Raspberry Pi GPIO mit Java

Java statt C – Raspberry Pi GPIO mit Java

#Java (https://jaxenter.de/tag/java)

7. Januar 2020 Prof. Dr.-Ing. Albrecht Weinert

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

PiVernetzbare, serienmäßige Standardkleincomputer verdrängen stellenweise schon die klassische Automatisierungstechnik. Unter solchen massenhaft gefertigten Kleinsystemen zählen der Raspberry Pi 3 und seine Brüder schon zu den Großen. Viel häufiger sind daumennagelgroße Mikrocomputersysteme mit integriertem WLAN und einem paar Dutzend I/O-Ports. Wenn solche vernetzten Kleinsteuerungen über VPN oder einen Clouddienst über das Internet erreichbar sind oder gemacht werden können, kann man sie unter dem zugkräftigen Begriff Internet of Things (IoT) propagieren.

Echt jetzt – Automatisieren mit Spielzeug?

Vorteile der Standardsysteme (COTS, Components off the Shelf) gegenüber industriellen Automatisierungssystemen sind viel preiswertere Rechen- und Netzwerkhardware, der Einsatz von Standardbetriebssystemen wie das freie Linux, Standardsprachen und -entwicklungssysteme, darunter freie von teilweise höchster Qualität. Alles ist in die normale IT-Welt integrierbar und mit normalem IT- und Informatikwissen 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 Standardhardware.

Die Stärke von 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 IO, leicht erfassbare, normgerechte LED-Farben (rot = Störung), elektrische Sicherheit usw.

Demgegenüber geht ein RasPi kaputt, wenn ein herausgeführter 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 hinzubauen. 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 er 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), denn damit hat man auch die Linux-Tools 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
- Win32 Disk Imager
- PuTTY

Zu Einzelheiten zum Einrichten siehe: „Raspberry for remote services (http://a-weinert.de/pub/raspberry4remoteServices.pdf)„. Natürlich geht das auch mit einem Linux-PC, nur gibt es dann kein WinSCP (Windows Secure Copy) und Skripting von FTP.

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 Win32 Disk Imager 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/*:

```
13.11.2018    1.866.465.280 2018-11-13-raspbian-stretch-lite.img
05.05.2019                0 ssh
05.05.2019              254 wpa_supplicant.conf
```

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

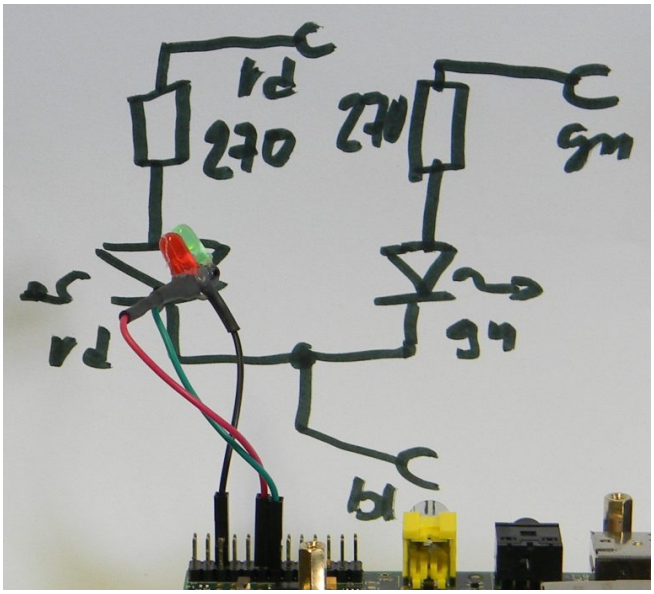
```
# Datei wpa_supplicant.conf in der Boot-Partition (Raspbian Stretch)
country=DE
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
  ssid="weAut2Netz"
  psk=c535a23fbf334cafe95338affe795c4711b9e9f129e62b6cc4094faf89914d51
}
```

Bei *psk* können Sie auch das Klartextpasswort Ihres Netzwerks in Anführungszeichen eintragen. Besser ist natürlich die mit dem Linux-Tool wpa_passphrase verschlüsselte Variante.

