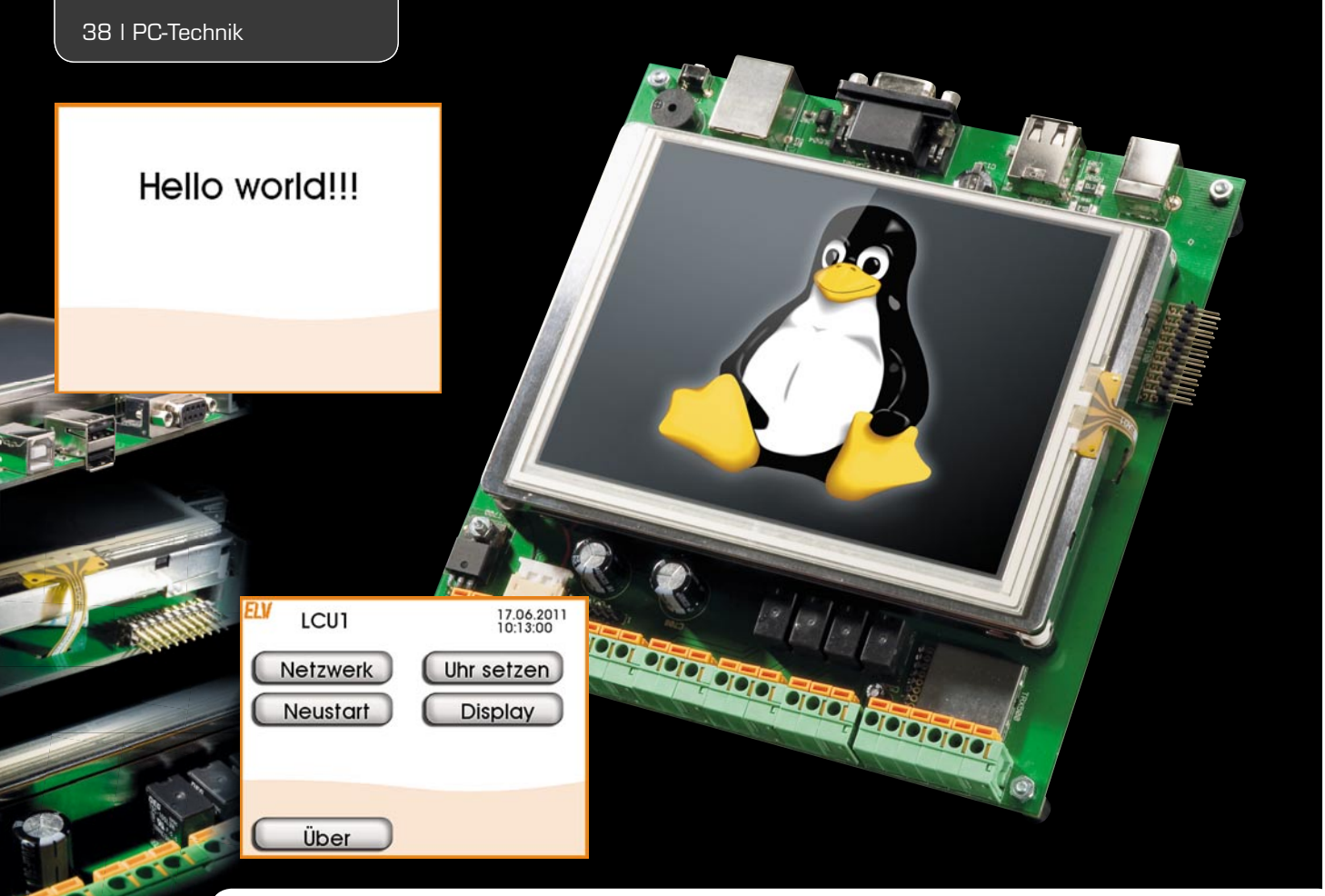

LINUX-Control-Unit LCU 1

Die Logiksteuerung

Nach der Vorstellung der UI-Engine im ELVjournal 5/2011 geht es nun daran, eine Verbindung zwischen der selbst erstellten Bedienoberfläche und den Ein- und Ausgängen der LCU 1 herzustellen. Dabei werden hier auch die Grundlagen gelegt, um im nächsten Teil der Artikelserie die Komponenten des HomeMatic-wired-Systems über den RS485-Bus ansprechen zu können.

Der Zugriff auf die Peripherie

Die in diesem Artikel vorgestelle Software besteht aus vier Teilen:

- **Plattformprozess pfmd**
  Dieses Programm stellt den Zugriff auf die Hardware-Ein- und -Ausgänge der LCU über das bereits vom HomeMatic-System bekannte XML-RPC-Protokoll bereit [1].

- **Logikprogramm**
  Das Logikprogramm implementiert die Logiksteuerung. Es wertet Logiksignale (siehe „Logik-Switch“) aus und schaltet in Abhängigkeit davon andere Logiksignale. Dabei können auch zeitliche Abläufe implementiert werden.

