Aligner 8 Due

8“ Aligner mit Dual Motorcontroller

Raspberry Pi mit Buildroot_2015_Xenomai
IO via AVR am I2C Bus
Visualisierung mit Pi TFT und HMI2 Bibliothek
Witty2 als Web-Interface für Loging und Firmwareupdates

Setup

Zur Erstinbetriebnahme sind folgende Schritte notwendig:

1. Erstellung einer Boot SD für den Raspberry Pi

2. Aufspielen der Firmware auf den Motorcontroller

3. Test der einzelnen Motoren

4. Konfiguration der Parameter

5. Einrichten des OCR Lesers

6. Funktionstest gegen den CAM Testserver

Boot SD

Für den Raspberry Pi genügt eine 4GB SD mit FAT32 Dateisystem. Es muss keine spezielle Bootpartition etc. angelegt werden. Vor dem Gebrauch alte Dateien entfernen oder neu formatieren.

Die Dateien und Ordner im Distributionsverzeichnis einfach auf die SD kopieren. Für den ersten Bootvorgang ist es sinnvoll einen eindeutigen Netzwerknamen zu vergeben, hierzu kann die Datei „hostname.ini“ entsprechend angepasst werden. Der Pi sollte nun booten und unter dem angegebenen Name im Netzwerk erreichbar sein.

Mit der Browser-Administrations-Oberfläche können nun die weiteren Einstellungen vorgenommen werden. Wie dies im Details funktioniert ist in einem gesonderten Kapitel beschrieben.

Firmware

Der AVR Mikrocontroller ist mit seiner Programmierschnittstelle mit den GPIOs des Pi verbunden und kann darüber mit der notwendigen Firmware versorgt werden. Hierzu wir das Programm „avrdude“ verwandt das auf dem Pi schon installiert ist und über die Weboberfläche gesteuert wird.

Die Firmware liegt in verpackt in deiner ZIP Datei vor, die ein Shellscript sowie das eigentliche Programm als Hexfile enthält. Die Skriptdatei enthält alle notwendigen Schritte zur Inbetriebnahme des AVRs. Damit keine unkontrollierte Veränderung der Firmware möglich ist, wird der Reiter „AVRDude“ durch ein Passwort gesichert.

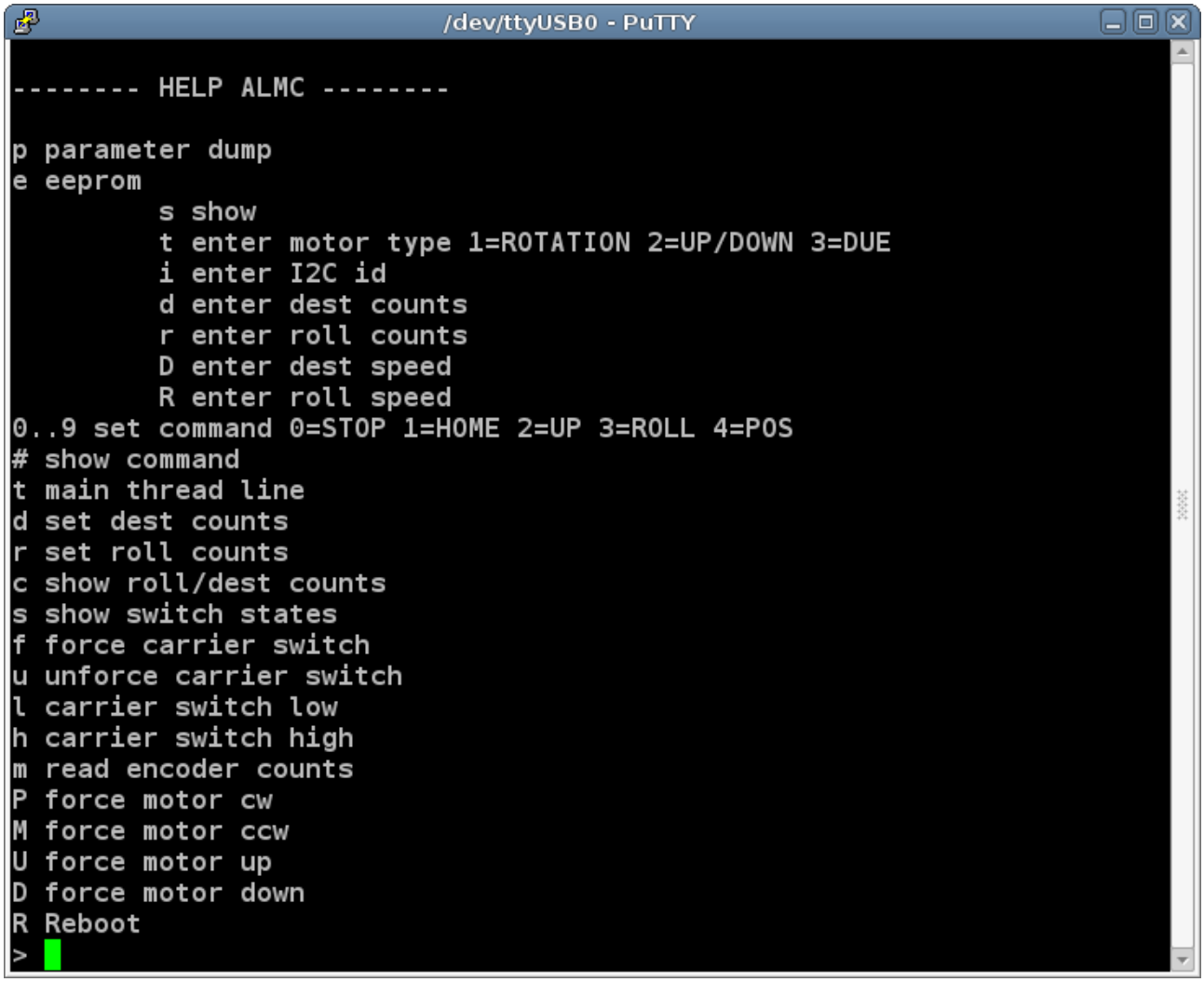
Ein erster Test ob der Controller überhaupt über das Programmierinterface erreichbar ist, besteht im Auslesen der Fuses. Hierzu muss vorab sowohl der richtige Controllertyp wie auch die Schnittstelle ausgewählt werden. Verwendet wird ein „AtMega328p“ an „gpio1“. Achtung, es gibt auch Varianten des Mikrocontrollers ohne „p“ am Ende die sich in der ID unterscheiden und vom „avrdude“ auch als verschieden Typen angesehen werden. Bitte keine Hexfiles aufspielen wenn die Fuses nicht korrekt gesetzt sind, am besten immer den Weg über ein Paket wählen.