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 Defaultnamen *raspberrypi* können Sie sich mit der Terminalsoftware PuTTY (https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html) einloggen und Hostname und Passwort ändern. Auch legen Sie ein Verzeichnis *~/bin/* an. Für den Zugriff auf die 26I/O-Ports des 40-poligen Pfostenverbinders installieren Sie noch pidpio (pdif2 (http://abyz.me.uk/rpi/pigpio/pdif2.html), sif (http://abyz.me.uk/rpi/pigpio/sif.html)):

```
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
```

Den Start des Servers sollten Sie noch mit *sudo crontab -e* und Anfügen einer Zeile *@reboot /usr/local/bin/pigpiod -s 10* automatisieren.



(https://jaxenter.de/wp-content/uploads/2019/12/weinert_raspberry_1.jpg)

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 Pin 6 z. B. anschließen (mit Vorwiderständen wie in **Abb. 1**), können Sie das Demoprogramm */rdGnPiGpioDBlink.c* (https://github.com/a-weinert/weAut) wie in Listing 1 ausprobieren.

Listing 1: „Cross“-Übersetzen, -Bauen und Übertragen in C

```
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
```

Wenn das läuft, 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 im ersten Listing oben erzeugten Git Clone. 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:

```
pi@piWLan97:~ $ rdGnPiGpioDBlink &
                ## maul maul lock file
pi@piWLan97:~ $ touch bin/.lockPiGpio
pi@piWLan97:~ $ rdGnPiGpioDBlink &
[1] 2300
pi@piWLan97:~ $ ps aux | grep rdGnPiGpioDBlink
```

Nun sollte das C-LED-Blinkprogramm 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 (http://abyz.me.uk/rpi/pigpio/pdif2.html) mit dem C-API zum Socket-Interface. Das nutzt der Autor auch für alle anderen Steuerungsanwendungen mit dem Pi. Alle anderen verbreiteten und getesteten I/O-Bibliotheken belasten den 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 der Raspbian. Also müsste man alle Steuerungsprogramme – auch solche in der Testphase – mit *sudo* laufen lassen. Bei manchen Raspbians geht einfaches Lesen und Schreiben einiger I/O-Ports auch so, aber bei besonderen Einstellungen, alternativen Funktionen oder Ähnlichem ist das wieder vorbei.

Die pigpio-Bibliothek 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* automatisiert. Programme, die wie *rdGnPiGpioDBlink* diesen Server nutzen, brauchen kein sudo.

Alles o. k., aber nun doch bitte endlich mit Java

Auf die mit dem Demoprogramm gezeigte Weise geht die I/O mit dem Raspberry – und das auch in viel größerem Maße. Nun wollen wir hierauf aufbauend den entsprechenden Einstieg mit Java zeigen. Bei Schnittstellen in die Außenwelt, Sensoren, Aktoren war Java nie gut, und Sun hatte das nie auf dem Schirm. Ein Beispiel war das plötzliche ersatzlose Streichen des CommApi durch Sun. So verloren einige Java-Serveranwendungen mit RS-485-Schnittstellen zu industriellen Steuerungen die offizielle Grundlage. Das Promoten 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-get install oracle-java8-jdk -y
pi@piWLan97:~ $ java -version
java version "1.8.0_65" (build 1.8.0_65-b17)
Java HotSpot(TM) Client VM (build 25.65-b01, mixed mode)
pi@piWLan97:~ $ which java
/usr/bin/java
```

Mit Java 8_65 haben wir auf dem Pi ein JDK/JRE ohne den „Modulkram“ 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/jdk-8-oracle-arm32-vfp-hflt/* und die Werkzeuge in */usr/lib/jvm/jdk-8-oracle-arm32-vfp-hflt/bin/* und in */usr/lib/jvm/jdk-8-oracle-arm32-vfp-hflt/jre/bin/*. Nun fügen wir *export PATH=/usr/lib/jvm/jdk-8-oracle-arm32-vfp-hflt/bin:$PATH* an das Ende von *.bashrc* an. 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/jdk-8-oracle-arm32-vfp-hflt/jre/lib/ext/
pi@piWLan97:~ java AskAlert
pi@piWLan97:~ java ShowProps
pi@piWLan97:~ java ShowPorts
```

