

devtest | Testprogramm für Steuerungen

Inhaltsverzeichnis

devtest | Testprogramm für Steuerungen

Einleitung

| | |
|---------|---|
| Loop | 2 |
| Devices | 2 |
| Events | 2 |
| devtest | 2 |

Beispiel: Heizung für ein Glashaus

| | |
|--------------------|---|
| Vorgaben | 3 |
| Lösungsweg | 3 |
| Loop | 3 |
| Events und Devices | 4 |
| Erweiterung | 4 |
| Steuerung testen | 4 |

Devices

| | |
|---------------------------|---|
| Devices für die Heizung | 5 |
| Dev Objekt | 5 |
| Konfiguration von Devices | 6 |
| Modul | 6 |
| DevId | 6 |
| Setupstring | 6 |
| Modus | 7 |
| DevNctl Funktion | 7 |
| Devices testen | 8 |

Events

| | |
|--------------------------|----|
| Event Objekt | 9 |
| Konfiguration von Events | 10 |
| Events testen | 10 |

Loop

| | |
|-------------|----|
| Loop testen | 12 |
|-------------|----|

Schnittstellen testen

| | |
|------------------|----|
| GPIO-Pins | 13 |
| Devices /sys/bus | 14 |
| Devices i2c | 14 |

Datei- und Ordnerübersicht

| | |
|------------------|----|
| Linkbibliotheken | 15 |
| Dateien | 15 |

Beispiel: Device BMP180

| | |
|----------------------------|----|
| GPIO-Schnittstellen | 17 |
| Linkordner c/pi/bininc/ | 17 |
| DevCtl-Schnittstelle | 17 |
| DevCtl Funktionen | 18 |
| Modulliste für die Devices | 19 |

Beispiel: Device TIMER

| | |
|-----------------|----|
| Timer Config | 20 |
| Aufbau | 20 |
| Config Befehle | 21 |
| Config anzeigen | 21 |
| Config testen | 21 |
| Config debuggen | 21 |

Beispiel: Bewässerung

| | |
|------------------------------------|----|
| Device Konfiguration: test.devices | 22 |
| Event Konfiguration: test.events | 22 |
| Timer Einstellungen: wasser.timer | 22 |
| Timereinstellungen zum Testen: | 22 |

Beispiel: Device LOGDATEI

GNU General Public License

Einleitung

In einfachen Steuerungen werden verschiedene Funktionmodule zum Auslesen von Sensoren und Ansteuern von Aktoren innerhalb einer Endlosschleife 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 verschiedenen Funktionsmodulen einfach zusammengestellt werden. Die passenden Module werden dabei über Konfigurationsdateien 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 Fenstersteuerung 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 Strukturen 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 Konfigurationdateien 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 Sourdateien folgen dem fixen Richtlinien von C-Projekt.

Die ersten Schritte:

- ▷ Das Programmgerüst von [gardenctl](#) mit Programm [newprg](#) in [c/pi/bin/](#) anlegen
- ▷ Linkbibliotheken 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 Linkbibliothek [c/pi/bininc/events/loop.*](#)

Das Steuerungsprogramm befindet sich normalerweise immer in der zentralen Verarbeitungsschleife 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()`
- ▷ `EventExecAndSetCheck()`
- ▷ `WaitSec=EventGetWaitSec()`
- ▷ `Taste=chkTasteSec(WaitSec)`

| |
|---|
| Aktuelle Uhrzeit oder Simulationszeit bestimmen |
| Device-Exec Funktionen ausführen und die nächste Werte für CheckTime bestimmen |
| WaitSec ist die kleinste CheckTime der wartenden Events |
| Maximal WaitSec auf eine Taste warten |
| Taste==taste_nil keine Taste in WaitSec, continue, Event behandeln |
| Taste==taste_select Device-Interrupt behandeln. Siehe Programm picamctl |
| Taste==Befehl Befehl ausführen |
| Taste auswerten |