Alle Dateien die über das Webinterface geladen und installiert werden müssen im Dateisystem des Rechners liegen auf dem der Webbrowser läuft, gegebenenfalls eine Verbindung herstellen oder eine lokale Kopie verwenden.

Unter dem Link
„http://jennifer/raid/archive_fab_projects/aligner_8_due/install/pi_nf8_1/firmware.zip“

steht das Firmwarepaket für den Download bereit. Nach der Installation die Ausgabe von „avrdude“ beachten, nur wenn keine Fehler aufgetreten sind kann mit der Inbetriebnahme fortgefahren werden.

Wurden bereits schon alle Komponenten, Motor und Schalter getestet und ist die korrekte Verdrahtung und Funktion sicher gestellt, können die Motoren mit der Steuerplatine verbunden werden. Ist dies nicht der Fall, ist vor weiteren Test der Auf/Ab-Motor Stecker zu ziehen. Über die serielle Schnittstelle des AVR, kann mit Hilfe eines FTDI-USB-Kabels der Debugger auf dem Motorcontroller angesprochen werden. Die Geschwindigkeit ist auf 57600 Baud eingestellt.



Die wichtigsten Befehle zum Testen:

s: Zeigt den Zustand der beiden Schalter an. Achtung: Schalter geschlossen entspricht einer 0, Schalter offen einer 1.

P: Motor positive Drehrichtung für 1 Sekunde.

M: Motor negative Drehrichtung für 1 Sekunde.

U und **D** entsprechend für den Auf / Ab Motor.

Achtung Endlagen werden nicht geprüft, Motorspindel kann sich festfahren !

Mit den Ziffern können die einzelnen Funktionen entsprechend dem normalen Programmablauf einzeln ausgeführt werden. Diese Funktionen können nur sinnvoll eingesetzt werden wenn das Steuerprogramm auf dem Raspberry-Pi gestoppt wurde. Soll der Test ohne Carrier durchgeführt werden, kann der Schalter mit folgenden Kommandos gesteuert werden.

f: Force, Überschreibung möglich

l: Low, Schalter aktiviert, Carrier vorhanden

h: High, Schalter inaktiv, kein Carrier

Fahrkommandos:

0: Stop

1: Home, Initialisierung Auf/Ab Motor, fährt bis zum Endlagenschalter

2: Up, Auf/Ab Motor fährt eine bestimmte Anzahl Impulse aufwärts. Der Wert ist im EEPROM hinterlegt. Der Befehl wird nur ausgeführt wenn beim Start der Endlagenschalter aktiv war. **Achtung falls die Anzahl der Impulse zu groß ist, besteht die Gefahr das sich der Motor fest fährt.**

3: Roll, drehen um alle Wafer alle in die Startposition zu bringen, Notch rastet ein.

4: Pos, Notch nach oben positionieren, die Impulse sind im EEPROM gespeichert.

EEPROM Kommandos:

e: eeprom, leitet ein Kommando ein.

- **s:** Show, zeigt den Inhalt des Speichers an
- **t:** Typ, Eingabe Motortyp
- **i:** ID, Eingabe der I2C Adresse
- **d:** Destination, Eingabe Zielwert Impulse Auf/Ab
- **r:** Roll, Eingabe Zielwert Rotationsachse, Notch Unten bis Notch Oben ½ Umdrehung
- **D:** Geschwindigkeit Auf/Ab
- **R:** Geschwindigkeit Rotation

Achtung, die in das EEPROM geschriebenen Werte werden erst nach einem Reboot der Motorcontrollers wirksam! Aus Sicherheitsgründen werden falsche Werte beim Booten sicherheitshalber begrenzt. Auf/Ab maximal 8000, Rotation maximal 25000. Die Geschwindigkeit sollte zwischen 0 und 255 liegen.

