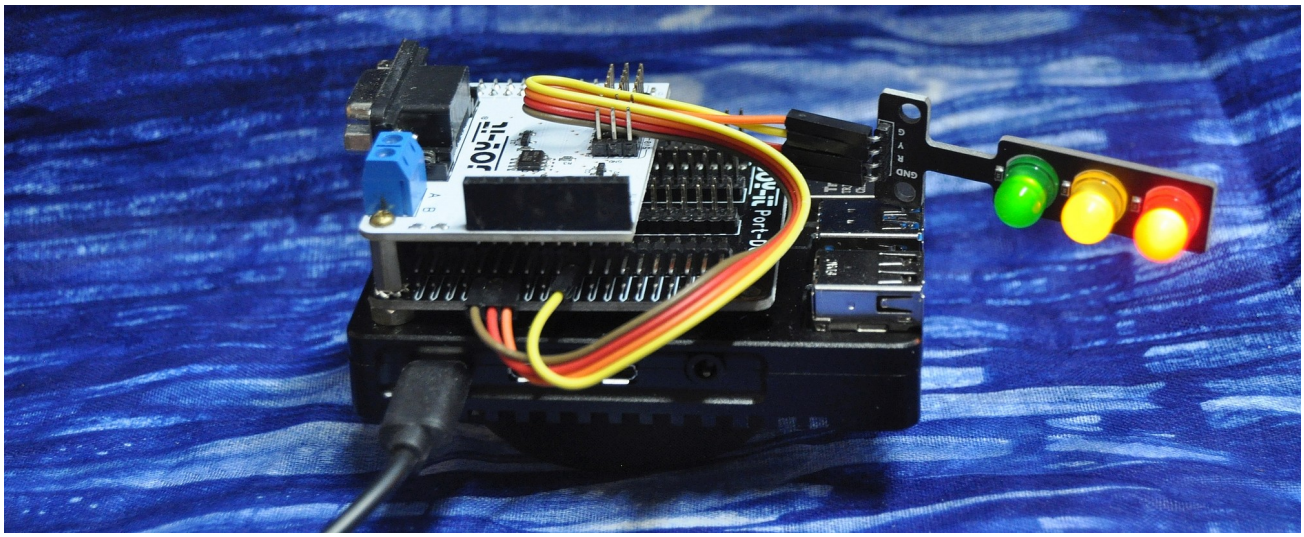

Java statt C

Raspberry Pi GPIO mit Java

Albrecht Weinert

Für kleine Steuerungsaufgaben werden zunehmend serienmäßige, preiswerte Standardkleincomputer, wie der Raspberry Pi, eingesetzt. Unter entsprechenden Randbedingungen und mit der Programmiersprache C geht dies durchaus. Aber geht es – das heißt insbesondere auch die Ansteuerung von Ein- und Ausgabe (general purpose IO) – auch mit Java?

Echt jetzt – Automatisieren mit „Spielzeug“

Vernetzbare serienmäßige Standardkleincomputer verdrängen stellenweise schon die klassische Automatisierungstechnik. Unter solchen massenhaft gefertigten Kleinsystemen zählen der Raspberry Pi und seine Brüder schon zu den Großen. Viel häufiger sind daumennagelgroße Mikrocomputersystemchen mit integriertem WLAN und einem paar Dutzend I/O-Ports. Wenn solche vernetzten Kleinsteuerungen mit VPN oder einen Cloud-Dienst via Internet erreichbar sind oder gemacht werden könnten, kann man sie unter dem zugkräftigen Begriff Internet of Things (IoT) propagieren.

Vorteile der Standardsysteme (COTS components of the shelf) gegenüber industriellen Automatisierungssystemen sind viel preiswertere Rechen- und Netzwerkhardware, der Einsatz von Standardbetriebssystemen wie das freie Linux, Standard-Sprachen und -Entwicklungssysteme darunter freie von teilweise höchster Qualität. Alles ist in die „normale IT-Welt“ integrierbar und mit „normalem“ IT- und Informatik-Wissen handhabbar.

Die Welt der Automatisierer, der Automatisierungssysteme, ihrer Laufzeitsysteme, Sprachen (IEC 61131, KOP, FUB, AWL, SFC) und Entwicklungsumgebungen und die Welt der Informatiker (und Java-Menschen) haben im Allgemeinen wenig Berührungspunkte. Wer von der Informatik oder Elektronik zu Automatisierungsaufgaben kommt, wird sich meist heftig sträuben, veraltete Sprachen und „komische“ Laufzeit- und Entwicklungssysteme – dem Stand der IT oft sehr weit hinterher – einzusetzen. Zudem lockt ja die Einsetzbarkeit preiswerter, verbreiteter Standard-Hardware.

Die Stärke von traditionellen Automatisierungsgeräten liegt in der Qualität der Hardware bei Verfügbarkeit, Sicherheit und Grunddiagnosefunktionen. Wir finden EMV-feste Ein- und Ausgänge, gepufferte und überwachte Versorgung der Verarbeitungseinheiten und der I/O, leicht erfassbare, normgerechte LED-Farben (rot = Störung), elektrische Sicherheit und viel Gutes mehr.

