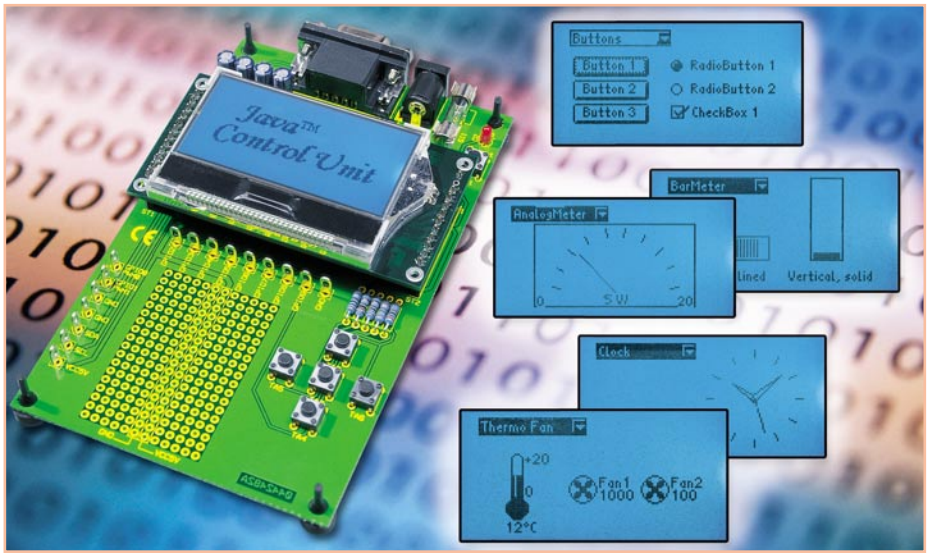
Steuern und visualisieren –



Java™-Control-Unit JCU 10 mit JControl-Technologie Teil 4

Nachdem wir im letzen Teil ausführlich auf die Benutzung der integrierten Entwicklungsumgebung JControl/IDE und das Erstellen eigener Projekte eingegangen sind, möchten wir in dieser Ausgabe anhand von kleinen Beispielprogrammen einige weitere Tipps und Hinweise zum Programmieren eigener Applikationen geben.

Ansteuerung der PWM-Kanäle

In diesem Programm soll dargestellt werden, wie sich die PWM-Kanäle der Java™-Control-Unit ansprechen lassen. In Abbildung 23 ist zunächst die Ausgabe des Displays zu sehen. Angezeigt werden jeweils drei Slider, drei Label und drei BarMeter. Die Label geben jeweils an, welche Funktion der Slider und das BarMeter links und rechts vom Label haben.

Mit dem ersten Slider lässt sich die Frequenz für alle PWM-Kanäle einstellen, mit dem zweiten und dritten Slider ist das Tastverhältnis (duty cycle) der Kanäle PWM0 und PWM1 wählbar. Die BarMeter auf der rechten Seite zeigen den jeweils eingestellten Wert an.

Programmerarbeitung

Um dieses Programm auf der Java™-Control-Unit zu testen, ist mit der JControl/IDE ein neues Projekt zu erstellen, und der Quelltext entsprechend Abbildung 24 einzugeben.

Das Programm beginnt wie gewohnt mit den *„Import“*-Anweisungen, die die Bereitstellung der im Programm benötigten Klassen vornehmen. Danach erfolgt das Anlegen der Hauptklasse PWM_DEMO (diese erbt Eigenschaften der Klasse *„Frame“*), und das Interface *„ActionListener“* wird implementiert. Auf die Funktionen der Klasse Frame und der ActionListener gehen wir an dieser Stelle nicht weiter ein, sie wurden schon ausführlich in der letzten Ausgabe beschrieben.

Nun legt man jeweils drei Variablen der Typen Slider, Label und BarMeter an, die für die angezeigten Komponenten benötigt werden. Danach beginnt der Konstruktorteil (public PWM_DEMO()). Den Slidern, Labels und BarMetern wird jeweils ein neues Objekt zugewiesen, hierbei legt man deren Eigenschaften wie Position, Größe, Ausrichtung usw. fest.