- **Logik-Switch**
  Der Logik-Switch verwaltet die Logiksignale. Jedes Logiksignal hat einen eindeutigen Namen. Ein Logiksignal kann den Zustand eines über XML-RPC angebundenen Ein- oder Ausgangs repräsentieren oder einfach ein internes Signal darstellen. Logiksignale können von der Oberfläche (UI-Engine) oder von der Logiksteuerung manipuliert werden.

- **Erweiterung der UI-Engine für Zugriff auf Logiksignale**
  Die UI-Engine wurde um Elemente in der Seitenbeschreibung und TCL-Befehle erweitert, um auf die Logiksignale zugreifen zu können.

Literaturverzeichnis

[1] Via Netzwerk auf HomeMatic zugreifen – XML-RPC-Schnittstelle/HomeMatic, ELVjournal 4/2010, S. 6 ff.
[2] Linux Control Unit – Oberflächen-Engine, ELVjournal 5/2011, S. 76 ff.

Der Plattformprozess pfmd

Der Plattformprozess exportiert die 4 Relaisausgänge, die 4 Digitaleingänge, die beiden Tastereingänge, den akustischen Signalgeber und die beiden ADC-Eingänge per XML-RPC (siehe auch [1]). Die XML-RPC-Schnittstelle wird über TCP-Port 2002 angesprochen. Über die Schnittstelle erfolgt der Export eines logischen Gerätes mit dem Namen „SYSTEM“.

Dieses Gerät hat 13 logische Kanäle. Jeder stellt einen Ein- oder Ausgang der LCU dar:

- Die Kanäle 1 bis 4 entsprechen den 4 Digitaleingängen. Jeder Kanal hat einen Wert vom Typ Boolean mit dem Namen „STATE“.
- Die Kanäle 5 und 6 entsprechen den beiden seitlichen Tastern. Sie verhalten sich genau wie die Digitaleingänge.
- Die Kanäle 7 bis 10 entsprechen den Relaisausgängen. Jeder Kanal hat einen Wert vom Typ Boolean mit dem Namen „STATE“.
- Der Kanal 11 entspricht dem akustischen Signalgeber. Er verhält sich genau wie die Relaisausgänge.
- Die Kanäle 12 und 13 entsprechen den ADC-Eingängen. Jeder Kanal hat einen Wert vom Typ Integer mit dem Namen „VALUE“. Der Wert entspricht jeweils der aktuellen ADC-Eingangsspannung in mV.

Der Quellcode zum pfmd befindet sich im Verzeichnis *pfmd*. Der Plattformprozess wird über die Datei */etc/init.d/S90pfmd* aus dem Quellcodeverzeichnis *scripts* beim Booten automatisch gestartet.

Der Quellcode muss vom Anwender nicht geändert werden.

Logikprogramm

Das Logikprogramm ist in der Programmiersprache immediateC geschrieben. immediateC ist eine Erweiterung der Sprache C. Sie verwendet die gleiche Syntax wie C. Zusätzlich zum semantischen Umfang der Sprache C können mit immediateC gegenläufige (parallele) Abläufe mittels Modellierung des Datenflusses beschrieben werden.

Um dies zu erreichen, verwendet immediateC das Konzept der „immediate“ (= unmittelbar) Variablen. Eine Änderung an einer immediate-Variablen wirkt sich sofort auf alle Ausdrücke aus, die von der Variablen abhängen. Die Änderung eines Eingangs pflanzt sich dadurch unmittelbar bis zum Ausgang fort. Möchte man das mit einer kontrollflussgesteuerten Sprache wie C oder C++ implementieren, ist normalerweise eine umfangreiche Ereignisverarbeitung nötig.

Die Programmierung in immediateC ähnelt der Modellierung von Hardware mit einer Sprache wie VHDL. Für die Benennung von Ein- und Ausgängen wird die in der Norm für Speicherprogrammierbare Steuerungen (SPS) IEC-1131 bzw. EN-61131 festgelegte Notation verwendet. Demnach beginnen Eingänge mit „I“ und Ausgänge mit „Q“. Der nächste Buchstabe gibt an, ob es sich um ein Bit („X“), ein Byte („B“), ein 16-Bit-Wort („W“) oder ein 32-Bit-Wort („L“) handelt. Danach folgt eine Nummer und bei einem Bit noch durch „.“ getrennt eine Bitnummer.

`IX1.2` ist somit ein Bit-Eingang,
`QB6` ist ein 8-Bit-Ausgang, und
`IL11` ist ein 32-Bit-Eingang.

Die Dokumentation zu immediateC befindet sich im Quellcodeverzeichnis *icc/doc*.

immediateC besteht aus einem Präprozessor, der aus immediateC-Quellcode „normalen“ C-Code erzeugt. Dieser C-Code wird dann kompiliert und gegen eine Bibliothek gelinkt, die ebenfalls Bestandteil von immediateC ist. Darüber hinaus gehört zu immediateC eine Sammlung von Tools, die in der Scriptsprache Perl geschrieben sind. Das wichtigste dieser Tools ist das Programm „iCserver“. Es dient als zentrale Instanz für die Kommunikation zwischen mehreren immediateC-Programmen, Visualisierungsapplikationen und Hardwaretreibern. Diese Kommunikation erfolgt über ein von immediateC definiertes TCP-Protokoll (Standardport 8778).