Wenn die drei Java-Tools laufen, weiß man, dass auch komplexere Java-Anwendungen auf dem Pi angemessen arbeiten und dass Installed Extensions funktionieren.

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. Eines davon ist von Neil Kolban (https://github.com/nkolban/jpigpio). 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 brauchen 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 (https://github.com/a-weinert/weAut) 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/j
mv      justLock       ~/bin/
```

Von dort holen Sie sich auch die kleine Hilfsanwendung justLock. Zu deren Hintergrund siehe „Raspberry Pi IO with Java (https://a-weinert.github.io/raspiGPIOjava.html)“ – man kann das inkompatible Verhalten von *java.nio.channels.FileLock* durchaus als einen Fehler des Linux-Ports des JDK werten.

Jetzt holen Sie noch die Quelle *RdGnJPiGpioDBlink.java* und setzen Sie Package-konform ins Verzeichnis *~/de/weAut/tests/*. 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:~ $
```

Unzufriedenheit 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 IO mit pure Java

Eigentlich sind wir fertig. Kolbans Bibliothek liefert den Zugang zur vielfach bewährten Socket-Schnittstelle von pidpiod. Allerdings umfasst sie auch ein 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. 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:

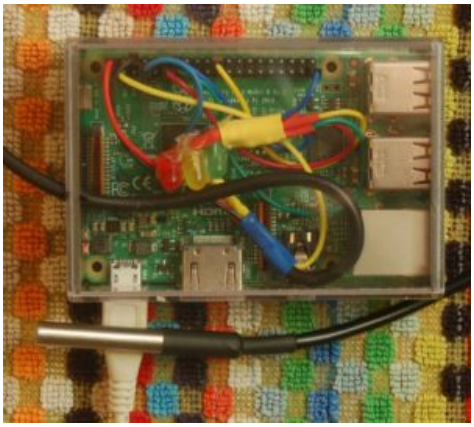
```
pi@piWLan97:~ $ javac de/weAut/tests/RdGnPiGpioDBlink.java
pi@piWLan97:~ $ java de.weAut.tests.RdGnPiGpioDBlink
```

Um auch etwas zu zeigen, das ohne pigpio auf I/O zugreift, gibt es die kleine 1-Draht-Thermometer-Demo de.weAut.tests.Pi1WireThDemo:

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

Die Handhabung von Übertragung, Übersetzen und Start ist wie

bei den anderen Demos. Allerdings müssen Sie ein 1-Wire-Thermometer anschließen, wie in **Abbildung 2** und Listing 3 gezeigt, und dessen ID (28-02119245cd92 ist das abgebildete Teil) herausbekommen. 1 Wire oder allgemeiner Device as File funktioniert also auch mit Java.



(https://jaxenter.de/wp-

Java auf dem Raspberry Pi – ein paar weitere Punkte

Die gute Nachricht: Es geht auch mit „pure Java“. Das wurde mit der Portierung von seit Jahren eingesetzten Pure-C-Ansätzen demonstriert. Dass es sich hier statt der großen Anwendungen um kleine Tests und Demos für 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), WatchDog, Linux Devices as File handelt, ist fast eine Frage der Skalierung oder des Fleißes beim Portieren komplexer Systeme. Betrachten wir im Folgenden noch ein paar Punkte.

content/uploads/2019/12/weinert_raspb

Abb. 2: Aufbau für „xyblink“ und 1-Wire

Die in der Eingangsbetrachtung zu industriellen Automatisierungssystemen vs. Standardkleincomputer und -Netzwerkhardware (off 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 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 vorn. 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_65 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 man diese Forderung so hart und für allgemeine Steuerung heute kaum noch so stellen würde, ist dies ein löbliches Prinzip. Es befreit von den Risiken des Garbage Collectors und der Speicherverwaltung und liegt auch den Beispielen (hier und in „Raspberry Pi IO with Java (https://github.com/a-weinert/weAut)„) zugrunde.