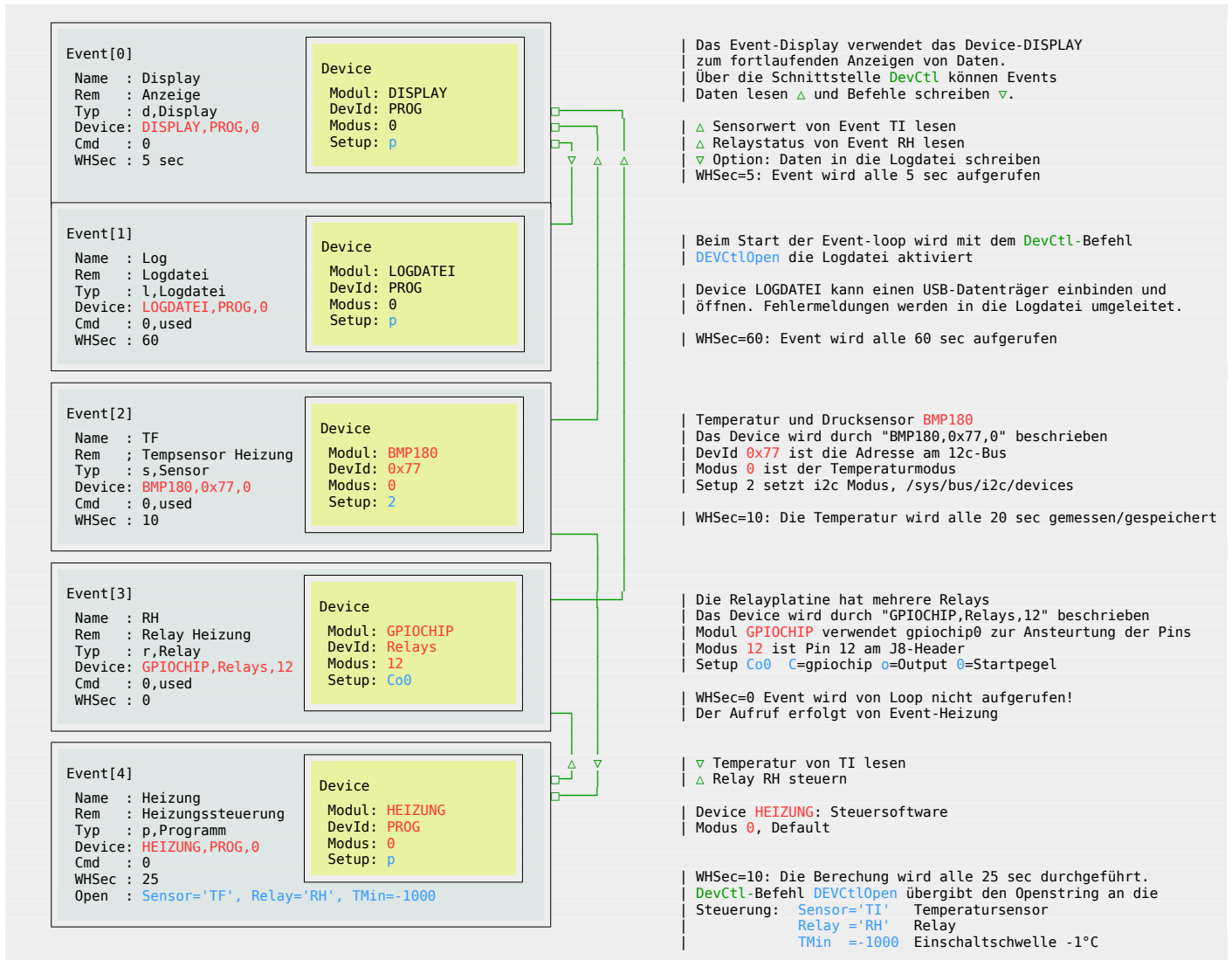
- ▷ `switch(Taste)`

Events and Devices

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

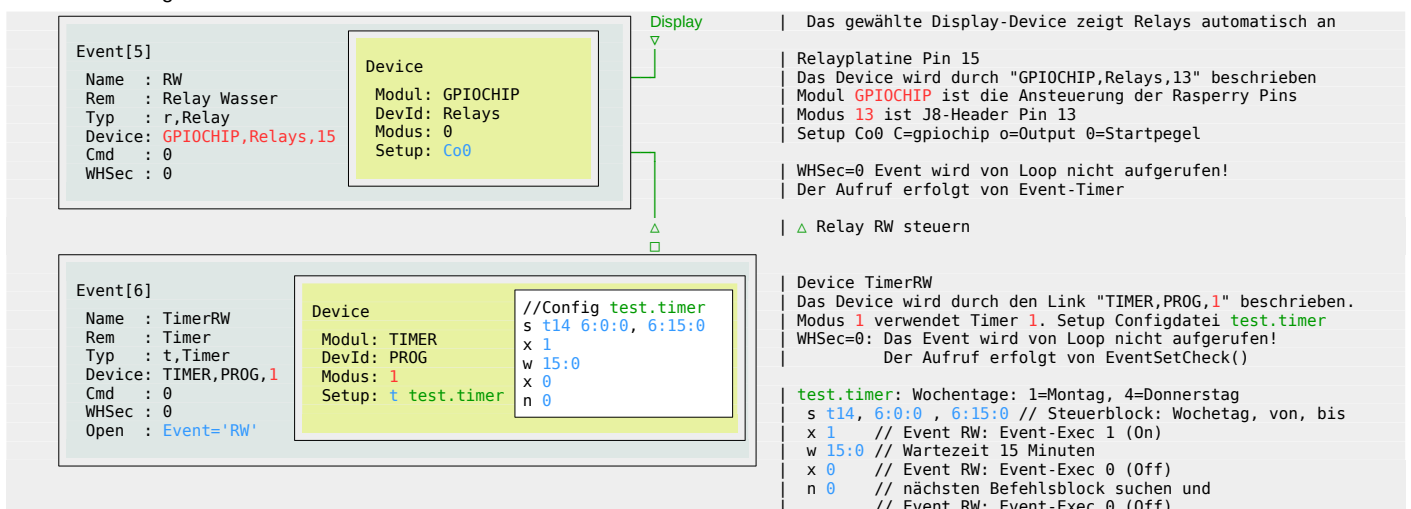
Die genaue Beschreibung der verwendeten Strukturen folgt in den Kapiteln [Event](#) und [Device](#).