Demgegenüber geht ein „RasPi“ kaputt, wenn ein herausgeführten Port mal kurz mehr als 3,5 V „sieht“. Diese Schwäche teilt er mit fast allen seinen Kollegen (Arduino, ESP …). Wenn man im realen Betrieb einsetzbare Systeme machen will, muss (!) man das Notwendige hinzu bauen. Dies wissend und beachtend, spricht allerdings nichts dagegen, anspruchsvolle „echte“ Automatisierungs- und Prozesssteueraufgaben mit einem Raspberry zu verwirklichen.

Ja – dann aber professionell und mit C

Zum effektiven Arbeiten macht man möglichst wenig lokal auf dem Pi, zumal dieser nur „headless“ bzw. „lite“ für Steuerung und Echtzeit einsetzbar ist. Auch wegen vieler anderer Vorteile nutzt man also eine Entwicklungs-Workstation im selben LAN/WLAN wie der Pi. Sie bietet:

- raspberry-gcc for Windows (raspberry-gcc4.9.2-r4.exe oder neuer);
  damit hat man auch die Linuxtools wie grep, make usw. auf Windows
- Eclipse CDT; mit Bezug auf raspberry-gcc, make project ohne make-file-Generierung
- git client (und svn client)
- Filezilla und WinSCP
- Win32DiskImager
- Putty

Zu Einzelheiten zum Einrichten siehe [weRasp4rem]. Natürlich geht das auch mit einem Linux-PC nur gibt es dann kein WinSCP (***Win****dows* ***S****ecure* ***Cop****y*) und Skripting von FTP.

```
27.05.2020   1.853.882.368  2020-05-27-raspios-buster-lite-armhf.img
05.05.2019               0                  ssh
05.05.2019             254  wpa_supplicant.conf
```

Listing 1: Zutaten für die µSD-Karte für die Einrichtung eines Pi; das ist dann seine „Platte“

Man braucht einen Raspberry mit Raspbian lite, der auf Anhieb über das WLAN via SSH vom PC bedienbar sein soll. Dazu kopieren Sie das Raspbian-Lite-Image mit dem Win32DiskImager auf eine µSD-Karte. Noch auf dem PC kopieren Sie die leere Datei ssh ins root-Verzeichnis (/) der SD-Karte und die Datei wpa_supplicant.conf nach /boot/ (s. Listing 1).

Die leere Datei ssh bewirkt, dass der SSH-Dienst ab dem allerersten Start des neuen Systems im Pi aktiviert wird. Mit der Datei wpa_supplicant.conf spezifizieren Sie das LAN oder WLAN, in dem sich Ihr Entwicklungs-PC und ein DHCP-Server befinden (s. Listing 2). Dort können Sie bei psk auch das Klartextpasswort Ihres Netzwerks in Anführungszeichen eintragen. Besser ist natürlich die mit dem Linux-Tool wpa_passphrase verschlüsselte Variante.

```
# Datei wpa_supplicant.conf in der Boot-Partition (Raspbian Stretch)
country=DE
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
  ssid="weAut2Netz"
  psk=c535a23fbf334cafe95338affe795c4711b9e9f129e62b6cc4094faf89914d51
}
```

Listing 2: wpa_supplicant.conf; Achtung: keine Leerzeichen beim=; Unix-text-format (nur LF)

Nun wird der Pi mit der µSD-Karte im [W]LAN gestartet. Nach Ermitteln der IP-Adresse (neuer Client des DHCP-Servers) oder über den Default-Namen raspberrypi können Sie sich mit der Terminalsoftware PuTTy ([ptty4win]) einloggen und hostname und Passwort ändern. Auch legen Sie ein Verzeichnis ~/bin/ an.

Für den Zugriff auf die 26 I/O-Ports des 40-poligen Pfostenverbinders installieren Sie noch pigdpio ([pigpiodCface], [pigpiodCsock]) gemäß Listing 3.

```
pi@pi:~ $ wget https://github.com/joan2937/pigpio/archive/master.zip
pi@pi:~ $ cd pigpio-master
pi@pi:~ $ make
pi@pi:~ $ sudo make install
pi@pi:~ $ sudo /usr/local/bin/pigpiod  -s 10 # server/daemon start
```

Listing 3: Installation von pigpio (siehe auch [weRasp4rem])

Den Start des Servers sollten Sie noch mit `sudo crontab -e` und Anfügen einer Zeile

```
@reboot /usr/local/bin/pigpiod  -s 10
```

automatisieren.