Verschiedene Kommandos:

R: Reboot Motorcontroller

#: Anzeige des aktuellen Zustands, entspricht der Nummer des Kommandos solange dies aktiv ist. Wird dies korrekt zu Ende geführt dann wird die Zustandsnummer um 10 erhöht. Folgende Werte sind in „aligner.h“ definiert:

```
#define CMD_STOP                    0
#define CMD_HOME                    1
#define CMD_UP                      2
#define CMD_ROLL                    3
#define CMD_POS                     4
#define CMD_5                       5
```

```
#define CMD_ABORT                   6
#define CMD_ERROR                   7

#define CMD_NACK                    8
#define CMD_ACK                     9

#define ACK_STOP                    10
#define ACK_HOME                    11
#define ACK_UP                      12
#define ACK_ROLL                    13
#define ACK_POS                     14
#define ACK_5                       15
#define ACK_ABORT                   16
#define ACK_ERROR                   17
```

t: Thread Position, Zeilennummer aus dem Main Thread für Debug Zwecke.

d: Destination, manuelle Zieleingabe

r: Rotation, manuelle Zieleingabe

c: Counts, Zählerwerte Ziel abfragen

m: Motor, aktuelle Zählerstände der Motoren

Üblicherweise wird die Schnittstelle nur für die Erstinbetriebnahmen und zur Fehlersuche auf dem Motorcontroller verwendet. Alle Parameter, auch der Inhalt des EEPROMs lassen sich komfortabel über die Web-Schnittstelle konfigurieren.

Anwendungsprogramm LUA

Als Basis dient ELFI 4.0, ein auf die Belange der Industrie und des 24/7 Betriebs hin optimiertes Buildroot Linux, in Kombination mit LUA als Scriptsprache. Grundsätzlich lassen sich alle für den Anwender relevanten Aufgaben über das Webinterface durchführen. Alternativ kann auch via SSH das Dateisystem von einem externen Rechner aus angesprochen werden oder eine SSH Remoteshell zur Programmierung und Systemadministration verwendet werden. Eine geeignete Entwicklungsumgebung hierzu bietet das Eclipse Projekt mit CDT und Remote System Tools.

Die Verwendung der Webschnittstelle ist insbesondere bei der Konfiguration, allerdings einer IDE oder SSH vorzuziehen, da sie einen Tabelleneditor für die LUA Konfigurationsdateien bietet. Der Editor Tab „LUA Admin“ ermöglicht eine syntaktisch korrekte Bearbeitung der Daten über entsprechende Eingabefelder an. Allen Tabellen gemeinsam ist die Möglichkeit sie mit „Save“ lokal im RAM des Gerätes für Testzwecke zu speichern und mit „Apply“ entsprechend anzuwenden. Es kann notwendig sein, das hierzu das Skript neu gestartet werden muss, der entsprechenden Befehl kann auf der Seite „LUA Script“ mit dem Knopf „Run“ ausgeführt werden.

Solange eine Script oder eine Konfiguration nicht mit „Save SD“ permanent gespeichert wurde, geht sie bei einem Neustart des System verloren. Achtung paralleles Bearbeiten über Web und SHH kann zu Inkonsistenz führen !

Eine Tabelle besteht aus dem Namen der Variablen, welcher aus Modulname und dem eigentlichen Namen, verbunden mit einem „.“ , besteht. Es existieren 3 Typen von Variable: Listen gekennzeichnet durch „{}“, Zahlen kenntlich durch „#“ sowie Zeichenketten mit dem „$“ in der Typ-Spalte. Im Feld Info steht ein Kommentar, der die Funktion näher beschreibt. Die eigentlichen Daten befinden sich im Feld Wert und sind, nach einem Login, änderbar. Ebenfalls können die Kommentare den eigenen Bedürfnissen angepasst werden. Soll der Wertebereich eingeschränkt werden ist dies durch Verwendung von Drop Down Menüs möglich. Wie dies im Detail geschieht und wie ein Konfigurations-Modul mit dem Scripteditor erweitert werden kann ist im Abschnitt Webinterface Witty erläutert.

Die jeweilige Konfiguration ist in Module eingeteilt, die einzeln mit „Load“ geladen werden können, sie werden im Folgenden detailliert dargestellt.