Bei den Slidern und BarMetern werden hier auch der Start- und Endwert festgelegt und bei den Slidern zusätzlich die Schrittweite, mit der sie sich beim Betätigen bewegen. Der Frequenz-Slider lässt sich im Bereich von 1000 bis 10000 in 500er-Schritten einstellen, dies entspricht einer einstellbaren Frequenz von 1 kHz bis 10 kHz. Die beiden Duty-Slider lassen sich im Bereich von -256 bis +256 einstellen. Diese Werte werden dabei von der Java™-Control-Unit so umgesetzt, dass sich bei der eingestellten Frequenz das Tastverhältnis entsprechend Abbildung 25 einstellen lässt. Das positive bzw. negative Vorzeichen bewirkt, das zu einem Zeitpunkt t_0 die Periode mit High- oder Low-Pegel begonnen wird.

Mit der Methode *„setNumericDisplay“* weist man den BarMetern zu, dass der angezeigte Wert zusätzlich als Zahl unter dem BarMeter erscheint. Weiter wird mit der Methode *„setActionListener“* für die drei Slider registriert, dass beim Verstellen der Slider die Methode *„onActionEvent“* ausgeführt wird.

Danach erfolgt eine Initialisierung der Slider, BarMeter und PWM-Kanäle, damit beim Einschalten die an den Slidern eingestellten Werte mit den angezeigten Werten und den ausgegebenen PWM-Signalen übereinstimmen. Die Frequenz wird auf 1000 gestellt, dies entspricht der Startposition des „Frequenz“-Sliders.

Mit der Methode *„setDuty“* erfolgt ein Setzen der Tastverhältnisse für die beiden PWM-Kanäle auf „0“ (duty0 und duty1 sind bei der Deklaration mit „0“ initialisiert). Dies hat zur Folge, dass gemäß Abbildung 25 Low-Pegel ausgeben wird. Die Einstellung der beiden Slider und BarMeter erfolgt ebenfalls auf den Wert „0“ – dies entspricht deren Mittelstellung.