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 // Devedefinition 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 I2c-Device-ID
    char *Setup; // Setupstring des Geräts
    int16_t Modus; // Wunschmodus für das Device:
    // // -1 Device nicht verwenden
    // // m Pin m oder Deviceoption m verwenden
    // -----
    // Programm-Parameter
    // // readonly: Steuerungsfunktion für das Modul. Wird mit dem Modulnamen
    tFnkDevCtl DevCtl; // aus der ModulLst[] in devicesm.c bestimmt
    // // 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: // Parametertausch mit dem Device
    int32_t ComInt; // In/Out Deviceparameter. Ungültig: DEVCtlNil
    char *ComStr; // In/Out String. Nur temporäre Strings!
}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 ersetzt 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
    DevId="PROG",             // Programm-Modul
    Rem = "Anzeige",         //
    Setup="p l",             // Setupstring, Programm, l Led
    Modus=0,                  // Wunschmodus
                                // -1
                                // n
  },
  { Modul="BMP180",           // Sensor BMP180
    DevId="0x77",             // Id für /dev/i2c
    Rem = "Temp Heizung",     // Remark: Beschreibung
    Setup="2",                // Interface /dev/i2c-1 mit ioctl()
    Modus=1,                  // 1 Temperatur, 2 Druck
  },
  { Modul="HEIZUNG",          // Modul: Steuerung
    DevId="PROG",             //
    Rem = "Prog Heizung"     //
    Setup="p",                //
    Modus=0,                  // verwenden
  },
  { Modul="GPIOCHIP",         // Modul: I/O-Steuerung
    DevId="Relay",            //
    Rem = "Relay Heizung"    //
    Setup="Co0",              // 'C' Chip, 'o' Output, '0' Startwert
    Modus=12,                 // Pinnummer vom J8-Header
  },
}
```

| | |
|--|-------------------------|
| Modulname aus Modulliste | Device-Id: Software |
| Optionen: Chip-ID, I2c-Device-Id oder PROG | Beschreibung des Moduls |
| Interface-Typ: p Software, Option: l Pi Led schalten | Gewünschter Devicemodus |
| Device n, Devicemodus n oder Pin n verwenden | Device nicht verwenden |

| | |
|---|-------------|
| Der Sensor BMP180 hat zwei Modi (Druck und Temperatur) und kann über verschiedene GPIO-Interfaces angesteuert werden. | i2c Adresse |
|---|-------------|

| | |
|---|-----------------|
| 1. Zeichen '2', i2c Ansteuerung über /dev/i2c-1 mit ioctl() | Temperaturmodus |
|---|-----------------|

| | |
|---|---------------------|
| Software: Liest die Temperatur von Modul="BMP180" und steuert das Relay mit Modul="GPIOCHIP". Die Terminalanzeige erfolgt mit Modul="DISPLAY" | Device-Id: Software |
|---|---------------------|

| |
|--|
| Interface-Typ oder p Software, t Timer |
|--|

| | |
|---|------------------|
| GPIO Interface gpiochip0 für Raspberry Pi | Device-Id: Relay |
|---|------------------|

| | |
|--|----------------------------|
| 1. Zeichen 'C' Interface gpiochip0, "o0" Setupstring | J8 Pinnummer 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 Geratetyp. Nicht eindeutig! |
| DevId : Relays | Device-Id: PROG, 1-Wire-ID, Chip-ID oder I2c-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/chipchip0
#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 DevCtl() (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

| n | Modul | DevId | Mode | St | Setup | C-Header | Rem |
|---|----------|-----------------|------|----|-------|-----------|-------------------|
| 0 | DISPLAY | PROG | 0 | 0 | p | devdisp.h | Anzeige |
| 1 | DS1820 | 10-000802e444d1 | -1 | -1 | S | NULL | Tempensor Aussen |
| 2 | BMP180 | 0x77 | 1 | 1 | 2 | bmp180i.h | Tempensor Heizung |
| 3 | GPIOCHIP | Relays | 12 | 12 | Co0 | relayc.h | Relay Heizung |
| 4 | HEIZUNG | PROG | 0 | 0 | p | devprog.h | Prog Heizung |

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 getestet werden.

Mir den Parametern ComInt und ComStr können Daten ausgetauscht werden.

Debug Detaillierte 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 Linkbibliothek `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 angelegt. 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 Device-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

```
devtest
devtest[0.62] Steuerung testen: Devices, Events, Loop und GPIO-Pins

Rechner: pc780mint
DirDat : /home/guenther/c/pi/bin/devtest/bin/_devtest/
Events : test.events
DeVICES: test.devices