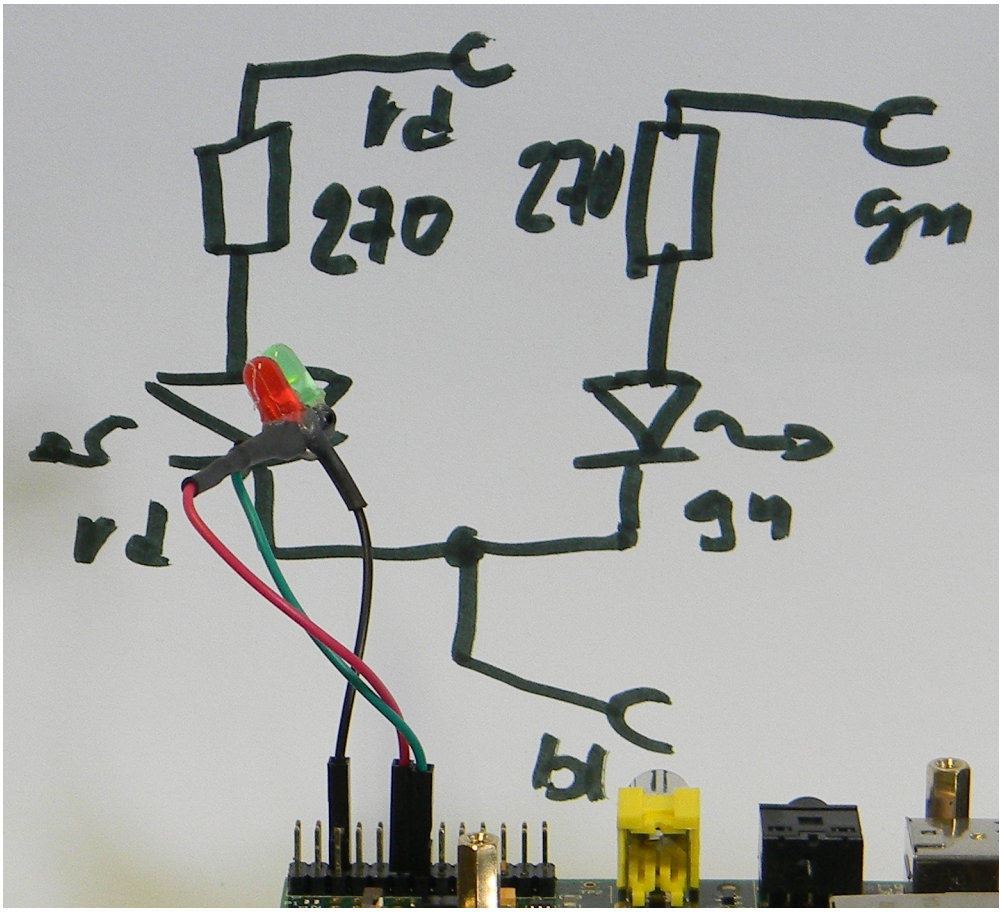

Abb. 1: LEDs am Pi

Wenn Sie nun noch zwei oder drei LEDs, rot, grün und gelb, an die Pins 11, 13 und 22, mit Masse an Pin6 z.B. anschließen (mit Vorwiderständen à la Abb. 1), können Sie das Demoprogramm /rdGnPiGpioDBlink.c (geholt von [weRaspGit]; s. Listing 4) ausprobieren.

```
D\: mkdir \git
D\: cd \git
D:\git>git clone https://github.com/a-weinert/weAut.git
D:\git>cd D:\git\weAut\rasProject_01part
@REM Weg 1a  Cross-Bauen  Test lokale winRaspi-Installation GCC make etc.
D:\git\weAut\rasProject_01part: make PROGRAM=rdGnPiGpioDBlink clean all
@REM Weg 1b Übertragen zum Zielsystem
@REM   Edieren Sie vorher gemäß Ihrem [W]Lan und Pi die Dateien
@REM   makeTarg_raspi61_setFTP.mk  und  makeTarg_raspi61_settings.mk
D:\git\weAut\rasProject_01part: make PROGRAM=rdGnPiGpioDBlink progapp
D:\git\weAut\rasProject_01part: make PROGRAM=rdGnPiGpioDBlink clean
```

Listing 4: „Cross-“ Übersetzten, -Bauen und Übertragen in C

Wenn das läuft, dann haben Sie eine Cross-Toolchain vom PC zum Raspberry Pi!

Auch falls Sie an C/GCC nicht interessiert sind, sollten Sie dennoch die mit C gebaute I/O-Demo laden und laufen lassen. Wo das nicht geht, brauchen Sie mit Java nicht anfangen. Wechseln Sie in das Verzeichnis D:\git\weAut\binaries (des in Listing 4 erzeugten GIT clones). Von dort transferieren Sie (mit Filezilla, WinSCP o.ä.) die Datei rdGnPiGpioDBlink in das Verzeichnis ~/bin/ Ihres Raspberry. Mit PuTTy eingeloggt starten Sie es nach Erstellen der erwarteten Lock-Datei (s. Listing 5).

```
pi@piWLan97:~ $ chmod +x bin/rdGnPiGpioDBlink # once after FTP-tranfer
pi@piWLan97:~ $ rdGnPiGpioDBlink &
                  ## maul maul lock file
pi@piWLan97:~ $ touch bin/.lockPiGpio
pi@piWLan97:~ $ rdGnPiGpioDBlink &
[1] 2300
pi@piWLan97:~ $ ps aux | grep rdGnPiGpioDBlink
```