Für das hier vorgestellte Softwarepaket wurde iCserver in C++ neu implementiert. Es kommt also das Programm iCserver von immediateC nicht zum Einsatz. Die neu implementierte Version von iCserver bezeichnen wir als Logik-Switch. Es wird im folgenden Abschnitt näher beschrieben.

Der Quellcode zu immediateC befindet sich im Verzeichnis *icc*. Während des Build-Vorgangs wird immediateC zweimal kompiliert, einmal für den Build-Rechner und einmal für die LCU. Das ist nötig, weil die immediateC-Build-Umgebung (Präprozessor etc.) auf dem Build-Rechner benötigt wird, die Laufzeitumgebung aber auf der LCU.

Das eigentliche Logikprogramm, das auf immediateC aufsetzt, befindet sich im Quellcodeverzeichnis *iclogic*. In der fertigen Firmware liegt es dann in */bin/iclogic*. Es wird beim Booten gestartet über das Startscript:

```
/etc/init.d/S92iclogic
```

Das Logikprogramm ist eine Beispielapplikation, die zeigt, wie man zeitliche Abläufe implementiert und wie man Ausgänge in Abhängigkeit von Eingängen schaltet. Die Beispielapplikation ist weiter unten beschrieben.

Logik-Switch

Der Logik-Switch ersetzt das Programm *iCserver* (siehe immediateC). Der Quellcode zum Logik-Switch befindet sich im Verzeichnis *icserver*. In der Firmware liegt der Logik-Switch unter */bin/icserver*. Er wird beim Booten über das Startscript */etc/init.d/S91icserver* gestartet. Die Dokumentation zu iCserver befindet sich im Quellcodeverzeichnis *icserver/doc*.

icserver liest beim Starten die Konfigurationsdateien */etc/icserver/icserver_system.conf* und */etc/config/icserver_user.conf* ein. Für diesen Artikel interessiert uns nur die erste dieser Dateien, die andere ist Gegenstand des nächsten Teils.

In der Konfigurationsdatei */etc/icserver/icserver_system.conf* (im Quellcode unter *scripts/icserver_system.conf*) werden die Logiksignale konfiguriert. Jedes Logiksignal bekommt dabei einen Namen gemäß IEC-1131 zugewiesen, damit das Logikprogramm mit dem Signal etwas anfangen kann. Optional kann jedem Signal noch ein zweiter, besser lesbarer Name als Alias zugewiesen werden. Über dieses Alias kann das Signal dann in der Programmierung der Oberfläche verwendet werden.

Über das Script *iclogic/xml2iha.tcl* wird beim Build des Logikprogramms die Datei *external_signals.iha* erzeugt, welche die Aliasnamen der Signale auf die IEC-1131-Namen der Signale abbildet. Durch diesen Trick kann im Logikprogramm ebenfalls mit den lesbaren Namen programmiert werden.

Jedem Logiksignal kann dann noch ein Wert eines über XML-RPC angebundenen Gerätes zugeordnet werden. Eine Änderung an einem Logikausgang führt dann sofort zum Schalten des zugeordneten physikalischen Ausgangs. Eine Änderung an einem physikalischen Eingang wirkt sich entsprechend sofort auf das Logiksignal aus.

Optional kann angegeben werden, dass ein Logiksignal persistiert werden soll. Der Wert wird dann bei jeder Änderung in eine Datei (*/etc/config/icserver_values*) geschrieben und steht dadurch nach einem Reset immer noch zur Verfügung. Darüber können z. B. Konfigurationseinstellungen persistiert werden.

Zugriff auf Logiksignale über die UI-Engine

Für die Arbeit mit Logiksignalen wurde die UI-Engine um den TCL-Befehl *control* erweitert. Mit diesem Befehl können Logiksignale gesetzt und abgefragt werden.

Zusätzlich wurde ein neues XML-Tag <*trigger*> eingeführt, mit dem sich eine TCL-Methode angeben lässt, die bei der Änderung eines Logiksignals automatisch aufgerufen wird. Die Dokumentation dazu befindet sich in [2].

Befehle zur Manipulation von Logiksignalen

Über den neuen Befehl *control* lassen sich die Werte von Logiksignalen abfragen und setzen. Außerdem kann asynchron auf Änderungen von Logiksignalen durch Aufruf einer Callback-Methode reagiert werden. Der Befehl

```
control set output1 1
```

setzt den Wert des Logiksignals *output1* auf 1. Statt *output1* darf auch der dem Signal entsprechende IEC-1131-Name angegeben werden, z. B. *QX2.1*.

Den Wert eines Logiksignals abfragen kann man z. B. mit:

```
control get input1
```

Auch hier darf der IEC-1131-Name verwendet werden, z. B. *IX3.1*.
Einen Trigger für ein Logiksignal setzt man so:

```
control trigger input1 OnInputChanged
```

Dieser Befehl sorgt dafür, dass bei einer Änderung des Wertes von *input1* die TCL-Methode *OnInputChanged* aufgerufen wird. Diese muss wie folgt definiert sein:

```
proc OnInputChanged { valueId value } {
}
```

Der Parameter *valueId* enthält beim Aufruf den Namen des geänderten Wertes und der Parameter *value* den neuen Wert.