Info | Help

1 DeVICES testen
2 Events testen
3 Loop testen

5 GPIO-Pins /dev/gpiochip
6 /sys/bus Devices
7 /dev/i2c-x Devices

Restart
Quit
```

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

```
devtest
Events

0 Eventübersicht
1 Alle Events anzeigen
2 Event mit Device anzeigen
3 Event testen

4 Events mit Devices linken und open Devices
5 Konfigurationsdatei anzeigen/laden

d Menu Devices
```

Befehl: **2 Events testen/ 0 Eventübersicht**

```
devtest


| n: | Event id | Typ | Device-Link              | Dev | Cmd | CmdE | Rem               |
|----|----------|-----|--------------------------|-----|-----|------|-------------------|
| 0: | Display  | d   | DISPLAY,PROG,0           | 0   | 0   | 0    | Anzeige           |
| 1: | TI       | s   | DS1820,10-000802e444d1,0 | 1   | 0   | 0    | Temp Aussen       |
| 2: | TF       | s   | BMP180,0x77,1            | 2   | 0   | 0    | Temp Fruehbeet    |
| 3: | RH       | r   | GPIOCHIP,Relays,12       | 3   | 0   | 0    | Relay Heizung     |
| 4: | Heizung  | p   | HEIZUNG,PROG,0           | 4   | 0   | 0    | Steuerung Heizung |


```

Befehl: 2 Events testen/ 1 Alle Events anzeigen

```

devtest
Events aus EventLst[] | Size:5

Die Eventliste enthält die konfigurierten Events. Jedem Event kann mit dem
dem Link Device="Modul,DevId,Modus" genau ein Device zugeordnet werden.
Events mit WHSec=0 oder CmdExec<0 werden in der Event-Loop nicht behandelt.
Für komplexe Termine kann das Device Timer verwendet werden.

Event-Liste
Event[0]
Name      : Display                | Eindeutige Event-Id
Rem       : Anzeige                | Remmark
Typ       : d,Display              | Art des Events
Device    : "DISPLAY,PROG,0       " | DeviceLink: "Modul,DevId,Modus"
Open      : "                      " | Device Open-String
Cmd       : 0,used                 | Wunschparameter für Event-Exec, -1 not used
WHSec     : 5                      | Event-Exec alle WHSec wiederholen

-----
CmdExec   = 0                      | Parameter für Event-Exec, -1 not used
DevN      = 0                      | Device[0], Rem: 'Terminalanzeige'
CheckTime = 00:00:00              | Nächste Check-Time für Loop oder 0
CheckCmd  = 0                      | Event-Exec Parameter bei Check
ErrCount  = 0                      | Fehlerzähler des Events
Debug     = 0                      | Debugausgabe für das Event

Event[1]
Name      : T1                    | Eindeutige Event-Id
Rem       : Temp Aussen           | Remmark

```

Befehl: 2 Events testen/ 2 Event mit Device anzeigen

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

```

devtest
Event und Device anzeigen
Event[4]
Name      : Heizung                | Eindeutige Event-Id
Rem       : Steuerung Heizung      | Remmark
Typ       : p,Programm             | Art des Events
Device    : "HEIZUNG,PROG,0       " | DeviceLink: "Modul,DevId,Modus"
Open      : "Sensor="TF", Relay="RH", TmpOn=25500" | Device Open-String
Cmd       : 0,used                 | Wunschparameter für Event-Exec, -1 not used
WHSec     : 25                     | Event-Exec alle WHSec wiederholen

-----
CmdExec   = 0                      | Parameter für Event-Exec, -1 not used
DevN      = 4                      | Device[4], Rem: 'Prog Frostschutz'
CheckTime = 2025-01-09 11:18:58   | Nächste Check-Time für Loop oder 0
CheckCmd  = 0                      | Event-Exec Parameter bei Check
ErrCount  = 0                      | Fehlerzähler des Events
Debug     = 0                      | Debugausgabe für das Event

DevLst[4]
Modul    : HEIZUNG                | Device-Modul. "Modul,DevId,Modus" bestimmt das Device!
DevId    : PROG                  | Device-Id: PROG, Chip-ID oder I2c-Device-Id
Modus    : 0                     | Gewünschter Modus, tatsächlicher Modus ist 'Status'
Setup    : "p                    " | Setupstring. Setup[0] ist der Interface-Typ
Rem      : Prog Heizung           | Rem Modul

-----
Header   : devprog.h             | Header/Beschreibung der Steuerfunktion
RemCtl   : Prog Frostschutz      | Rem Steuerfunktion, DEVCTLRem
DevCtl   : 0x55f8270ea85b        | Steuerfunktion FnDevCtl() aus Moduliste von 'devices.m'
Status   : 0                     | Nach Open: Tatsächlicher Modus, Close:-1
ComInt   : 0                     | Last DevCom Parameter: int32_t
ComStr   : Prog Frostschutz      | Last DevCom Parameter: String
Event    : Heizung, Steuerung Heizung | Besitzer des Devices

```

Befehl: 2 Events testen/ 2 Event testen

```

devtest
Event | testen
Event[4]
Name      : Heizung                | Eindeutige Event-Id
Rem       : Steuerung Heizung      | Remmark
Typ       : p,Programm             | Art des Events
Device    : "HEIZUNG,PROG,0       " | DeviceLink: "Modul,DevId,Modus"
Open      : "Sensor="TF", Relay="RH", TmpOn=25500" | Device Open-String
Cmd       : 0,used                 | Wunschparameter für Event-Exec, -1 not used
WHSec     : 25                     | Event-Exec alle WHSec wiederholen

