Bewässerung
// Datei: wasser.timer
// Wasser am Montag, Mittwoch und Freitag
// um 6 Uhr einschalten.
// Nach 15 Minuten ausschalten

s t134, 6:0:0, 6:15:0 // Steuerblock: Tage 124 von 6:0:0 bis 6:15:0
x 1                   // Befehl: Wasser ein
w 15:00               // Warte 15 Min
n 0                   // Blockende, Befehl: Wasser aus
```

Timereinstellungen zum Testen:

```
// Timertest für Bewässerung
// Datei: test.timer
// Test: Ab 6 Uhr einschalten.

s 6:0:0, 23:59:59 // Steuerblock
x 1     // Exec 1: einschalten
w 1:0   // Wartezeit 1 Min
x 0     // Exec 0: ausschalten
w 1:0   // Wartezeit 1 Min
l 0     // loop, Exec 0: ausschalten
```

Beispiel: Device LOGDATEI

Mit dem Device LOGDATEI kann der Ablauf der Steuerung protokolliert werden. Device LOGDATEI verwendet das Objekt Log zum Anlegen und Schreiben der Logdatei.

Objekt Log kann unabhängig von Events/Devices/Loop zum Verwalten und Schreiben von Logdateien verwendet werden. Die Einstellungen erfolgen in der Konfigurationsdatei des Programms. Der automatisch vergebene Logdateiname ist das Erstellungsdatum mit dem Suffix '.log'.

Beispiel: Ausschnitt aus der Konfiguration von devtest: devtest.config

```
...

// Logdatei Einstellungen für Device-Log
LogLabel    ="Scandisk";  // Name des USB-Datenträgers für Logdatei
LogDirOpt   ="";          // Optionales Dir für Logdatei

LogOn  =1;                // 0/1 Logdatei schreiben
LogErr =1;                // 0/1 Fehler in Logdatei schreiben

...
```

Option: LogLabel="Name" Name des USB-Datenträgers für die Logdatei. Der Datenträger wird automatisch eingebunden.
Ohne "Name" wird die Logdatei im Konfigurationsverzeichnis angelegt.
Option: LogErr=1 Device-Log schreibt alle Programmfehler in die Logdatei.

Zur Verwendung von Objekt Log in einer Loop() wird ein Event mit dem Device LOGDATEI angelegt.

Beispiel: Ausschnitt aus der Konfiguration Events: test.events

```
...

 {Name="Log",                  // Event-Id: Logdatei
  Rem="Logdatei",              // Remark
  Typ="l",                     // Logdatei
  Device="LOGDATEI,PROG,0",    // Device-Link
  Open="RW",                   // Option: Die Exec Funktion von Event "RW" loggen
  Cmd=0,                       // Exec Parameter
  WHSec=20,                    // LogExec() alle 20 Sec
 },

...
```

WHSec =20 Alle 20 Sekunden werden in LogExec() alle passenden Events mit Device-Read gelesen und geloggt.
Open ="RW, xx, ..." Option: Die gelisteten Event-Namen werden bei jedem Aufruf von Event-Exec geloggt.

Beispiel: Ausschnitt aus der Konfiguration Devices: test.devices

```
...

 { Modul="LOGDATEI",        // Logdatei,
   DevId="PROG",            // Programm-Modul
   Modus=0,                 // verwenden
   Rem  ="Logdatei schreiben",
   Setup="p",               // Programm
 },

..
```

Das Testen der Logunktionen kann mit einem Timer durchgeführt werden.
Beispiel: Konfiguration desTimers: test.timer

```
// Timertest für Bewässerung
// Datei: test.timer
// Test: Ab 6 Uhr einschalten.

s 6:0:0, 23:59:59 // Steuerblock
x 1     // Exec 1: einschalten
w 1:0   // Wartezeit 1 Min
x 0     // Exec 0: ausschalten
w 1:0   // Wartezeit 1 Min
l 0     // loop, Exec 0: ausschalten
```

Mit obigen Einstellungen wird folgende Logdatei erzeugt.

```
Logstart [2025-01-12 17:29:35] Prog:devtest Ver:0.59
LogIsOn: true
LogErr : true                                |Fehler werden auch in die Logdatei beschrieben
LogDev : Open                                |Device LOGDATEI in Verwendung
Events : RW                                  |Exec von Event RW überwachen
WHsec  : 20                                  |Alle 20 Sec alle Events mit LogExec()loggen

Err| LogIsOn: Test Err Umleitung             |Fehlerumleitung aktiviert
 [17:29:39] T1= Fehler, TF= Fehler, RW= OFF  |Beim Start fehlen noch die Daten
RW [17:29:39] T1= 14.5 C, TF= 15.0 C, RW= ON |Überwachtes Event RW: Exec 1, Relay On
 [17:29:59] T1= 14.5 C, TF= 15.0 C, RW= ON   |LogExec()alle 20 Sec
 [17:30:19] T1= 14.5 C, TF= 15.0 C, RW= ON
 [17:30:39] T1= 14.5 C, TF= 15.0 C, RW= ON
RW [17:30:39] T1= 14.5 C, TF= 15.0 C, RW= OFF|Überwachtes Event RW: Exec 0, Relay Off
 [17:30:59] T1= 14.5 C, TF= 15.0 C, RW= OFF
 [17:31:19] T1= 14.5 C, TF= 15.0 C, RW= OFF
 [17:31:39] T1= 14.5 C, TF= 15.0 C, RW= OFF
RW [17:31:39] T1= 14.5 C, TF= 15.0 C, RW= ON
 [17:31:59] T1= 14.5 C, TF= 15.0 C, RW= ON
Logend [2025-01-12 17:31:59]                 |Ende der Aufzeichnung
```

GNU General Public License

```
/*
 *
 * Copyright 2022-2025 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.
*/
```