<trigger> als Element der XML-Seitenbeschreibung

Das neue XML-Element <trigger> erfüllt die gleiche Funktion wie der TCL-Befehl *control trigger*. Ein Beispiel dazu ist in Bild 1 zu sehen. Hier wird ein Trigger für das Signal *intInOnboardADC_0* registriert. Bei jeder Änderung des Signals wird die TCL-Methode *OnAdcChanged()* aufgerufen.

Der Trigger wird nur ausgelöst, solange die entsprechende Seite angezeigt wird. Soll ein Trigger unabhängig von der angezeigten UI-Seite aufgerufen werden, so kann das über das XML-Attribut *global="true"* erreicht werden. So kann z. B. bei der Änderung eines wichtigen Logiksignals auch gleich die entsprechende Oberflächenseite angezeigt werden.

Beispielapplikation

Die Beispielapplikation besteht aus Oberflächenseiten im Quellcodeverzeichnis *uidescription*. Die für diesen Artikel relevanten Seiten fangen alle mit *„io"* an.

Dazu gehören das Logikprogramm im Quellcodeverzeichnis *iclogic* und die Konfigurationsdatei für den Logikswitch unter *scripts/icserver_system.conf*.

Auf der Übersichtsseite wurde eine Schaltfläche „UI" hinzugefügt, über die sich das UI der Beispielapplikation, wie in Bild 2 zu sehen, aufrufen lässt.

Die Applikation besteht aus zwei Teilen. Der Teil „Signalübersicht" zeigt, wie per Oberflächenprogrammierung auf die Ein- und Ausgänge zugegriffen wird. An diesem Teil der Applikation ist noch keine Logiksteuerung mit immediateC beteiligt.

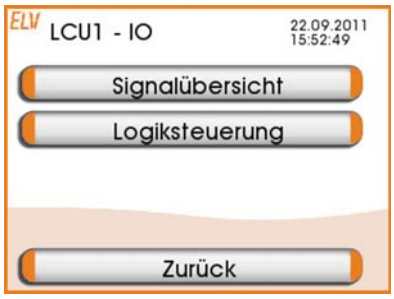

Bild 2: Die Oberflächenseite der Beispielapplikation

Signalübersicht

Auf der Seite *Signalübersicht* werden die Zustände aller Ein- und Ausgänge angezeigt (Bild 3). Änderungen an den Eingängen wirken sich sofort auf die Oberfläche aus. Die Ausgänge und der Signalgeber lassen sich über den Touchscreen schalten.

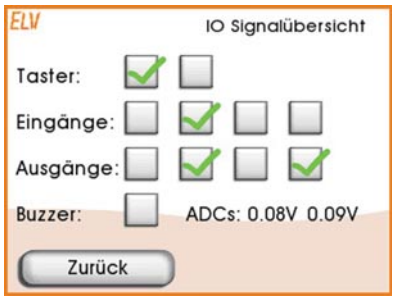

Bild 3: Die Signalübersicht zeigt den Zustand aller Ein- und Ausgänge.

```
<ui>
  <page id="adc" font="fonts/decker.ttf" font_height="30">

    <elements>
      <image id="background" x="0" y="0"
             file="images/background.png" onclick="exit"/>
      <text id="text_1" x="160" y="50" height="30" center="true"/>
    </elements>

    <triggers>
        <trigger value="intInOnboardADC_0" method="OnAdcChanged"/>
    </triggers>

    <code><![CDATA[
        proc OnAdcChanged { valueId value } {
            ui set text_1.text „$valueId = $value"
        }
]]></code>
  </page>
</ui>
```

Bild 1: Programmbeispiel zum XML-Element <trigger>

Konfiguration des Logik-Switches

Damit das funktioniert, müssen zunächst in der Konfigurationsdatei für den Logik-Switch *(/etc/icserver/icserver_system.conf)* Signale für die Ein- und Ausgänge definiert und mit den Kanälen des *pfmd* verbunden werden.

Zuordnung der XML-RPC-Kanäle

Die Definition der Kanäle des vom *pfmd* bereitgestellten „SYSTEM“-Gerätes sieht wie in Bild 4 dargestellt aus.