Listing 5: Starten des C-Programms mit lock-Datei

Nun sollte das C-LED-blink-Program endlos im Hintergrund laufen. Merken Sie sich die Prozessnummer für das Killen, oder ermitteln Sie sie mit ps aux. Das Starten einer zweiten Instanz dieses Programms oder entsprechender Programme wird über die Lock-Datei verhindert.

pigpio-Dämon – Hintergrund

Im C-Beispiel rdGnPiGpioDBlink verwendeten wir die pigpio-Bibliothek mit dem C-API zum Socket-Interface ([pigpiodCface]). Dies nutzt der Autor fast ausschließlich für seine Steuerungsanwendungen mit dem Pi. Anderen verbreitete und getestete I/O-Bibliotheken belasten Nutzer mit Dingen wie

- Initialisierung der GPIO-Speichernutzung (memory mapping)
- Adaptieren an wechselnde (virtuelle) Adressen
- Kämpfen mit Zugriffsrechten

Kein Betriebssystem mit etwas Selbstachtung wird Nutzersoftware an die I/O lassen, so auch Raspian. Also müssten alle Steuerungsprogramme – auch solche in der Testphase – mit sudo laufen. Bei manchen Raspians geht einfaches Lesen und Schreiben einiger I/O-Ports auch so, aber bei speziellen Einstellungen, alternativen Funktionen oder ähnlichem ist das wieder vorbei.

Die pigpio-Bibliothek ([pigpiodCface]) hat einen anderen Ansatz. Sie definiert einen Server oder Dämon, der sich um alle Initialisierungen kümmert und die I/O gemäß Aufträgen bedient. Dieser Server muss mit sudo gestartet werden. Das haben wir oben mit crontab und @reboot-Eintrag automatisiert. Programme, die wie rdGnPiGpioDBlink diesen Server nutzen, brauchen kein sudo.

Alles OK – aber nun doch bitte endlich mit Java

Auf die mit dem Demoprogramm gezeigte Weise geht die I/O mit dem Raspbrerry – und das auch in viel größerem Maße. Nun wollen wir hierauf aufbauend den entsprechenden Einstieg mit Java. Bei Schnittstellen in die Außenwelt, Sensoren, Aktoren war Java nie gut und Sun hatte das nie auf dem Schirm. Ein Symptom dieser Haltung war das plötzliche ersatzlose Streichen der commApi durch Sun. Da verloren einige Java-Serveranwendungen mit RS485-Schnittstellen zu industriellen Steuerungen die offizielle Grundlage. Voranbringen der Idee „Automatisieren mit Java“ geht irgendwie anders, aber wir packen es auf dem Raspberry an.

Der Schritt Null, das Installieren von Java, ist einfach:

```
pi@piWLan97:~ sudo apt install openjdk-8-jdk
pi@piWLan97:~ $ java -version
openjdk version "1.8.0_212" (build 1.8.0_212-8u212-b01-1+rpi1-b01)
OpenJDK Client VM (build 25.212-b01, mixed mode)
pi@piWLan97:~ $ which java
/usr/bin/java
```

Mit Java 8_212 haben wir auf dem Pi ein JDK/JRE ohne den Modul-Kram und mit funktionierenden installed extensions. Durch Verfolgen der Links in /usr/bin/java zu ihrem Ziel finden wir die wirkliche Java-Installation in

`/usr/lib/jvm/java-8-openjdk-armhf/`
mit den JVM-Werkzeuge (java …) in

`/usr/lib/jvm/java-8-openjdk-armhf/jre/bin/`
und den übrigen (javac ...) in

`/usr/lib/jvm/java-8-openjdk-armhf/bin/`
Um die link-Orgie zu kürzen können wir an das Ende von .bashrc

```
export PATH=/usr/lib/jvm/java-8-openjdk-armhf/bin/:$PATH
```

anfügen. Zum Testen und für später installieren wir noch Frame4J:

```
pi@piWLan97:~
    wget  https://weinert-automation.de/software/frame4j/frame4j.jar
pi@piWLan97:~ sudo cp frame4j.jar
                    /usr/lib/jvm/java-8-openjdk-armhf/jre/lib/ext/
pi@piWLan97:~ java AskAlert
pi@piWLan97:~ java ShowProps
pi@piWLan97:~ java UCopy -help -de
```

Wenn die drei Java-Tools laufen, weiß man, dass auch komplexere Java-Anwendungen auf dem Pi angemessen arbeiten und dass installed extensions gehen.

Ein Blink-blink als „Hello process control world“

Für den Java-Anschluss an den Dämon/Server der vielfach bewährten pigpio-Bibliothek findet man eine Handvoll kleiner Projekte; sie sind letztlich alle nicht nötig, es geht auch mit „pure Java“ im nächsten Kapitel.