Modul locale

Dieses Module legt die einzelnen Zeichenketten fest die für die Ein und Ausgabe verwendet werden fest. Hier ist es auch möglich eine andere Sprache zu wählen, daher der Name. Die Listen-Variable „locale.caminfo“ besitzt acht Element wie anhand der nachgestellten Zahl erkennbar ist. Im Lua Script sind solche Variable wie Arrays ansprechbar, z.B. :

locale.caminfo[idx] liefert mit einem Wert von idx = 3 die Zeichenkette „Los nicht gefunden“

Name	Typ	Wert	Info
locale.caminfo	{}		CAM3 Meldungen
locale.caminfo.1	$	Los / Anlage falsch	Fehlertext
locale.caminfo.2	$	Anlage down	Fehlertext
locale.caminfo.3	$	Los nicht gefunden	Fehlertext
locale.caminfo.4	$	Rezept falsch	Fehlertext
locale.caminfo.5	$	SCN	Fehlertext
locale.caminfo.6	$	Deamon	Fehlertext
locale.caminfo.7	$	Unbekannt	Fehlertext
locale.caminfo.8	$	Deamon	Fehlertext
locale.equipment	$	Anlage	Display Label Equipment
locale.header	$	DNS6 und DNS7	Display Kopfzeile
locale.lot	$	Losnummer	Display Label Los
locale.spec	$	Spec	Display Label Spec
locale.split	$	Split	Display Label Split
locale.variant	$	Variante	Display Label Variante
locale.version	$	Version 1.0	Software Version

Modul app

Das Modul app ist die zentrale Stelle an der das Anwendungsprogramm für den jeweiligen Anwendungsfall konfiguriert werden kann, die Variablen im Einzelnen:

Name	Typ	Wert	Info
app.camFormat	$	5100 xxx 5101 xxx_xxx 5110 xxxx;xxxx	Format CAM Antwort

Das Format der CAM Antwort kann aus drei Varianten gewählt werden. Die Namen erinnern an die zugehörigen TCP-Ports des CAM Servers. Die Formate variieren leicht, dahinter stehen für

spezielle Bereiche optimierte Anfragen an CAM. 5100 liefert nur die dreistellige Variante, 5101 zusätzlich vorangestellt die Spezifikation. 5110 hängt hinten die Splitlos Nummer an und benötigt hierzu aber zusätzlich die Wafernummer vom OCR Reader.

Name	Typ	Wert	Info
app.ocrFormat	$	Losnummer Losnummer;Wafernummer	Format CAM Antwort

Das OCR Format wird durch den angeschlossen Reader bestimmt und kennt passend zum CAM Format zwei Varianten, mit und ohne Wafernummer. Benötigt das CAM Format eine Wafernummer und wird nur die Losnummer geliefert, so wird diese standardmäßig auf „01“ gesetzt.

Name	Typ	Wert	Info
app.camMode oder app.ocrMode	$	CAM oder OCR Manuell Aus	Anfrage Modus

Für Test- und Spezialzwecke ist es möglich den Verarbeitungsmodus für die OCR bzw. CAM Daten einzustellen. „CAM bzw. OCR“ stellen den Standardmodus dar, „Manuell“ ermöglicht es die Daten in den Feldern „app.camTestReply“, „app.ocrTestLot“ und „app.ocrTestWafer“ vorzugeben und so Antworten von OCR Reader und CAM zu simulieren. Hierzu sind genau Kenntnisse des Aufbaus der Nachrichten der einzelnen Formate notwendig. Mit „Aus“ kann die Anfrage ganz deaktiviert werden. Eine mögliches Szenario besteht darin nur die Wafer auszurichten oder ausschließlich die Losnummer anzuzeigen.

ELFI 4.0

Embedded Linux

Raspberry Pi mit Buildroot_2015_Xenomai
IO via I2C und SPI Bus
Visualisierung mit Pi TFT und HMI2 Bibliothek
Witty2 als Web-Interface für Loging und Firmwareupdates

Setup

1. Schritt Vorbereitung der SD Karte.

Den Inhalt aus dem entsprechenden „dist_xxx“ Ordner auf eine leere SD Karte mit FAT32 System kopieren. In den PI einlegen und Stromversorgung einschalten. Die Grüne LED sollte kurz aufleuchten. Falls es sich um eine Distribution mit „BootTFT Kernel“ handelt dann ist auch der eigentliche Bootvorgang auf dem TFT Display sichtbar. Wenn dies nicht der Fall ist dann kann am HMI Ausgang zur Fehlersuche ein Display angeschlossen werden. Ist die Konsole aktiviert „ttyAMA0 Kernel“ dann kann der Bootvorgang mit Hilfe eines USB Adapters am seriellen Port des PI mitverfolgt werden.

2. Netzwerk und Hostname

Falls die weitere Konfiguration über das Netzwerk erfolgen soll, kann die Netzwerkadresse und der Hostname auf der SD Karte mit einem einfachen Texteditor angepasst werden. Defaultmäßig ist DHCP eingestellt und der Name ergibt sich aus der Distribution, z.B. „dist_hmi“ hat als Hostname „pi_hmi“. Achtung, der Hostname sollte im Netzwerk eindeutig sein und möglichst rasch geändert werden, anschließend ein Reboot durchführen.