Unter *<devices>* sind alle über XML-RPC angebundenen Geräte aufgelistet. *<device>* definiert das „SYSTEM“-Gerät des *pfmd*. Es wird auf das In-

```
<icserver_system>
    <devices>
        <device id="INOUT" type="SYSTEM" interface="SYSTEM"
                                         address="SYSTEM">
            <channel index="1" terminal="KL1.23">
                <value id="STATE" target="binInOnboardInput_0"/>
            </channel>
[...]
            <channel index="5" terminal="Taster oben">
                <value id="STATE" target="binInOnboardInput_4"/>
            </channel>
[...]
            <channel index="7" terminal="KL1.9 - KL1.11">
                <value id="STATE" source="binOutOnboardRelay_0"/>
            </channel>
[...]
            <channel index="11" terminal="Piezo">
                <value id="STATE" source="binOutBuzzer"/>
            </channel>
            <channel index="12" terminal="ADC0">
                <value id="VALUE" target="intInOnboardADC_0"/>
            </channel>
[...]
            <channel index="13" terminal="ADC1">
                <value id="VALUE" target="intInOnboardADC_1"/>
            </channel>
        </device>
    </devices>
[...]
```

Bild 4: Die Definition der Kanäle des vom pfmd bereitgestellten „SYSTEM“-Gerätes

```
[...]
    <values>
        <!-- On board inputs -->
        <loop start="0" count="6">
            <value name="IX1.[n]" alias="binInOnboardInput_[n]"/>
        </loop>

        <!-- On board Relay -->
        <loop start="0" count="4">
            <value name="QX1.[n]" alias="binOutOnboardRelay_[n]"/>
[...]
        </loop>

        <!-- On board buzzer -->
        <value name="QX1.5" alias="binOutBuzzer" iccignore="true"/>

        <!-- On board ADCs -->
        <loop start="0" count="2">
            <value name="IL[1+n]" alias="intInOnboardADC_[n]"/>
        </loop>

    </values>
[...]
```

Bild 5: Die Definition der verwendeten Logiksignale

terface *„SYSTEM“* verwiesen, das weiter unten in der Datei definiert ist.

Unterhalb von <*device*> werden die für die Logiksignale relevanten Kanäle jeweils als ein Element <*channel*> aufgelistet.

Das Attribut *index* des Elements <*channel*> gibt die Kanalnummer an, das Attribut *id* des Elements <*value*> gibt die Id des Kanalwerts an. Die Attribute *source* für Ausgänge bzw. *target* für Eingänge stellen die Verbindung zu den weiter unten definierten Logiksignalen her.

Es werden also die Digitaleingänge auf die Logiksignale *binInOnboardInput_0* bis *binInOnboardInput_3* gelegt.

Die beiden Taster werden auf *binInOnboardInput_4* und *binInOnboardInput_5* gelegt.

Die Relais werden zu *binOutOnboardRelay_0* bis *binOutOnboardRelay_3*.

Der Signalgeber wird *binOutBuzzer* und die ADC-Eingänge schließlich werden *intInOnboardADC_0* und *intInOnboardADC_1*.

Definition der Logiksignale

Als Nächstes werden die oben bereits verwendeten Logiksignale definiert (Bild 5). Unterhalb des Elements <*values*> werden in Form von <*value*>-Elementen alle Logiksignale aufgelistet. Dabei kann mit dem <*loop*>-Element Schreibarbeit gespart werden. Das <*loop*>-Element arbeitet wie eine Programmschleife, die *count*-Zyklen durchlaufen wird. Die Laufvariable ist immer *„n“* und wird, beginnend bei *start,* mit jedem Durchlauf inkrementiert.

Innerhalb der Schleife können in Attributen mit *[]* markierte Ausdrücke verwendet werden, die auf *„n“* basieren. Die gleiche Syntax für Schleifen wird auch von immediateC-Arrays verwendet (siehe immediateC-Dokumentation).

Für jedes Logiksignal wird ein Name in IEC-1131-Notation und ein sprechender Alias angegeben. Oben in dem Abschnitt mit der XML-RPC-Anbindung werden die Logiksignale über die Aliasnamen referenziert.

So werden z. B. den Logiksignalen *IX1.0* bis *IX1.5* die Aliasnamen *binInOnboardInput_0* bis *binInOnboardInput_5* zugeordnet, welche über die obige XML-RPC-Zuordnung wiederum den 4 Digitaleingängen und den beiden Tastern entsprechen.

Das Attribut *iccignore=“true“* am Signal *binOutBuzzer* sorgt dafür, dass *xml2iha.tcl* (siehe oben) für dieses Signal keine Definition für die Logiksteuerug in der Datei *external_signals.iha* erzeugt. Der Buzzer wird nämlich von dem unten beschriebenen Logikprogramm nicht verwendet, und auf diesem Wege wird eine Fehlermeldung des immediateC-Compilers für einen undefinierten Ausgang umgangen.

Definition der XML-RPC-Interface-Prozesse

Jetzt muss *iCserver* nur noch wissen, wie der XML-RPC-Interfaceprozess für das Gerät „SYSTEM“ angesprochen werden kann (Bild 6).

Hier wird ein Interface mit dem Namen „SYSTEM“ definiert, das auf dem lokalen Rechner (IP-Adresse 127.0.0.1) auf dem Port 2002 angesprochen wird.

Programmierung der Oberflächenseite

Die oben abgebildete Oberflächenseite ist in der Datei *uidescription/iosignals.xml* programmiert. Die Definition der Oberflächenelemente birgt keine Überraschungen. Die Vorgehensweise dazu wurde bereits im letzten Artikel beschrieben, daher wird hier darauf nicht weiter eingegangen.

Neu ist jedoch die Definition von Triggern, die auf die Veränderung von Logiksignalen reagieren, siehe Bild 7.