Eines dieser Projekte ist von ist von Neil Kolban ([nkolbanJp]). Es liefert auch einen C-Wrapper mit JNI für den Nicht-Socket-Zugang. Für den verwendeten Socket-Zugang benötigen wir eine passende .jar; für den Nicht-Socket-Zugang brauchten wir auch die C-Bibliothek:

```
## not used  root root  206623 2019-05-12  libJPigpioC.so
-rw-r--r-- 1 root root   26260 2019-05-19  jpigpio.jar
-rwxrwxr-x 1 pi   pi     36768 2019-05-14  justLock
```

In Kolbans GIT-Repository finden Sie die Quellen für die jar. Alternativ holen Sie sich diese fertig vom Autor ([weRaspGit], Verzeichnis binaries)) und installieren sie mit:

```
sudo cp jpigpio.jar     /usr/lib/jvm/jdk-8-oracle-arm32-vfp-hflt/jre/lib/ext/
## not used sudo cp libJPigpioC.so /usr/lib/jvm/jdk-8-oracle-arm32-vfp-hflt/jre/lib/arm/
cp   justLock     ~/bin/
chmod  +x    ~/bin/justLock
```

Von dort holen Sie sich auch die kleine Hilfsanwendung justLock. Zu derem Hintergrund siehe bitte [weRaspBlog] – man kann das inkompatible Verhalten von `java.nio.channels.FileLock` durchaus für einen Fehler des Linux-Ports des JDK halten.

Jetzt holen Sie noch die Quelle RdGnJPiGpioDBlink.java und setzen Sie (package-konform) ins Verzeichnis ~/de/weAut/tests/ (oder arbeiten in <gitClone>/weAut/frame4j_part). Und:

```
pi@piWLan97:~ $ javac de/weAut/tests/RdGnJPiGpioDBlink.java
pi@piWLan97:~ $ java de.weAut.tests.RdGnJPiGpioDBlink
RdGnJPiGpioDBlink start
RdGnJPiGpioDBlink getIOlock error: can't lock the lock file
RdGnJPiGpioDBlink shutdown
pi@piWLan97:~ $
```

Gemaule wegen I/O-Lock beziehungsweise Lock-Dateien? Dann müssen Sie erst Ihr im Hintergrund noch laufendes C-Programm rdGnPiGpioDBlink „killen“. Dann aber haben wir ein dazu völlig äquivalentes Java-Programm auf dem Pi, das auch noch dort übersetzt wurde.

Raspberry PI I/O mit pure Java

Eigentlich sind wir fertig. Kolbans Bibliothek liefert den Zugang zur vielfach bewährten Socket-Schnittstelle von pidpiod. Allerdings umfasst sie auch eine C-Schicht-JNI für den Nicht-Socket-Zugang, was sie umfangreich und komplex macht. Sie ist teilweise spärlich dokumentiert, und an keiner Stelle sieht man die Vermeidung von Wegwerfobjekten (vgl. Anmerkungen unten).

Diese und andere Punkte ließen den Wunsch nach einer kompakteren reinen Java-Lösung aufkommen. Der entsprechende Port von C heißt RdGnPiGpioDBlink.java (ohne das J für jpigpio). Wie oben holen, dann übersetzen und starten:

```
pi@piWLan97:~ $ sudo apt-get update
pi@piWLan97:~ $ sudo apt-get upgrade
pi@piWLan97:~ $ sudo apt-get install git-core  # only if no Git, yet
pi@piWLan97:~ $ git clone https://github.com/a-weinert/weAut.git
pi@piWLan97:~ $ cd weAut/frame4j_part
pi@piWLan97:~.. $ javac de/weAut/tests/RdGnPiGpioDBlink.java
pi@piWLan97:~.. $ java de.weAut.tests.RdGnPiGpioDBlink
```