So wurde für periodenfeste Delays eigens eine Klasse *LeTick* eingeführt, die eine absolute *long*-Millisekundenzeit (für *ThreadLocal*) in einem Objekt kapselt, wie in Listing 2 gezeigt. 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.

Listing 2: „LeTick“, im Wesentlichen ein mutable „Long“

```
// static public final ThreadLocal<Long> lastThTick // no Long throw 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
```

In Fällen wie dem in Listing 2 lassen 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 3 zeigt das Auslesen eines 1-Wire-Thermometers, das es auch edelstahlgekapselt (**Abb. 2**) für die üblichen Einsteckröhrchen gibt.

Listing 3: Auslesen eines 1-Wire-Thermometers, gekürzt um Exception Handling und Auswertung

```
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
```

Der zugehörige Linux-Device-Treiber liefert ein Messergebnis in zwei (!) Textzeilen. Nach deren Einlesen schließt sich die Datei automatisch. Deswegen muss das Dateiöffnen in Listing 3 in die den Dauerbetrieb andeutende Endlosschleife. Jede Messung erzeugt also einen *BufferedReader*, einen *FileReader* und indirekt einen *FileInputStream*, einen *FileDescriptor* usw. und schließlich direkt noch zwei String-Objekte. Von diesen mindestens sechs 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 Output Stream – 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 UTF-8. Das wird dann ein Problem, wenn man größere Ausgangsprojekte auf PCs oder Workstations hat, die ISO 8859-1, -15 beziehungsweise die analogen Windows-Codepages wie CP 850 haben. Hier einzelne Dateien oder gar das Ganze für ein kleines Zusatzprojekt zu ändern, erzeugt oft ein Kuddelmuddel mit Auswirkung auf andere. In einer ISO-8859-1-Quelle bewirkt nun allerdings ein Umlaut oder ein Grad-Zeichen (0°C), dass sie auf den Raspberry übertragen dort nicht kompiliert wird, wie nachfolgend zu sehen:

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

Andere javac-Fehler wird man auf dem Raspberry kaum sehen, da sie beim Übersetzen auf dem PC beziehungsweise dort schon im Eclipse sichtbar sind. Hierfür gibt es zwei Workarounds. Das erste ist Kompilieren auf dem PC und Übertragen der *.class*-Dateien auf den Raspberry, wie nachfolgend und in Listing 4 zu sehen ist:

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

Listing 4: FTP auf der Windows-Kommandozeile

```
# 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
```

Siehe da: Die auf Windows mit Java 1.8.0_141 übersetzte ISO-8859-1-Quelle läuft auf dem Linux UTF-8-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 dortiges Kompilieren 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 vonstatten. 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-Kompilierung sei der zweite Workaround geschildert: Übertragen der ISO-8859-1-Quelle auf den Raspberry und dann dort umkodieren, übersetzen und testen:

```
pi@piWLan97:~ $ java de.frame4j.UCopy -toUTF8 de/weAut/tests/Pi1WireThDemo.ja
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
```

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 handelt, ist fast nur noch eine Frage der Skalierung.

So scheinen größere Echtzeitanwendungen, die mit C 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 (http://a-weinert.de/pub/raspberry4remoteServices.pdf). 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.

VERWANDTE THEMEN: Java (https://jaxenter.de/tag/java)

GESCHRIEBEN VON

Prof. Dr.-Ing. Albrecht Weinert

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

Alle Beiträge von Prof. Dr.-Ing. Albrecht Weinert (https://jaxenter.de/author/albrechtweinert)