Hier werden TCL-Methoden an Logiksignale gebunden. Bei einer Änderung eines Logiksignals wird die entsprechende Methode aufgerufen.

Die Triggermethode kümmert sich darum, das entsprechende Oberflächenelement zu aktualisieren.

Für die Digitaleingänge und Taster übernimmt dies z. B. die Methode OnInputChanged:

```
[...]
     <interfaces>
        <interface id=“SYSTEM“ url=“bin://127.0.0.1:2002“/>
    </interfaces>
</icserver_system>
```

Bild 6: Die Definition des Interface-Prozesses

```
[...]
     <triggers>
          <trigger value=“intInOnboardADC_0“ method=“OnAdcChanged“/>
          <trigger value=“intInOnboardADC_1“ method=“OnAdcChanged“/>
          <trigger value=“binInOnboardInput_0“ method=“OnInputChanged“/>
          <trigger value=“binInOnboardInput_1“ method=“OnInputChanged“/>
          <trigger value=“binInOnboardInput_2“ method=“OnInputChanged“/>
          <trigger value=“binInOnboardInput_3“ method=“OnInputChanged“/>
          <trigger value=“binInOnboardInput_4“ method=“OnInputChanged“/>
          <trigger value=“binInOnboardInput_5“ method=“OnInputChanged“/>
          <trigger value=“binOutOnboardRelay_0“ method=“OnOutputChanged“/>
          <trigger value=“binOutOnboardRelay_1“ method=“OnOutputChanged“/>
          <trigger value=“binOutOnboardRelay_2“ method=“OnOutputChanged“/>
          <trigger value=“binOutOnboardRelay_3“ method=“OnOutputChanged“/>
    </triggers>
[...]
```

Bild 7: Die Definition von Triggern, die auf die Veränderung von Logiksignalen reagieren

```
proc OnInputChanged { valueId value } {
   set index [lindex [split $valueId _] end]
   ui set checkbox_input_${index}.index $value
}
```

Die erste Zeile extrahiert aus dem als Parameter *valueId* übergebenen Signalnamen die hinter dem Unterstrich stehende Zahl. Die zweite Zeile generiert aus der extrahierten Zahl mit dem Präfix *checkbox_input_* die Id des zugehörigen Oberflächenelements, in diesem Fall eine Checkbox. Über den Befehl *ui set* (siehe Artikel in Heft 5/2011) wird dann die anzuzeigende Grafik (index 0 = ohne Haken; index 1 = mit Haken) festgelegt.

Die anderen Triggermethoden arbeiten analog.

Für die Oberflächenelemente der Relais und des Signalgebers ist jeweils eine Methode hinterlegt, die beim „Klick" auf das Oberflächenelement aufgerufen wird:

```
proc OnBuzzerClick { control } {
    set value [ui get ${control}.index]
    set value [expr !$value]
    ui set ${control}.index $value;
    control set binOutBuzzer $value
}
```

Hier wird zunächst auf den jeweils anderen Grafikindex umgeschaltet, um den angezeigten Zustand der Checkbox umzukehren. Danach wird per *control set* der Signalgeber ein- oder ausgeschaltet.

Logiksteuerung

In diesem Teil der Beispielapplikation kommt immediateC ins Spiel. Es wurde per immediateC ein Steuermodul programmiert, das für jedes Relais instantiert wird.

Das Steuermodul beherrscht vier verschiedene Modi:

- *„Dauer"* Fester Wert (ein oder aus)
- *„Regler"* Zweipunktregler mit Hysterese im Heiz- oder Kühlbetrieb; als Eingang kann jeder der beiden ADC-Eingänge verwendet werden
- *„Monoflop"* Monoflop mit einstellbarer Einschaltdauer in Sekunden, optional retriggerbar
- *„Zeitschaltuhr"* Klassische Zeitschaltuhr mit Einstellung des Einschaltzeitpunktes und der Einschaltdauer; Option für tägliches Schalten

Definition der internen Signale

Für die Parameter des Steuermoduls wird eine Reihe von internen Signalen benötigt. Diese Signale müssen pro Relaisausgang vorhanden sein. Sie werden daher in der Konfigurationsdatei zu *iCserver* innerhalb der Schleife mit den Relais (siehe oben), wie in Bild 8 zu sehen, definiert.

Die unterhalb von *binOutOnboardRelay_[n]* aufgelisteten Signale haben keine Verbindung zur Außenwelt. Sie werden von der Bedienoberfläche verwendet, um das Logikprogramm zu parametrieren. Das Attribut *default_value* gibt den initialen Wert für das jeweilige Signal an. Das Attribut *persistent="true"* sorgt dafür, dass der Wert des entsprechenden Signals über einen Neustart hinaus erhalten bleibt.