-----
CmdExec   = 0                      | Parameter für Event-Exec, -1 not used
DevN      = 4                      | Device[4], Rem: 'Prog Frostschutz'
CheckTime = 2025-01-09 11:20:44   | Nächste Check-Time für Loop oder 0
CheckCmd  = 0                      | Event-Exec Parameter bei Check
ErrCount  = 0                      | Fehlerzähler des Events
Debug     = 1                      | Debugausgabe für das Event

Time: 2025-01-09 11:21:04 Sim +20 | Device Infos | Debug | RETURN
Event: ◀ ▶ | Open | Check and Exec | Close | Set Parameter | ESC

```

Loop

Loop testen

Befehle:

Time Time oder SimTime einstellen

```
devtest
Loop testen

Time: 2025-01-09 15:01:58 Uhr | Simtime on/off

Check      | CheckTime und CheckCmd berechnen/anzeigen
Debug Loop | Fortlaufende Debuganzeige, Debug=1
Loop       | Ausführen, Blockanzeige
Restart    | Konfigurationen neu einlesen

Befehl | ESC ?
```

Befehl: **Check**

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

```
devtest

EventSetCheck Display Nxt=0 > CheckTime:2025-01-09 14:19:21 CheckCmd:0
EventSetCheck T1 Nxt=0 > CheckTime:2025-01-09 14:19:21 CheckCmd:0
EventSetCheck TF Nxt=0 > CheckTime:2025-01-09 14:19:21 CheckCmd:0
EventSetCheck RH Nxt=0 > CheckTime:00:00:00 CheckCmd:0
EventSetCheck Heizung Nxt=0 > CheckTime:2025-01-09 14:19:21 CheckCmd:0

Time: 2025-01-09 14:19:21 Uhr | CheckTime und CheckCmd anzeigen.

EventLst Name |T|CmdE| CheckTime | CheckCmd
Event[0] Display |d| 0 | 2025-01-09 14:19:21 | 0
Event[1] T1 |s| 0 | 2025-01-09 14:19:21 | 0
Event[2] TF |s| 0 | 2025-01-09 14:19:21 | 0
Event[3] RH |r| 0 | 00:00:00 | 0
Event[4] Heizung |p| 0 | 2025-01-09 14:19:21 | 0
```

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 setzen
Stopp Loop anhalten

```
devtest
EventExecAndSetCheck( All, TimeSec:2025-01-09 15:41:47)
Loop[2025-01-09 15:41:47 Uhr]

T1: 14.5 C | Temp Aussen
TF: 15.0 C | Temp Fruehbeet
RH: ON | Relay Heizung

>>> Exec T1 CheckTime:2025-01-09 15:41:52 > skip
>>> Exec TF CheckTime:2025-01-09 15:41:52 > skip
>>> Exec RH CheckTime:00:00:00 > skip
>>> Exec Heizung CheckTime:2025-01-09 15:42:07 > skip

EventGetWaitSec TimeSec:2025-01-09 15:41:47 WaitSecMax:10
Display CheckTime: 2025-01-09 15:41:52
T1 CheckTime: 2025-01-09 15:41:52
TF CheckTime: 2025-01-09 15:41:52
Wait 5 sec auf Taste

Time: Uhr | Anzeige | Check | Debug: 1 | Write Com | Stopp
```

Befehl: **Loop Run**

Loop im Simulations-Modus mit fortlaufender Anzeige aufrufen.

```
devtest
Loop[2025-01-09 14:46:23 Uhr]

T1: 14.5 C | Temp Aussen
TF: 15.0 C | Temp Fruehbeet
RH: ON | Relay Heizung

Exit Terminal | Menu | Refresh | Anzeige
```

Schnittstellen testen

GPIO-Pins

Menubefehl: **5 GPIO-Pins /dev/gpiochip**

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

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

```

pi@pi6: ~
GPIO Pins testen

Pin[J8Nr]: Zeile 1 zeigt Gpio-Array | Zeile 2 zeigt Hardware
[12] BCM 18 pwm0
    └─Flags:o p 0x2

Pin   Physikalische J8Nr 1-40 wählen
Setup Mit Setupstring einstellen

Write Pegel für I/O Pin setzen
Read   Pegel von I/O Pin lesen
Event Event-Pin aktivieren | Testloop aufrufen
Alle  Alle Event-Pins aktivieren | Testloop aufrufen

Close Pin deaktivieren | Setup bleibt im Gpio-Array
Delete Gpio-Objekt freigeben

Infos  zum Gpio-Objekt
Layout für den Header J8

ESC   Test beenden
  
```

Option: **Infos zum Gpio-Objekt**

Es werden alle aktuellen Einstellungen des J8 Headers angezeigt,

```

pi@pi6: ~
Infos zum Gpio-Objekt
GPIO Einstellungen

Chipinfo '/dev/gpiochip0'
name : gpiochip0
label : pinctrl-bcm2835
lines : 54, J8-Pins=40
fgpio : 4

[ 3] BCM 2 sda i2c: 1
    └─Flags:i p 0x0
[ 5] BCM 3 scl i2c: 1
    └─Flags:i p 0x0
[ 7] BCM 4 gpclk0 W1: 2
    └─User:onewire@0 Flags:o p d k 0xb
[ 8] BCM 14 txd
    └─Flags:i p 0x0
[10] BCM 15 rdx
    └─Flags:i p 0x0
[11] BCM 17
    └─Flags:i p 0x0
/tmp/less1261 lines 1-21/119 23%
  
```

Option: **Layout für Header J8**