Im nächsten Abschnitt erfolgt mit der Methode *„setActive“* die Aktivierung der

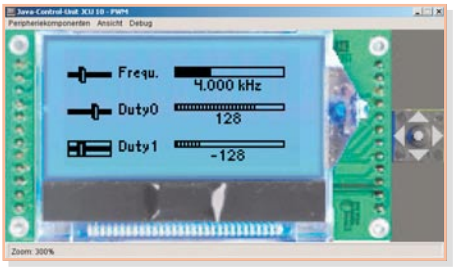


Bild 23: Ansicht der Display-Ausgabe des PWM-Demoprogramms

```
/** PWM-Demo Dateiname PWM_DEMO.java */

import jcontrol.ui.vole.Frame;
import jcontrol.io.PWM;
import jcontrol.ui.vole.Label;
import jcontrol.ui.vole.Slider;
import jcontrol.ui.vole.meter.BarMeter;
import jcontrol.ui.vole.event.ActionListener;
import jcontrol.ui.vole.event.ActionEvent;

public class PWM_DEMO extends Frame implements ActionListener
{  int duty0 = 0;                                                    // Tastverhältnis für beide Kanäle
   int duty1 = 0;                                                    // initialisieren 0 -> kein Ausgangssignal
   Slider      SlideFrequenz,SlideDuty0,SlideDuty1;                  // Variablen vom jeweiligen Typ anlegen
   Label       LFrequenz,LDuty0,LDuty1;
   BarMeter    BarFrequenz,BarDuty0,BarDuty1;
   public PWM_DEMO()                                                 // Konstruktor
   {  // Slider zum Einstellen der Frequenz und der Tastverhältnisse für PWM0 und PWM1
      SlideFrequenz  = new Slider(2,10,24,1000,10000,500);           // von 1000 bis 10000 Hz Step 500 Hz
      SlideDuty0     = new Slider(2,30,24,-256,+256,16);             // von -256 bis 256  Step 16
      SlideDuty1     = new Slider(2,50,24,-255,+256,16);             // von -256 bis 256  Step 16

      // Label als Beschriftung
      LFrequenz      = new Label("Frequenz", 30, 10, 32, 10, Label.ALIGN_LEFT);
      LDuty0         = new Label("Duty0" , 30, 30, 25, 10, Label.ALIGN_LEFT);
      LDuty1         = new Label("Duty1" , 30, 50, 25, 10, Label.ALIGN_LEFT);

      // BarMeter für Frequenz und Tastverhältnisse zur Darstellung der eingestellten Werte
      BarFrequenz    = new BarMeter(65,10,60,14,1000,10000,
                                    BarMeter.ORIENTATION_HORIZONTAL,BarMeter.FILL_SOLID);
      BarDuty0       = new BarMeter(65,30,60,12,-256,256,
                                    BarMeter.ORIENTATION_HORIZONTAL,BarMeter.FILL_LINE);
      BarDuty1       = new BarMeter(65,50,60,12,-256,+256,
                                    BarMeter.ORIENTATION_HORIZONTAL,BarMeter.FILL_LINE);

      // Die eingestellten Werte unter den BarMetern darstellen
      BarFrequenz.setNumericDisplay(5, 3, " kHz");                 // 5 Ziffern, 3 Nachkommastellen, Einheit kHz
      BarDuty0.setNumericDisplay(3, 0, "");                        // 3 Ziffern, 0 Nachkommastellen, keine Einheit
      BarDuty1.setNumericDisplay(3, 0, "");

      SlideFrequenz.setActionListener(this);                       // ActionListerner für die Slider hinzufügen
      SlideDuty0.setActionListener(this);
      SlideDuty1.setActionListener(this);

      PWM.setFrequency(1000);                                      // Startwert für Frequenz (alle PWM-Kanäle) auf 1 kHz
      PWM.setDuty(0, duty0);                                       // Tastverhältnisse auf Startwert (0) setzen
      PWM.setDuty(1, duty1);
      BarDuty0.setValue(duty0);                                    // BarMeter für Tastverhältnis auf Startwert setzen
      BarDuty1.setValue(duty1);
      SlideDuty0.setValue(duty0);                                  // Slider für Tastverhältnis auf Startwert setzen
      SlideDuty1.setValue(duty1);
      PWM.setActive(0, true);                                      // PWM-Signal für Kanal 0 und 1 aktivieren
      PWM.setActive(1, true);
      this.add(SlideFrequenz);                                     // Label, Buttons, Slider und
      this.add(SlideDuty0);                                        // BarMeter zum Frame hinzufügen
      this.add(SlideDuty1);
      this.add(LFrequenz);
      this.add(LDuty0);
      this.add(LDuty1);
      this.add(BarFrequenz);
      this.add(BarDuty0);
      this.add(BarDuty1);
      this.show();                                                 // den Frame und seinen Inhalt anzeigen
   }

   public void onActionEvent(ActionEvent event)
   {  // Identifiziere die Slider anhand der Ergeignisquelle
      if(event.getSource() == SlideFrequenz)                       // Frequenz einstellen
      {  PWM.setFrequency(SlideFrequenz.getValue());
         PWM.setDuty(0,duty0);                                     // nach dem Ändern der Frequenz müssen die
         PWM.setDuty(1,duty1);                                     // Tastverhältnisse neu gesetzt werden
         BarFrequenz.setValue(SlideFrequenz.getValue());
      }
      else if(event.getSource() == SlideDuty0)                     // Tastverhältnis PWM0 einstellen
      {  PWM.setDuty(0,SlideDuty0.getValue());
         BarDuty0.setValue(duty0=SlideDuty0.getValue());
      }
      else if(event.getSource() == SlideDuty1)                     // Tastverhältnis PWM1 einstellen
      {  PWM.setDuty(1,SlideDuty1.getValue());
         BarDuty1.setValue(duty1=SlideDuty1.getValue());
      }
   }

   public static void main(String[] args)
   {  new PWM_DEMO();                                               // Eine neue Instanz der Klasse PWM_DEMO anlegen
   }
}
```