3. Konfiguration

Die weiter Konfiguration kann dann über die Weboberfläche stattfinden, Details siehe unten im Kapitel „Web Administration mit Witty2“ . Sie beinhaltet auch die Installation von Firmware auf den AVR Huckepackboards die über SPI und I2C mit dem PI kommunizieren. Hierzu stehen neben C-Bibliotheken auch LUA-Bindings zur Verfügung, so das auch auf Scriptebene Datenaustausch

mit der Zusatzhardware möglich ist. Mehr hierzu im Abschnitt „eLua Scripting“.

4. Debuggen

Wenn das Netzwerk korrekt eingerichtet ist und funktioniert ist der Zugriff auch Remote über ein ssh Verbindung möglich. Der File transfer erfolgt über sftp und kann mit den Remote System Tools von Eclipse direkt aus dem Projekt benutzt werden. Remote Debugging mit dem gdb ist sowohl auf der Kommandozeile, lokal wie auch remote, sowie aus Eclipse heraus möglich.

Linux Buildroot

I2C und SPI Hardware

Da der Raspberry Pi nur eine begrenzte Anzahl von Digitalen I/Os besitzt müssen Analoge Signale durch geeignete Hardwareergänzungen zur Verfügung gestellt werden. Auch für die Digitalen Signale gibt es einfache Huckepackboards die für die notwendige Signalaufbereitung sorgen. In vielen Fällen reicht einfache Hardware aus, da mit der Echtzeiterweiterung Xenomai hinreichend hohe Abtastraten erreicht werden können. Größere Freiheiten bieten Erweiterungen die auf AVR Mikrocontroller basieren und mit dem PI via I2C oder SPI Daten austauschen. Die in den Controllern enthaltene Peripheriebausteine, wie Analogeingänge, Timer und PWMs erweitern das Einsatzspektrum und ermöglichen auch eine unabhängige Datenvorverarbeitung.

DAQ Data Acquisition System

Im Gegensatz zum HMI besteht das Datenerfassungsmodul DAQ aus einer eLua Laufzeitumgebung die keine Daten auf einem Bildschirm oder einer Schnittelle ausgibt sondern in einem lokalen Speicher ablegt. Die im „shared memory“ gespeicherten Daten stehen allen Prozessen offen die sie weiter verarbeiten wollen. Der wichtigste Client ist der virtuelle Wago Prozess „vwago“ der diese Daten an der standardisierten Wago-Modbus-Schnittstelle zur Weiterverarbeitung im Netzwerk zur Verfügung stellt.

Der Zugriff auf die Hardware, sowie auch auf das Shared Memory ist auch aus LUA heraus möglich, hierzu stehen die entsprechenden Bibliotheken zum dynamischen nachladen zur Verfügung.

Ein kleines Beispiel aus der Distribution „dist_daq“ soll dies verdeutlichen:

```
require "env"
socket      = require('socket')

PLC = require('vwago')

periphery = require('periphery')
SPI = periphery.SPI

channelScale = {};

function readAD(chan)
    local data_in = spi:transfer({0x18+(chan-1), 0x00, 0x00})
    return bit32.rshift( bit32.lshift(data_in[2],8) + data_in[3],4)
end

function setup()
      spi = SPI("/dev/spidev0.1", 0, 1e5);
      channelScale[1] = app.getScale(1);
      sampleTime = app.sampleTime[1];

      plc = PLC.VWago("wago")
```

```
end

function loop()
      analog_1 = readAD(1)*channelScale[1];

      print(string.format("%4.1f",analog_1))

      if ( plc ) then
            plc:setReal(1,analog_1)
            plc:update()
      end

      socket.sleep(sampleTime)
end

io.stderr:write("Run LUA\n");
```

Serielle Schnittstelle

Die Lua Bibliothek „periphery“ die auch die Treiber für SPI und I2C enthält, stellt auch eine Reihe von Funktionen bereit mit der sich die serielle Schnittstelle komfortabel ansteuern lässt.

```
periphery = require('periphery')
Serial    = periphery.Serial

function setup()
    .....
    serial = Serial("/dev/ttyAMA0", 9600)
    .....
end

function loop()
    .....
    while serial:poll(0) do
       c = serial:read(1)
       if c == "\r" then
           io.stderr:write("\n")

       end
       io.stderr:write(c)
    end
    .....
end
```

Wie üblich wird die Bibliothek mit „require“ geladen und eine Instanz der Klasse „Serial“ angelegt. Dem Konstruktor können Gerätenamen und Baudrate übergeben werden, alle anderen Parameter werden auf Defaults (8Bit, keine Parität, kein Handshake, 1StopBit) gesetzt. Mit der Methode „poll“ kann die Schnittstelle abgefragt werden, ist der Parameter „timeout“ wie hier auf Null gesetzt wird nicht blockiert und die Funktion kehrt mit „true“ zurück falls Daten zur Abholung bereit stehen. Mit „read(1)“ wird dann genau ein Zeichen abgeholt und auf der Konsole ausgegeben. Die genauen Funktionsbeschreibungen und weitere Details sind in der Bibliotheksdokumentation enthalten.