```

pi@pi6: ~
3.3V      [ 1][ 2] 5V
BCM 2 sda [ 3][ 4] 5V
BCM 3 scl [ 5][ 6] GND
BCM 4 gpclk0 [ 7][ 8] BCM 14 txd
GND       [ 9][10] BCM 15 rdx
BCM 17    [11][12] BCM 18 pwm0
BCM 27    [13][14] GND
BCM 22    [15][16] BCM 23
3.3V     [17][18] BCM 24
BCM 10 misi [19][20] GND
BCM 9  miso [21][22] BCM 25
BCM 11 sclk [23][24] BCM 8 ce0
GND      [25][26] BCM 7 ce1
BCM 0 id_sd [27][28] BCM 1 id_sc
BCM 5     [29][30] GND
BCM 6     [31][32] BCM 12 pwm0
BCM 13 pwm1 [33][34] GND
BCM 19 miso [35][36] BCM 16
BCM 26    [37][38] BCM 20 mosi
GND       [39][40] BCM 21 sclk
  
```

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.

```

pi@pi6: ~
Infos zu /sys/bus Geräten
Verfügbare Device ID's unter /sys/bus

SBusListDir(w1/devices)
Befehl: ls /sys/bus/w1/devices
SBusListDir(i2c/devices)
Befehl: ls /sys/bus/i2c/devices

w1/devices
10-000802e444d1
10-000802e45d3c
w1_bus_master1
i2c/devices
1-0040
i2c-1

Infos zum SBus-Objekt (gpiosb.h)

SBus-Objekt: Zugriff auf sysf Schnittstelle
SysDir   : /sys/bus      | SBus
Simulation : 0          | Simulationsflag
SBusIsOk() : 1          | Objekt bereit

Ende mit q
/tmp/less1261 lines 1-25/26 99%

```

Devices i2c

Menubefehl: 7 /dev/i2c-x Devices

```

pi@pi6: ~
Verfügbare i2c Devices

Infos I2c-Objekt (gpioid2.h)

I2c-Objekt: Zugriff auf /dev/i2c-X mit ioctl()
Simulation : 0          | Simulationsflag
I2cLst[]   : 1          | Anzahl der Geräte

Path=/dev/i2c-1 id=0x77 fd=6

i2cdetect -y 1

   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40: 40  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50: 50  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  77  --  --  --  --  --  --  --  --

Weiter mit Taste

```

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 Linkbibliothek 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 relativ und symbolisch sein. Die Bibliotheklinks können mit dem Programm `chelp` überprüft werden. Befehl: **Einstellungen / Projektlinks prüfen / Pfade der symbolischen Links in c/ prüfen**

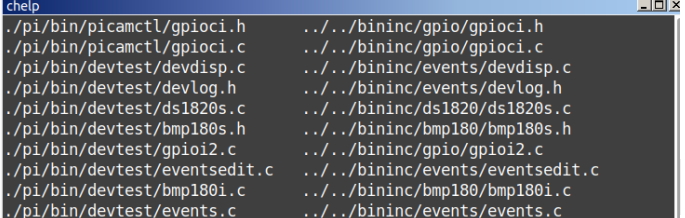
Das Bild zeigt einen Auszug der geprüften Links.

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
```



```
chelp
./pi/bin/picamctl/gpioci.h  ../../bininc/gpio/gpioci.h
./pi/bin/picamctl/gpioci.c  ../../bininc/gpio/gpioci.c
./pi/bin/devtest/devdisp.c  ../../bininc/events/devdisp.c
./pi/bin/devtest/devlog.h   ../../bininc/events/devlog.h
./pi/bin/devtest/ds1820s.c  ../../bininc/ds1820/ds1820s.c
./pi/bin/devtest/bmp180s.h  ../../bininc/bmp180/bmp180s.h
./pi/bin/devtest/gpioi2.c   ../../bininc/gpio/gpioi2.c
./pi/bin/devtest/eventseedit.c  ../../bininc/events/eventseedit.c
./pi/bin/devtest/bmp180i.c  ../../bininc/bmp180/bmp180i.c
./pi/bin/devtest/events.c   ../../bininc/events/events.c
```

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- und Hilfsfunktionen |
| devicesm.c | | Modulverzeichnis. Liste der verfügbaren C-Programme für Devices |
| makefile | | makefile |
| <hr/> | | |
| 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 | | |
| @eventseedit.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 Timerprogramme für Events |
| @devtimer.c | | |
| <hr/> | | |
| 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 das Chip-Interface /dev/gpiochip0 zum Steuern. |
| @gpioci.c | | Kopie von <linux/gpio.h> vom Raspberry Pi |
| @gpioci_pi.h | | 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 Temperatursensoren 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/gpiochip0
- ▷ **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/gpiochip0
│   └── 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
      return Bmp180iOpen(Dev); | DevId="0x77" i2c-Adresse
      | 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);