Bild 24: Quelltext des PWM-Demoprogramms

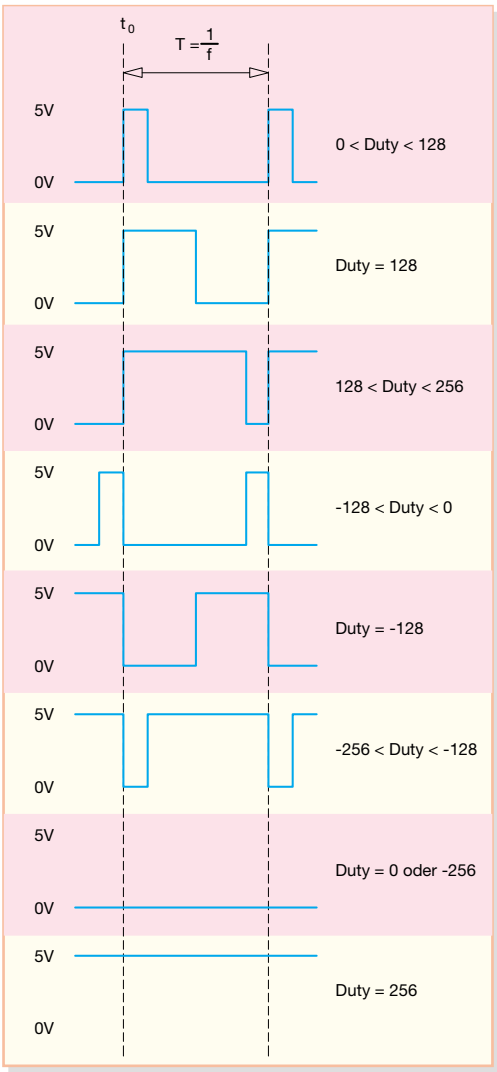


Bild 25: Signalverläufe am PWM-Ausgang der Java™-Control-Unit

beiden PWM Kanäle – bislang hatten sie die Funktion eines IO-Pins (GPIO0/1).

Mit der Methode *„add“* fügen wir die neu angelegten Objekte zu unserer Klasse PWM_DEMO hinzu, und mit der Methode *„show“* wird der Frame mit den enthaltenen Objekten sichtbar gemacht.

Als Nächstes wird die Methode *„onActionEvent“* erstellt, sie kommt zur Anwendung, wenn man die Position von einem der drei Slider verändert. Mit den „If“-Abfragen wird ermittelt, welcher Slider das ActionEvent ausgelöst hat.

Entsprechend erfolgt beim „Frequenz“-Slider mit der Methode *„setFrequency“* die Änderung der Frequenz, abhängig von der jetzt aktuellen Slider-Stellung. Danach müssen die Tastverhältnisse der PWM-Kanäle neu gesetzt werden, dies geschieht mit der Methode *„setDuty“*, wobei die Variablen *„duty0“* und *„duty1“* die aktuellen Positionen der „Duty“-Slider enthalten. Die Methode *„setValue“* aktualisiert schließlich das BarMeter.

Beim Verändern eines der „Duty“- Slider wird jetzt mit der Methode *„setDuty“* das Tastverhältnis des entsprechenden PWM-Kanals geändert und anschließend das zugehörige BarMeter aktualisiert.

Am Schluss des Quelltextes findet sich die Main-Methode, die beim Programmstart ausgeführt wird, und dabei eine neue Instanz der „PWM_Demo“-Klasse anlegt.

Nach dem Kompilieren wird das Projekt in den Speicher der Java™-Control-Unit geladen und ist dort ausführbar.

AD-Wandlung

In unserem zweiten Beispiel wollen wir zeigen, wie einfach es ist, mit dem AD-Wandler der Java™-Control-Unit eine Spannungsmessung durchzuführen und die gemessene Spannung mit einem AnalogMeter ansprechend auf dem Display darzustellen. Für die AD-Wandlung stehen, wir erinnern uns, die IO-Pins GPIO2 bis 9 zur Verfügung. Wir nutzen GPIO2/ADC0 zur Spannungsmessung. Das mit dem folgend beschriebenen Programm zu erreichende Anzeigeergebnis nimmt Abbildung 26 schon einmal vorweg.

Programmaufbau

In Abbildung 27 ist der nur aus wenigen Zeilen bestehende Quelltext zu sehen.

Hier erkennt man schnell schon bekannte Programmierschritte, die wir weiter unten noch näher ansehen:

- Importieren der benötigten Klassen
- Variable vom Typ AnalogMeter anlegen
- Beginn des Konstruktors (public ADC_DEMO())
- Zuweisung des „FLOATING“-Status zum verwendeten GPIO-Pin, dadurch arbeitet der Pin als Eingang ohne Pull-up-Widerstand.
- schließlich der Variable „Anzeige“ ein neues Objekt vom Typ „AnalogMeter“ zuweisen, hierbei werden Position, Größe, Start- und Endwert der Skala, Öffnungswinkel, Anordnung von Zeiger und Skala sowie die Anzahl der Skalenmarkierungen zugewiesen.