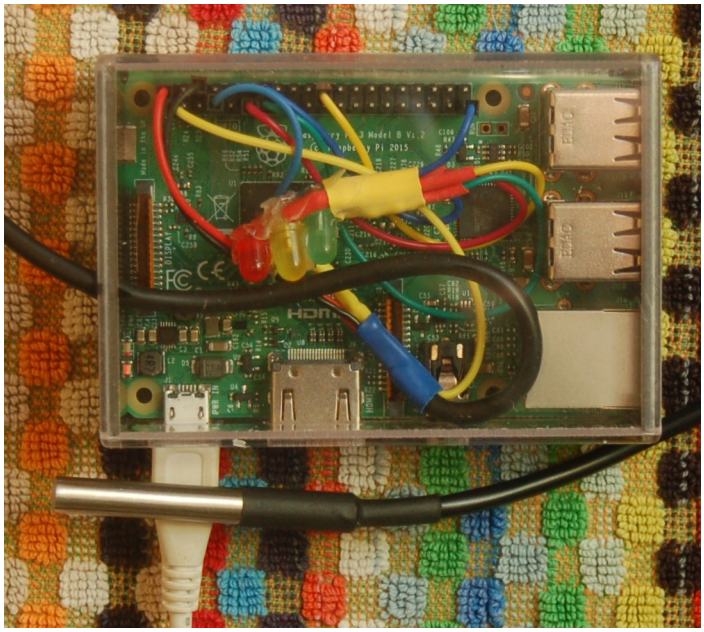

Abb. 2: Aufbau für “xyblink” und 1-wire

Das kleine 1-Draht-Thermometer-Demo de.weAut.tests.Pi1WireThDemo zeigt I/O ohne pigpio:

```
pi@piWLan97:~ $ javac de/weAut/tests/RdGnPiGpioDBlink.java
pi@piWLan97:~ $ java de.weAut.tests.Pi1WireThDemo 28-02119245cd92
```

Die Handhabung von Übertragung, Übersetzten und Start ist wie bei den anderen Demos. Allerdings müssen Sie ein 1-wire-Thermometer anschließen (s. Abb. 2 und Listing 8 unten), und Sie müssen dessen ID (28-02119245cd92 ist das abgebildete Teil) herausbekommen. Also 1-wire oder allgemeiner „device as file“ geht also auch mit Java.

Java auf dem Raspberry PI – Ein paar Punkte noch

Die gute Nachricht: Es geht auch mit „pure Java“. Die Portierung von seit Jahren eingesetzten „pure C“- Ansätzen zeigt es. Dass es sich hier statt der „großen“ Anwendungen um kleine Tests und Demos für GPIO, watchdog, Linux devices as file handelt, ist fast eine Frage der Skalierung oder des Fleißes beim Portieren komplexer Systeme. Betrachten wir noch ein paar Punkte.

Die in der Eingangsbetrachtung zu Industriellen Automatisierungssystemen vs. Standard-Kleincomputer und -Netzwerkhardware (of the shelf) genannten Aspekte und Anforderungen zu ***Verfügbarkeit*** und ***Sicherheit*** gelten auch für Lösungen in C.
Diese wichtigen Fragen sind jedenfalls kein Java-Problem.

Verglichen mit C könnte Java aber Probleme mit der ***Ausführungsgeschwindigkeit*** und mit der Echtzeit haben. Die kolportierte Langsamkeit von Java verglichen mit C oder gar Assembler ist in Zeiten fortgeschrittener „JIT“-Compiler ja eher tradierte Nachrede – inzwischen liegt Java hier zum Teil vorne. Und der langsame Start von Java-Anwendungen spielt keine Rolle. Wie die vergleichbaren C-Automatisierungsanwendungen auf dem Raspi sollen sie ja monatelang 24/7 laufen. Allerdings ist es nicht leicht (und war es nie), zur jeweiligen Qualität des Java-Laufzeitsystems und des JIT-Compilers verlässliche Daten zu bekommen. Trifft die heute mit Recht angenommene Qualität auch auf ein („Schrumpf-“?) Java 1.8.0_xy auf dem Raspberry zu?

Selbst wenn eine Java-Anwendung bezüglich hinreichender Geschwindigkeit gemessen und lange getestet wurde, bleibt bezüglich ***Echtzeit*** die Frage der Stabilität dieses Verhaltens oder der Abwesenheit von sporadischen Ausreißern. Hier gilt der Garbage-Collector als Staatsfeind Nummer 1. In einem Sicherheitsarbeitskreis wurde zur Automatisierung mit objektorientierten Sprachen gefordert, dass im zyklischen Betrieb keine Objekte mehr erzeugt werden dürfen. Selbst wenn diese Forderung so hart und für allgemeine Steuerung heute kaum noch so stellen würde, ist dies ein löbliches Prinzip – insbesondere für 24/7-Anwendungen. Es befreit von den Risiken des Garbage-Collectors und der Speicherverwaltung, und es liegt auch den Beispielen (hier und [weRaspGit]) zugrunde.

So wurde für periodenfeste Delays eigens eine Klasse LeTick eingeführt, die eine absolute long-Millisekundenzeit (für ThreadLocal) in einem Objekt kapselt (s. Listing 7). Von der Funktion her hätte die im allerersten Ansatz verwendete Klasse Long genügt und mit auto boxing auch die Lesbarkeit verbessert. Da Long immutable ist, bedeutet jede Änderung aber ein Wegwerfobjekt.

```
// static public final ThreadLocal<Long> lastThTick // no Long thwrow away
   static public final ThreadLocal<LeTick> lastThTick
                  = new ThreadLocal<>();

   public final class LeTick {
      long tick;
      public long getTick(){ return tick; }
      public void setTick(final long tick){ this.tick = tick; }

      public long add(final long adv) { return this.tick += adv; }

      public LeTick(long tick) {this.tick = tick; }
   } // LeTick
```