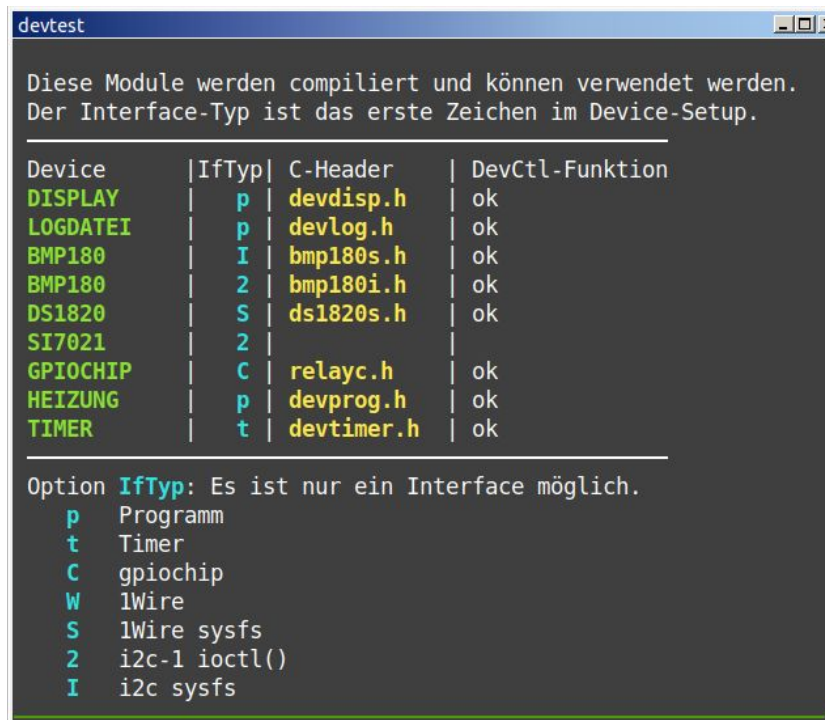
    default: Dev->ComStr=NULL; Dev->ComInt=DEVComIntNil; return false;
  }
  return false;
} // -----
```

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 Konfigurationsdatei für 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



```
devtest
Diese Module werden kompiliert und können verwendet werden.
Der Interface-Typ ist das erste Zeichen im Device-Setup.

Device | IfTyp | C-Header | DevCtl-Funktion
DISPLAY | p | devdisp.h | ok
LOGDATEI | p | devlog.h | ok
BMP180 | I | bmp180s.h | ok
BMP180 | 2 | bmp180i.h | ok
DS1820 | S | ds1820s.h | ok
SI7021 | 2 | | |
GPIOCHIP | C | relayc.h | ok
HEIZUNG | p | devprog.h | ok
TIMER | t | devtimer.h | ok

Option IfTyp: Es ist nur ein Interface möglich.
p Programm
t Timer
C gpiochip
W 1Wire
S 1Wire sysfs
2 i2c-1 ioctl()
I i2c sysfs
```

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`

```
devtest
Timer Config testen

Die Konfigurationsdatei des Device-Timers ohne Loop testen.

DevLst[7]
Device : TIMER,PROG,1 | Rem: Device Timer
Software: devtimer.h
Status : 0 | Index für TimerLst[]

TimerLst[0] | Timer-Daten zum Device Modus
DevMode : 1 | Device Modus
EventN : 6 > RW | Event für Exec > Event-Name
ChkTime : 2025-01-13 23:56:00 | Event CheckTime
ChkCmd : 1 | Event CheckCmd
Config : /home/guenther/c/pi/bin/devtest/bin/_devtest/test.timer
NxtPos : 291 | Config: Nächste Leseposition
BlkPos1 : 268 | Config: Position von Befehl 1 im Block
EndTime : 2025-01-13 23:59:59 | Config: Endzeit für Block
Line : 1 // Befehl: ein | Config: Last getline()
LineNr : 13 | Config: Aktuelle Zeilennummer
LineNr1 : 12 | Config: Zeilennummer von Befehl 1 im Block
fConf : 0x5591216b6430 | Config: Filepointer

Befehle: Read Config | Test Timer-Check Auto/Kurz/Lang | Befehle
```

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 nach Blockzeiten aufsteigend definiert werden.

```
// Beispiel 1
s t14, 6:0:0, 6:18:0      s Blockanfang | An den Wochentagen 1=Montag und 4=Donnerstag von 6:0:0 bis 6:15:0
x 1 // Exec 1           x 1 | Exec 1 (On) aufrufen. Das Event wird mit Open="Event='Name'" festgelegt.
w 18:0 // Warte 18 min  w 18:0 | Wartezeit bis zum nächsten Event | Wunsch 18 Minuten | Maximal Blockende
n 0 // Exec 0 bei Blockende n nächster Block | Zeitpunkt 6:18:0 | Event Exec 0 (Off) aufrufen
```

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

```
// Beispiel 2
s t06, 6:0:0           s Blockanfang | t06 nur am Sonntag 0 und Freitag 6 | Zeitpunkt 6:0:0
x 3 // Sensor abfragen x 3 | Exec 3 | Beispiel Sensorabfrage 0 | keine Wartezeit
w 0                    weiter mit dem nächsten Block
n
```