Zusätzliche Bibliotheken

Die Scheduler Bibliothek enthält einig Klassen die für Steuerung des zeitlichen Ablaufs nützlich sein können. Ein Timer ermöglicht es in der Hautschleife gezielt einzelne Aufgaben zeitlich geordnet auszuführen. Ein Timer startet bei seiner Erzeugung und wenn die Methode „restart()“ aufgerufen wird.

```
Scheduler = require('scheduler')

function setup()
      .....
      updateTimer = Scheduler.Timer(1000)
      .....
end

function loop()
      ......
      if updateTimer.elapsed() then
            updateTimer.restart()
            .......
      end
      ......
end
```

Die Methode „elapsed()“ fragt den Timer ab und ist genau einmal „true” nach dem die Zeit abgelaufen ist. Im Gegensatz hierzu liefert die Methode „running()“ ein stetiges Signal.

```
Scheduler   = require('scheduler')

updateTimer = Scheduler.Timer(5000)
oneSecond   = Scheduler.Timer(1000)

while updateTimer:running() do
      if oneSecond:elapsed() then print("1 Sekunde ist um") end
end
```

Die Schleife wartet 5 Sekunden und nach 1 Sekunden wird der Text genau einmal ausgegeben.

eLua Scripting

Um die Flexibilität zu erhöhen sind alle wichtigen Funktionen auch über die Scriptsprache Lua zugänglich. In der HMI2 Bibliothek ist ein Lua Interpreter integriert und die HMI2 Anwendung hmi erwartet standardmäßig ein lua Script mit dem Namen run.lua um es auszuführen, falls nicht eine andere Datei auf der Kommandozeile angegeben wird. Hmi erzeugt eine App Objektinstanz die im Lua Programm referenziert werden kann und als Basis für das Benutzerinterface dient.

Das eigentliche Programme besteht dann aus zwei Funktionen die im Script hinterlegt werden müssen. Nach dem Laden des Scripts und dem Anlegen der globalen Variablen, wird nach der Erzeugung des App Objekts die Funktion setup() ohne Parameter aufgerufen. Als ersten Schritt sollte eine Referenz auf diese Instanz angelegt werden! Die nachfolgenden Schritte entsprechen dem Vorgehen wie in der HMI2 C++ Bibliothek. Layout definieren und einzelne Objekte anlegen.

```
function setup()
    instance    = HMI.App.instance;
    layout      = HMI.Layout(instance,32,24);

    label = HMI.Label(layout,0,0,32,5,“Hallo“);
    label:setFont(HMI.fonts.fontBig);
    label:setTextColor(HMI.Color.white);
    label:setBackground(HMI.Color.blue);
end
```

Die Syntax von Lua unterscheidet sich an einigen Punkten von C++. Die Bezeichner für den Zugriff auf Konstanten, statische Elemente und Namespaces werden über den '.' mit einander verbunden z.B. „HMI.Color.white“ oder „ HMI.App.instance“. Objekte, die es so in Lua nicht gibt und über Metatabellen realisiert werden, sprechen über den Operator ':' ihre Methoden an, wie zum Beispiel das Textfeld label: „label:setTextColor(HMI.Color.white);“. Konstruktoren werden wie statische Element behandelt, der Destructor wird über den Garbage Collector (GC) der Speicherverwaltung von Lua aus aufgerufen. Deshalb sollten GUI Objekte immer innerhalb der Funktion setup(), während der Laufzeit definiert werden, damit sie nicht am Ende des Ladevorgangs u.U. irrtümlich vom GC entsorgt werden

Nach der Initialisierung, wird zyklisch die Ereignisschleife der HMI2 Bibliothek abgearbeitet und die Input-Events von Tatstatur und Touchscreen bzw. Maus an die entsprechenden HMI2 Objekte weitergeleitet. Die HMI2lib kann über Callback-Funktionen, die Abarbeitung dieser Ereignisse auch an Lua Funktionen delegieren. Zwischen der Verarbeitung der Ereignisse und dem Auffrischen des Bildschirmes wird falls vorhanden die Lua Funktion loop() aufgerufen.

Ausgabe Objekte können innerhalb von loop() durch Zuweisung eines neuen Wertes aufgefrischt werden. Die Vorgehensweise soll am Beispiel eines einfachen Zählers gezeigt werden. Zuerst ein passendes Output Objekt in der Funktion setup() erzeugen:

```
counterOut = HMI.Output(layout,10,11,12,6,"0000");
counterOut:setFont(HMI.fonts.fontBig);
```

und anschließend in loop() inkrementieren und als Text in das Objekt einfügen:

```
function loop()
   counterOut:setText(counter);
   counter = counter + 1;
end
```

Die Idle Funktion der HMI2 Bibliothek wird alle 10ms aufgerufen, mit ihr verbunden ist auch die Lua Funktion loop(), d.h. das der Zähler maximal 100 Schritte pro Sekunde zählen kann. Werden innerhalb der Hauptschleife z.B. Messwerte über SPI oder I2C abgefragt, so liegt auch die maximal erzielbare Abtastrate bei 100Hz. Wird Echtzeitverhalten gewünscht, so sollte dies immer in einem entsprechen hoch priorisierten eigenen Prozess erfolgen, harte Echtzeit erfordert die Einbeziehung der Realtime Erweiterung Xenomai.