Listing 7: LeTick, im Wesentlichen ein mutable Long

In Fällen wie diesem (s. Listing 7) lässt sich Garbage-Collection und Speicherverwaltung durch gezielte Wiederverwendung veränderbarer Objekte verhindern. Bei dem verbreiteten Linux-Ansatz „Gerät als Datei“ (device as file) sieht es dabei aber schlecht aus. Listing 8 zeigt das Auslesen eines 1-wire-Thermometers, das es auch edelstahlgekapselt (s. Abb. 2) für die üblichen Einsteckröhrchen gibt.

```
String line1;
String line2;
BufferedReader thermometer;
File  thermPath = new File(
  "/sys/bus/w1/devices/w1_bus_master1/" + args[0] + "/w1_slave");
for(;runningOn;) { // zyklischer Dauerbetrieb
  thermometer = new BufferedReader(new FileReader(thermPath));
  line1 = thermometer.readLine();
  line2 = thermometer.readLine();
  // Auswertung
```

Listing 8: Auslesen eines 1-wire Thermometers, gekürzt um exception handling und Auswertung

Der zugehörige Linux-Device-Treiber liefert ein Messergebnis in zwei Textzeilen [sic!]. Nach deren Einlesen schließt sich die Datei automatisch. Deswegen muss das Dateiöffnen in Listing 8 in die den Dauerbetrieb andeutende Endlosschleife. Jede Messung erzeugt also:

- 1 BufferedReader,
- 1 FileReader und indirekt
- 1 FileInputStream,
- 1 FileDescriptor usw. und schließlich direkt noch
- 2 String-Objekte.

Von diesen mindestens 6 Objekten pro Messung ließen sich mit einem byte-Array (und verminderter Lesbarkeit) nur die Reader und die Strings vermeiden. Willkommen Wegwerfgesellschaft! Der Watchdog, auch als device as file implementiert, ist hierbei wesentlich besser: Er – das heißt in Java sein OutputStream – bleibt die ganze Zeit zum Schreiben geöffnet.
Aber hierauf, sprich auf im System verankerte Device-Treiber, haben wir keinen Einfluss.

Eine Falle ist noch die auf Linux-Systemen unvermeidliche Kodierung ***UTF8***. Dies ist ein Problem, wenn man größere Ausgangsprojekte auf PCs oder Workstations hat, die ISO8859-1, -15 beziehungsweise die analogen Windows-Codepages wie CP850 haben. Hier einzelne Dateien oder gar das Ganze für ein kleines Zusatzprojekt zu ändern, erzeugt oft ein großes Kuddelmuddel. In einer ISO8859-1-Quelle bewirkt nun allerdings ein Umlaut oder ein Grad-Zeichen (0°C), dass sie auf den Raspberry übertragen dort nicht compiliert wird; Listing 9. Andere javac-Fehler wird man auf dem Raspberry kaum sehen, da sie beim Übersetzen auf dem PC beziehungsweise dort schon im Eclipse sichtbar sind.

```
pi@piWLan97:~ $ javac de/weAut/tests/Pi1WireThDemo.java
de/weAut/tests/Pi1WireThDemo.java:99: error: unmappable character for encoding UTF8
        erg.append(       "   0.000C"); // position
1 error
```

Listing 9: Übersetzen auf dem Raspberry mit Fehler

Hierfür gibt es zwei Workarounds. Das erste ist Compilieren auf dem PC und Übertragen der .class-Dateien auf den Raspberry (s. Listing 10, 11).

```
D:\eclipse18-09WS\frame4j>javac.exe -d build  de\weAut\tests\*.java
D:\eclipse18-09WS\frame4j>winscp.com /script=progTransWin /parameter
pi:piPass 192.168.178.97 de/weAut/tests build\de\weAut\tests\*
```

Listing 10: Übersetzen auf dem PC und Übertragen auf den Raspberry

```
# Transfer a program respectively a class to the target machine
# Copyright 2017, 2019 Albrecht Weinert          a-weinert de
# $Revision: 205 $ ($Date: 2019-05-30) $)
# param 1 : user:passwd  (ftp user) e.g. pi:raspberry
# param 2 : target machine (IP)    e.g. 192.168.178.97
# param 3 : target directory (within Pi user's ~) e.g. de/weAut/test
# param 4 : program     e.g. Pi1WireThDemo.class (wildcards allowed)
# use by: winscp.com /script=progTransWin
##         /parameter pi:raspberry 192.168.178.97 bin rdGnBlinkBlink
open sftp://%1%@%2%
cd %3%
option batch continue
option confirm off
put %4% -preservetime -permissions=775
exit
```

Listing 11: FTP auf der Windows-Kommandozeile

Siehe da: Die auf Windows mit Java 1.8.0_141 übersetzte ISO8859-1-Quelle läuft auf dem Linux UTF8-Raspberry mit Java 1.8.0_65. UND das „°C“ wird auf der Normalausgabe (PuTTy) richtig angezeigt.