Die immediateC-Applikation

Die immediateC-Applikation befindet sich im Quellcodeverzeichnis *iclogic*. Die Applikation besteht aus verschiedenen Dateitypen:

- **.ih* immediateC-Headerdateien
- **.iha* immediateC-Headerdateien mit Arrays; diese Dateien werden vom immediateC-Array-Compiler (immac) zu „normalen" .ih-Dateien kompiliert
- **.ic* immediateC-Quellcodedateien
- **.ica* immediateC-Quellcodedateien mit Arrays, analog *.iha
- **.cpp* „normale" C++-Dateien mit Funktionen, die aus immediateC heraus verwendet werden

Übersichtsseiten

Es gibt eine Übersichtsseite zur Logiksteuerung (Bild 9), welche für alle vier Relaisausgänge den jeweiligen Betriebsmodus anzeigt *(iologiccontrol.xml)*.

Über die zum Relais gehörende Schaltfläche gelangt man zur Seite *iorelay.xml*, auf der für ein Relais der Betriebsmodus ausgewählt und zur entsprechenden Konfigurationsseite verzweigt werden kann (Bild 10).

Dem Betriebsmodus für ein Relais entspricht das Signal *intUiRelayMode_<Relaisnummer>*. Die Signale heißen also *intUiRelayMode_0* bis *intUiRelayMode_3*. Diese werden in der immediateC-Quellcodedatei *iclogic.ica* ausgewertet. Abhängig davon werden die richtigen Signale aus den anderen Modulen auf die Relais durch-

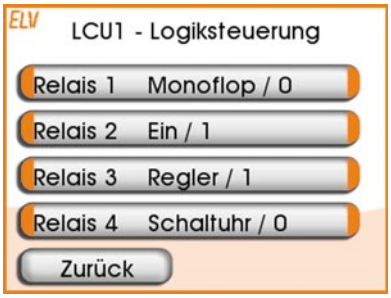

Bild 9: Die Übersichtsseite zur Logiksteuerung

```
<!-- On board Relay -->
<loop start="0" count="4">
   <value name="QX1.[n]" alias="binOutOnboardRelay_[n]"/>
   <value name="IB[1+n]" alias="intUiRelayMode_[n]" persistent="true"/>
   <value name="IL[20+n]" alias="intUiRelayMonoflopTime_[n]" default_value="5"
          persistent="true"/>
[...]
   <value name="IB[10+n]" alias="intUiRelayControllerAdcIndex_[n]"
          default_value="0" persistent="true"/>
</loop>
```

Bild 8: Die Definition der internen Signale

geschaltet. Auch die anderen internen Steuersignale werden von diesem Modul ausgewertet und als Parameter an die in den anderen Dateien als immediateC-Funktionen implementierten Module übergeben. Als zentrale Quellcodedatei bindet *iclogic.ica* per *#include* alle anderen immediateC-Quellcodedateien ein. Das ist zwar nicht schön, verhindert aber Link-Fehler.

Regler

Dieser Modus implementiert den Zweipunktregler mit Heiz- und Kühlbetrieb. Die Oberflächenseite (*iocontroller.xml*) dafür zeigt Bild 11. In Bild 12 sieht man den zugehörigen immediateC-Code aus *controller.ic*.

Die Funktion *controller()* schaltet abhängig vom Zustand der obigen Checkbox „Kühlbetrieb“ (*Signal cooling*) die Funktion *cooling_controller()* bzw. *heating_controller()* auf den Ausgang durch. Die Funktionen *cooling_controller()* und *heating_controller()* verwenden die immediateC-Standardfunktion für ein Speicherelement *LATCH()*, um den Zweipunktregler mit Hysterese zu implementieren.

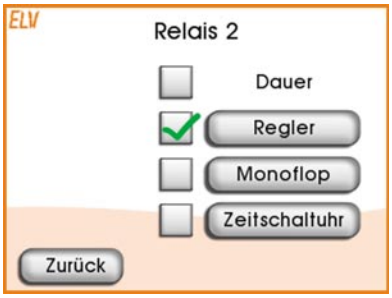

Bild 10: Die Seite iorelay.xml für die Auswahl des Relais-Betriebsmodus und die Verzweigung zur entsprechenden Konfigurationsseite

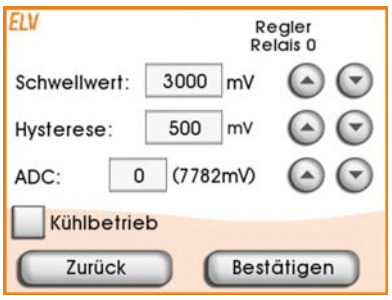

Bild 11: Die Oberflächenseite iocontroller.xml für den Zweipunktregler mit Heiz- und Kühlbetrieb

Monoflop

Dieser Modus implementiert ein optional retriggerbares Monoflop. Die Oberflächenseite (*iomonoflop.xml*) ist in Bild 13 zu sehen. Bild 14 zeigt den zugehörigen immediateC-Code aus *monoflop.ic*.