Konventionen

Da die Referenzspannung des AD-Wandlers beim Einsatz der Java™-Control-Unit im Evaluationboard 5 V beträgt, haben wir den Endwert der Anzeige bei 50 festgelegt, dies ermöglicht die Anzeige von Werten bis 5 V mit einer Nachkommastelle. Der AD-Wandler kann in eigenen Schaltungen prinzipiell auch mit einer Spannung, die geringer ist als 5 V, betrieben werden. Wenn jedoch gleichzeitig der

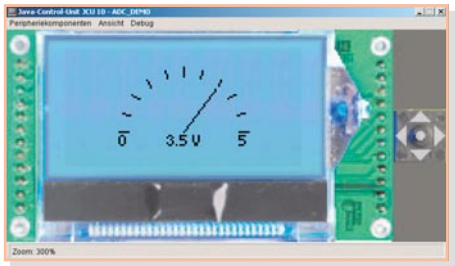


Bild 26: Ansicht der Display-Ausgabe des AD-Wandler-Demoprogramms

in Teil 2 („ELVjournal“ 5/2004) besprochene Tastaturdecoder zum Einsatz kommen soll, ist allerdings zu beachten, dass der Spannungsteiler aus R 5 bis R 9 (R14) nicht an 5 V, sondern ebenfalls an die geringere Referenzspannung anzuschließen ist – oder die Widerstandswerte sind entsprechend zu ändern.

Die verwendeten Methoden

Wir wollen die hier eingesetzten Methoden einmal etwas näher betrachten.

Mit der Methode *„setNumericDisplay“* legt man fest, dass der Anzeigewert auch als Zahl mit zwei Ziffern, von denen eine als Nachkommastelle zu werten ist, mit der Einheit „V“ (Volt) unter dem AnalogMeter angezeigt wird.

Die Methode *„setCaption“* zeigt die Werte „0“ und „5“ als Start- bzw. Endwert der Skala an.

Das AnalogMeter wird mit der Methode *„add“* zu unserer Klasse ADC_DEMO hinzugefügt, und mit der Methode *„show“* macht man schließlich den Frame mit dem AnalogMeter sichtbar.

In der Main-Methode läuft hier eine Endlosschleife, in der die AD-Wandlung durchgeführt wird.

Die Methode *„getValue“* liefert einen Wert zwischen 0 und 255, je nach Spannung, die am entsprechenden AD-Wandler-Kanal (in unserem Beispiel ADC0/GPIO2) anliegt. Dieser Wert wird auf einen Bereich von 0 bis 50 umgerechnet, mit der Methode *„setValue“* an das AnalogMeter übergeben und unter dem AnalogMeter angezeigt. Dazu schlägt der Zeiger entsprechend aus.

Als Letztes finden wir in der Main-Methode die *„Sleep“*-Anweisung. Diese Anweisung führt dazu, dass die laufende Anwendung für die Zeit, die als Parameter angegeben ist, „schläft“, in unserem Beispiel 100 Millisekunden. Dies bewirkt also eine Pause von 100 ms zwischen den einzelnen AD-Wandlungen, sprich Messungen.

Neu ist an dieser Stelle auch die so genannte „Try-Catch“-Anweisung, die den Aufruf der Methode *„sleep“* umgibt. Hierbei handelt es sich um das Abfangen einer Exception (Ausnahme, Fehler) die während der Laufzeit des Programms auftreten kann.

```
/**   ADC-DEMO Dateiname ADC_DEMO.java */

import jcontrol.io.Backlight;
import jcontrol.io.ADC;
import jcontrol.ui.vole.Frame;
import jcontrol.ui.vole.meter.AnalogMeter;
import jcontrol.lang.ThreadExt;
import jcontrol.io.GPIO;

public class ADC_DEMO extends Frame
{  AnalogMeter Anzeige;                                  // Variable vom Typ AnalogMeter anlegen

   public ADC_DEMO()                                     // Konstruktor
   {  Backlight.setBrightness(Backlight.MAX_BRIGHTNESS);
      GPIO.setMode(2,GPIO.FLOATING);                     // Modus von GPIO2 für AD-Wandlung auf Floating
                                                         // Anzeige ein neues Objekt vom Typ AnalogMeter zuweisen
      Anzeige = new AnalogMeter(10,10,108,44,0,50,180,AnalogMeter.ORIENTATION_CENTER,11);
      Anzeige.setNumericDisplay(2,1," V");               // Ausgabe des Anzeigewertes unter dem AnalogMeter
                                                         // mit zwei Ziffern, eine Nachkommastelle
      Anzeige.setCaption("0","5");                       // Start- und Endwert des AnalogMeters anzeigen

      this.add(Anzeige);                                 // das AnalogMeter zum Frame hinzufügen
      this.show();                                       // den Frame mit Inhalt anzeigen
   }

   public static void main(String[] args)
   {  ADC_DEMO My_ADC1 = new ADC_DEMO();                 // My_ADC1 wird eine neue Instanz
                                                         // der Klasse ADC_DEMO zugewiesen
      while(true)
      {  My_ADC1.Anzeige.setValue(ADC.getValue(0)*50/255); // Berechnung des Anzeigewertes,
                                                         // die AD-Wandlung liefert Werte von 0-255
                                                         // 5V entspricht 50; eine Nachkommastelle // -> 5,0V
         try
         {
            ThreadExt.sleep(100);                        // 100 ms Pause zwischen den Messungen
         }  catch (InterruptedException e) {}
      }
   }
}
```

Bild 27: Quelltext des AD-Wandler-Demoprogramms

Auf diese Art der Fehlerbehandlung gehen wir im nächsten Abschnitt genauer ein.