Erstaunlich? Nein! Genau DAS ist Java. Für C-Entwicklung auf dem PC und dort Compilieren benötigt man für jedes Zielsystem spezielle Cross-Compiler und spezifische Werkzeugsätze. Für Raspberrys (BCM-Prozessoren und Linux-Derivate) heißen sie arm-linux-gnueabihf-gcc usw. Und alle diese Zusatztools müssen widerspruchsfrei im Pfad verankert und mit Eclipse bekannt gemacht werden. Selten gehen diese Vorgänge ohne Nebenwirkungen und Kollateralschäden. Für Java haben wir eh C:\util;C:\util\jdk\bin; im Pfad und Eclipse kennt es fast unvermeidbar.

Trotz der angeklungenen Begeisterung für die Plattformunabhängigkeit und die einfache Cross-Compilierung sei der zweite Workaround geschildert: Übertragen der ISO8859-1-Quelle auf den Raspberry und dann dort umkodieren, übersetzen und testen (s. Listing 12).

```
pi@piWLan97:~ $ java UCopy -toUTF8 de/weAut/tests/Pi1WireThDemo.java
pi@piWLan97:~ $ javac de/weAut/tests/Pi1WireThDemo.java
pi@piWLan97:~ $ java de.weAut.tests.Pi1WireThDemo 28-02119245cd92

Pi1WireThDemo start
de fe 55 05 7f 7e 81 66 60 t=-18125
              measurement  -18.125°C
```

Listing 12: Umkodieren, Übersetzen und Testen; das 1-wire Thermometer ist im Tiefkühlschrank

Automatisieren mit Java auf dem Raspberry PI – Resümee

Die gute Nachricht: Es, das heißt GPIO, watchdog, Linux devices as file, geht auch mit pure Java! Dies wurde mit der Portierung von seit Jahren eingesetzten „100% pure C“-Ansätzen demonstriert. Dass es sich hier statt der „großen“ Anwendungen um kleine Tests und Demos für handelt, ist fast nur noch eine Frage der Skalierung.

So scheinen größere Echtzeitanwendungen, die mit C ja auf Raspberrys im Dauerbetrieb gut laufen, auch mit Java möglich. Allerdings wurden in solchen Fällen alle Störquellen und Zusatzlasten konsequent eliminiert, beginnend mit der grafischen Oberfläche ([weRasp4rem]). Hier bringt Java mit unvermeidbaren Wegwerfobjekten Garbage-Collection und seine Speicherverwaltung als Zusatzlast mit. Ab welchen Lasten, Mengengerüsten und Zykluszeiten das untragbar stört, wird sich zeigen.

Abkürzungen

24/7	Ununterbrochener Dauerbetrieb (24h, 7d) auch über Monate und Jahre
GPIO	General purpose IO. Gemeint sind hier die µP-I/O-Pins die keinerlei Bedeutung für den Raspberry noch für sein Betriebssystem haben, aber dankenswerterweise über einen 40-poligen Pfostenverbinder gut dokumentiert zugänglich gemacht wurden.
GUI	Graphical user interface
IDE	Integrated development environment (wie Eclipse)
I/O	Input and Output, Ein- und Ausgabe im Sinne elektrischer Schnittstellen zur Außenwelt
JNI	Java Native Interface
LED	Light Emitting Diode; Leuchtdiode

Referenzen und Links

[weRaspIoJav] A. Weinert, Raspberry Pi GPIO mit Java — Java statt C Java Magazin 1|2020, pp.78-84 jaxenter.de/java/java-programmiersprache-c-raspberry-pi-90176

[weRasp4rem] A. Weinert, Raspberry for remote services Report, May 2017 a-weinert.de/pub/raspberry4remoteServices.pdf

[weRaspBlog] A. Weinert, Raspberry Pi IO with Java GitHub pages blog post May 2019 https://a-weinert.github.io/raspiGPIOjava.html

[weRaspGit] A. Weinert, Raspberry Pi IO with Java — the project GitHub repository May 2019 https://github.com/a-weinert/weAut

[pigpiodCface] Joan N.N., pigpio library – pigpiod C interface – C API fürs socket interface http://abyz.me.uk/rpi/pigpio/pdif2.html

[pigpiodCsock] Joan N.N., pigpio library – pigpiod socket interface – Spezifikation http://abyz.me.uk/rpi/pigpio/sif.html

[ptty4win] Putty für Windows https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

[nkolbanJp] Neil Kolban, A Java interface to the Raspberry Pi pigpio library GitHub repository https://github.com/nkolban/jpigpio

Prof. Dr.-Ing. Albrecht Weinert ist Gründer und Leiter von weinert-automation.

Bei der Siemens AG entwickelte er hochverfügbare und für sicherheitskritische Anwendungen geeignete Automatisierungssysteme.

Danach war er Professor für Angewandte Informatik an der Hochschule Bochum.

E-Mail: albrecht@a-weinert.de

Aktualisierte und korrigierte Version des urspr. (Sept. 2019) Manuskripts von [weRaspIoJav]
Rev. 12 08.06.2020 Photos A. Weinert