Callback bzw. Eventhandler ermöglichen es die entsprechenden Aktionen auch in Lua auszuführen. Hierzu ist eine passende Funktion dem HMI Objekt zu übergeben, dies kann auch eine anonyme Funktion mit der passenden Argumentgenanzahl sein. Der Handler wird mit „registerHandler(function)“ Lua bekannt gemacht und mit der Methode bind(handler) des HMI Objekts, mit diesem verbunden. Die Touch Ereignisse werden entsprechend ihrer Koordinaten dann über das Objekt weitergeleitet, welches seinerseits via Callback die entsprechende Lua Funktion, den registrierten und gebundenen Handler, ausführt. Nachfolgend ein Beispiel das dies verdeutlichen soll.

Zuerst wird in setup() ein Button Objekt erzeugt:

```
buttonClear = HMI.Button(layout,10,18,12,6,"Clear")
buttonClear:setFont(HMI.fonts.fontBig);
```

und mit einem Handler verbunden:

```
buttonClear:bind(registerHandler(
      function(x,y,typ)
            print("LUA Touch @ "..x..","..y);
            counter = 0;
      end
));
```

Eine Button Touch Handler erwartet eine Callbackfunktion mit drei Parametern, x und y Koordinate und dem Ereignistyp. Realisiert als anonyme Funktion, die direkt registerHandler(..) übergeben wird. Die zurückgegeben Referenz wird ohne Zwischenschritt direkt an den Button gebunden. Über den Touchscreen kann so der Zähler zurückgesetzt werden.

Es ist auch möglich direkt aus dem Handler eine Funktion aus dem HMI API aufzurufen. In der HMI Bibliothek gibt es zwei Arten von Objekte, Ausgabeobjekte die sich aktualisieren wenn neue Daten geschrieben werden und Gestalterische Elemente die nur auf explizite Anweisung neu gezeichnet werden, wie z.B. Labels. Wenn asynchron, z.B. durch Events auserhalb der Hauptschleife letztere verändert werden, sollte am Ende der Funktion loop() die gesamte Instanz der Seite aufgefrischt werden.

```
Setup()
...
name = HMI.Label(layout,0,0,32,5,locale.header[1]);
...
```

Im Codeschnippel Handler, wird aus einer Liste dem Objekt Name eine neue Anlage zugewiesen und der Index in die Liste um ein erhöht. Da die Labels ihren Inhalt nur im Konstruktor neu zeichnen, muss der Inhalt noch mit draw() auf den Bildschirm geschrieben werden.

```
Handler
...
      equipment = app.getEquipment(equipIdx);
      name:setText(equipment);
      equipIdx = equipIdx + 1;
      if equipIdx > #app.equipment then
            equipIdx = 1;
      end
      needUpdate = true;
...
```

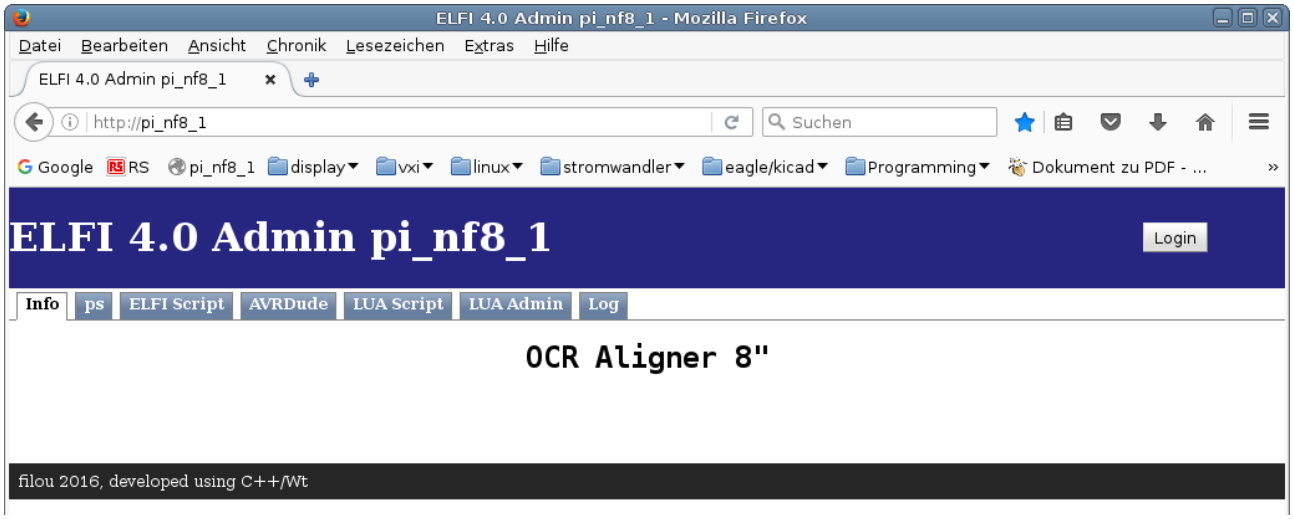
Sinnvollerweise geschieht dies für alle Element gemeinsam, die Variable needUpdate signalisiert dies der Hauptschleife.

```
loop()
...
      if needUpdate then
            instance:draw();
            needUpdate = false;
      end
end;
```

Damit wird verhindert das zu oft und unnötigerweise das komplette Bild neu gezeichnet wird, da sich die eher dekorativen Elemente selten ändern, bzw. das Seitenlayout oft als ganzes neu erstellt wird. Ein Flicker und Vorhangeffekt lässt sich dadurch ebenfalls vermeiden und das Benutzerinterface bleibt auch bei großer Zahl an Elementen flüssig bedienbar.