Die Funktion *monoflop()* verwendet ein Set-Reset-Flipflop (Standardfunktion *SR()*), ein D-Flipflop (Standardfunktion *D()*) sowie einen auf dem vordefinierten 10-Hz-Clock-Signal *TX0.4* basierenden Timer, um das Monoflop zu implementieren. Das SR-Flipflop wird mit der steigenden Flanke auf dem Triggersignal eingeschaltet. Gleichzeitig beginnt die Einschaltverzögerung des D-Flipflops zu laufen. Ist die Einschaltverzögerung des D-Flipflops abgelaufen, wird dadurch der Reset-Eingang des SR-Flipflops getriggert und der Ausgang fällt wieder ab. Eine positive Flanke auf dem Triggersignal startet das D-Flipflop erneut. Das Signal *retriggerable* wird als Gate für die Retriggerung verwendet. Bei *retriggerable==0* wird also die Triggerflanke unterdrückt.

Zeitschaltuhr

Dieser Modus implementiert eine Zeitschaltuhr, deren Oberflächenseite *iotimer.xml* in Bild 15 zu sehen ist. Der immediateC-Code (Bild 16) dazu befindet sich in *alarmclock.ic*. Hier wird zunächst die weiter unten verwendete C-Funktion *get_local_time()* deklariert. Diese Funktion stammt aus der Quellcodedatei *helper_functions.cpp*.

Alle Zeiten und Daten werden als 32-Bit-Zahl in der Form:

„Sekunden seit 01.01.1970 00:00:00 Uhr“

dargestellt. Im unteren Viertel der Datei wird die Variable *alarmclock_now* berechnet. Diese Variable wird jede Minute aktualisiert und enthält die aktuelle Uhrzeit (lokale Zeit). Das Signal *alarmclock_updateTime* hat zu jeder vollen Minute eine steigende Flanke. Auf diese steigende Flanke wird per immediateC-Flankenerkennung *RISE()* reagiert und die Variable

Bild 13: Die Oberflächenseite iomonoflop.xml des retriggerbaren Monoflops

```
imm bit heating_controller( imm int input, imm int threshold,
                            imm int hysteresis )
{
    this = LATCH( input < threshold, input >= threshold + hysteresis );
}

imm bit cooling_controller( imm int input, imm int threshold,
                            imm int hysteresis )
{
    this = LATCH( input > threshold, input <= threshold - hysteresis );
}

imm bit controller( imm int input, imm int threshold, imm int hysteresis,
                    imm bit cooling )
{
    imm bit out = cooling ?
        cooling_controller( input, threshold, hysteresis ) :
        heating_controller( input, threshold, hysteresis ) ;
    this = out;
}
```

Bild 12: Der immediateC-Code zum Regler

```
imm bit monoflop( imm bit trigger, imm int time, imm bit retriggerable )
{
    imm timer timer100ms = TIMER(TX0.4);
    this = SR( trigger, D( this & ~RISE(trigger & retriggerable), timer100ms,
                                                                  time * 10));
}
```

Bild 14: Der zum Monoflop gehörende immediateC-Code

alarmclock_now aktualisiert. *alarmclock_now* ist als *immC* deklariert. Das bedeutet, dass diese Variable zwar als Eingangsvariable in immediateC-Ausdrücken verwendet werden kann, die Variable selbst sich aber wie eine „normale“ C-Variable verhält, die nach jeder Zuweisung ihren Wert einfach speichert.

Weiter oben befindet sich die Funktion *iotimer()*, welche abhängig vom Signal *daily* (entspricht der Checkbox „täglich“) entweder den Ausgang von *alarmclock_daily()* oder den von *alarmclock()* auf den Ausgang durchschaltet.

Die beiden *alarmclock*-Funktionen oben in der Datei berechnen dann einfach durch einen Vergleich der Startzeit und der Endzeit mit dem Signal *alarmclock_now,* ob das Ausgangssignal 0 oder 1 sein soll.

alarmclock_daily() ignoriert dabei per *modulo 86400* (= Anzahl Sekunden am Tag) den Datumsteil der Startzeit.

Bild 15: Die Oberflächenseite der Zeitschaltuhr

```
%{ int get_local_time(); %}

imm bit alarmclock( imm int startTime, imm int endTime )
{
    extern immC int alarmclock_now;
    this = (alarmclock_now > 0) & (alarmclock_now >= startTime) &
           ( (endTime == 0) | (alarmclock_now < endTime) );
}

imm bit alarmclock_daily( imm int startTime, imm int endTime )
{
    extern immC int alarmclock_now;
    imm int time_of_day = alarmclock_now % 86400;
    this = ( endTime >= startTime ) ?
           ( (time_of_day >= startTime) & (time_of_day < endTime) ) :
           ( (time_of_day >= startTime) | (time_of_day < endTime) );
}

imm bit iotimer( imm int startTime, imm int duration, imm bit daily,
                 imm bit useDuration )
{
    imm int endTime = useDuration ? startTime + duration : 0;
    this = daily ? alarmclock_daily( startTime % 86400, endTime % 86400 ) :
                   alarmclock( startTime, endTime );
}

imm timer alarmclock_timer1s = TIMER1(TX0.5);
immC int alarmclock_now;

imm bit alarmclock_updateTime =
        D( ~alarmclock_updateTime, alarmclock_timer1s, 59-(alarmclock_now%60));

if( RISE(alarmclock_updateTime) )
{
    alarmclock_now = get_local_time();
}
```

Bild 16: Der Code aus alarmclock.ic für die Zeitschaltuhr