Exceptions

Die Programmiersprache Java™ verfügt über einen Mechanismus, der es dem Programmierer ermöglicht, Fehler, die während des Programmablaufes auftreten können, strukturiert zu behandeln. Hierzu werden von Methoden, bei denen es im Programmablauf zu Fehlern kommen kann, so genannte Exceptions ausgelöst. Hierbei handelt es sich um eine Fehlermeldung eines bestimmten Ausnahmetyps. Dem Java™-Compiler ist bekannt, welche Methoden welche Typen von Exceptions auslösen können. Benutzt der Programmierer nun eine solche Methode, erwartet der Compiler eine Behandlung dieser Exception, die beim Ausführen der Methode auftreten kann. Löscht man in unserem Beispielprogramm „ADC_DEMO“ die „Try-Catch“-Anweisung, beanstandet der Compiler eine nicht abgefangene Exception.

Die „Try-Catch“-Anweisung übernimmt das Abfangen von Fehlermeldungen und kann den Programmierer in gewünschter Weise auf einen Fehler reagieren lassen, denn nicht immer ist das Auslösen von Exceptions ein Grund, das Programm zu beenden.

In den geschweiften Klammern nach der *„Try“-Anweisung* befinden sich eine oder mehrere Anweisungen, die eine Exception auslösen können. Tritt während des laufenden Programms nun ein Fehler auf, wird die Abarbeitung an dieser Stelle unterbrochen, und es wird nach einer „Catch“-Anweisung, die als Parameter eine Variable des richtigen Ausnahmetyps hat, gesucht. Es können also auch nach dem *„Try“*-Block durchaus mehrere *„Catch“*-Anweisungen folgen, die verschiedene Fehler auswerten und entsprechende Reaktionen einleiten. Findet das Programm eine *„Catch“*-Anweisung mit dem richtigen Ausnahmetyp, wird der als Parameter angegebenen Variable („e“ vom Typ *„InterruptedException“* in unserem ADC_DEMO) ein Fehlerobjekt zugewiesen und danach der Programmteil ausgeführt, der in den geschweiften Klammern nach der *„Catch“*-Anweisung folgt. In unserem Beispiel kann die Exception nur ausgelöst werden, wenn die Methode *„sleep“* durch einen anderen Thread unterbrochen wird. Da dieser Fall nicht eintreten kann, wird die Exception nicht weiter behandelt. Aus diesem Grund ist in den geschweiften Klammern kein Befehl enthalten.

So weit zu unseren ausführlich besprochenen und dokumentierten Programmbeispielen, die den Einstieg in eigene Applikationen sicher erleichtern, haben wir doch insbesondere oft genutzte Methoden vorgestellt.

Weitere gut kommentierte Beispielprogramme werden bei der Installation der JControl/IDE im Verzeichnis *..\JControl\Demo* installiert. Hier finden sich Programme, die sämtliche Funktionen der Java™-Control-Unit zeigen.

Abschließend zu diesem Thema noch der Hinweis auf zwei besonders interessante Internetadressen.

Unter [1] finden Sie noch mehr Informationen zum Thema JControl, neben ausführlichen Tutorials steht hier auch immer die neuste Version der JControl/IDE zum Download bereit.

Auf [2] finden Sie das Handbuch der Java-Programmierung, ein sehr ausführliches Nachschlagewerk in Deutsch, das einen guten Einstieg in die Programmiersprache und das objektorientierte Programmieren ermöglicht. Im Download-Bereich dieser Seite steht eine kostenlose HTML-Version des Buches für den privaten Gebrauch zur Verfügung.

In der nächsten Ausgabe stellen wir ein IO-Interface vor, mit dem die Java™-Control-Unit als komplettes Gerät inkl. Gehäuse universell zur Messung und Steuerung eingesetzt werden kann. ELV

[1] http://www.jcontrol.org
[2] http://www.javabuch.de