```
// Beispiel 3
s 2024-12-16 7:0:0, 2024-12-17 7:5:0 s Blockanfang | Zeitraum 2024-12-16 bis 2024-12-17 | Zeit von 7:0:0 bis 7:5:0
x 1                     x 1 | Event-Exec 1
w 20                    w 20 | 20 sec Wartezeit
x 2                     x 2 | Event-Exec 2
w 0                     w 0 | keine Wartezeit
n                       weiter mit dem nächsten Block
```

```
// Beispiel 4
s 2024-11-30 7:0:0    s Blockanfang | Zeitraum ab 2024-11-30 | Täglich um 7:0:0
x 3                   x 3 | Exec 3
w 0                   w 0 | keine Wartezeit
n                     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 auf 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 Zeilennummern verfolgt werden.

```
devtest
Timer Befehle für Config

Befehlszeilen in Config: Befehlszeichen [Parameter]

Befehlszeichen | Parameter
s Start Block | [t[0-6],] [StartDatum] Startzeit,[, [Enddatum] EndZeit]
x Event-Exec | Parameter für Event-Exec
w Wartezeit | Zeitstring
l Loop | [Parameter] Block wiederholen, Event-Exec am Ende
n Nächster Block | [Parameter] Event-Exec am Ende
/ Rem, Blank oder Leer

Befehlszeile s:
, Trennzeichen Items
t Wochentage: t01 nur am Sonntag und Montag
: Trennzeichen Zeit
- Trennzeichen Datum
Beispiel: t01, 24-05-12 6:0:0, 7:18:0

Interne Steuerbefehle:
e Blockende: lnend
~ End Of File: eof
f Fehler: err
Blank: ' '
```

```
devtest
Timer testen | TimerTestReadCfg()

Config: /home/guenther/c/pi/bin/devtest/bin/_devtest/test.timer
BEFEHL: / s x w l n | Das erste Zeichen derZeile.
Zeile : BEFEHL[ Parameter]

Die Configdatei wird zeilenweise gelesen und decodiert.

1 / Line=' Timertest für Bewässerung'
2 / Line=' Datei: test.timer'
3 / Line=' Test: Ab 6 Uhr einschalten. '
4 / Line=''
5 s Line='6:0:0, 23:59:59 // Steuerblock'
6 x Line='1 // Befehl: ein'
7 w Line='10 // Wartezeit'
8 x Line='0 // Befehl: aus'
9 w Line='10 // Wartezeit'
10 l Line='0 // loop, Befehl: aus'
11 ~ eof
```

```
devtest
Test Timer-Check | TimerTestChkTimeAndCmd1()

Config : /home/guenther/c/pi/bin/devtest/bin/_devtest/test.timer
TimeNow: 2025-01-05 05:00:00
Die Config-Datei lesen und für den nächsten Check ChkTime und
ChkCmd berechnen. TimeNow hochzählen und bei TimeNow==ChkTime
'Exec' anzeigen und den Check neu berechnen.

Start : 2025-01-05 05:00:00
TimeNow 2025-01-05 05:00:00 > Exec
▶ Zeile 6 Nächste ChkTime 2025-01-05 06:00:00 | ChkCmd 1
TimeNow: 2025-01-05 05:00:00
TimeNow: 2025-01-05 05:00:01
TimeNow: 2025-01-05 06:00:00
TimeNow 2025-01-05 06:00:00 > Exec
▶ Zeile 8 Nächste ChkTime 2025-01-05 06:01:00 | ChkCmd 0
TimeNow: 2025-01-05 06:00:00
Auto TimeNow | RETURN | ESC ?
```

```
devtest
Timer: TimerTestChkTimeAndCall() testen

Config : /home/guenther/c/pi/bin/devtest/bin/_devtest/test.timer
TimeNow: 2025-01-05 06:57:04
Timer Definitionen aus Configdatei fortlaufend auswerten.
Rückgabe: ▶ Zeilennummer, ChkTime und ChkCmd.

TimeNow: 2025-01-05 06:57:04
TimerReadConfig Line=6 NxtCmd=n 2025-01-05 06:57:04
1 Cmd=/ |
2 Cmd=/ |
3 Cmd=/ |
4 Cmd=/ |
5 Cmd=s | Block Anf | '6:0:0, 23:59:59 // Steuerblock'
TimerParseBlkAnf() '6:0:0, 23:59:59'
TagAnf 2025-01-05 00:00:00
TagEnd 2025-01-05 23:59:59
BlkAnf 2025-01-05 06:00:00
BlkEnd 2025-01-05 23:59:59
NxtCmd=x
6 Cmd=x
Cmd_x: 1
ChkOk: ChkTime 2025-01-05 06:57:04 | ChkCmd 1 | NxtCmd w
▶ Zeile 6 ChkTime 2025-01-05 06:57:04 | ChkCmd 1
EndTime 2025-01-05 23:59:59 | NxtCmd=w
```

Beispiel: Bewässerung

Erweiterung: Bewässerung an bestimmten Wochentagen.

Hardware: Zum Einschalten der Bewässerung wird ein Relais an Pin 15 des Raspberry 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 folgenden 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 Logfunktionen kann mit einem Timer durchgeführt werden.

Beispiel: Konfiguration des Timers: [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 <v.schardinger@gmx.net>
 *
 * 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.
 */
```