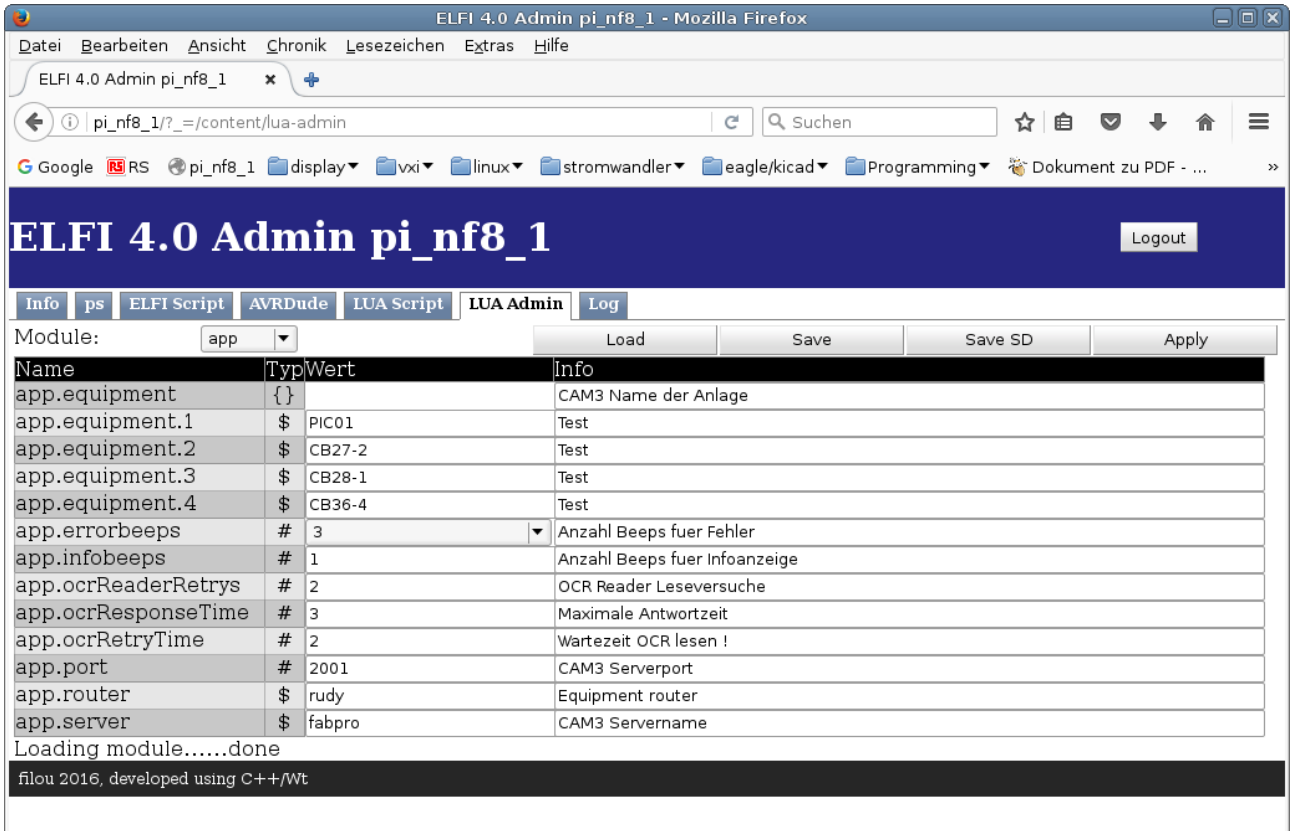
HMI2

Web Administration mit Witty2



Die Konfiguration der einzelnen Parameter ist über des Web-Interface nach dem Login möglich. Auch die Einzelnen Skripte können online editiert werden. Auch ohne Login sind Prozessstatus und Logfiles einsehbar.

Die Parameter sind in Gruppen zusammengefasst und können auf der Seite LUA Admin in tabellarischer Form editiert werden. Die verschiedenen Module können über ein Auswahlfeld selektiert werden, z.B. app:



Die Tabelle umfasst 4 Spalten, Name, Typ, Wert und Info.
Auf die Variable kann unter dem angegebenen Name im je nach Typ auf verschiedene Art und Weise im LUA Script zugegriffen werden. Handelt es sich um eine Zeichenkette (Typ $) oder um eine Zahl (Typ #) in einem einfachen Eingabefeld so steht der Wert im ersten Element der Tabelle mit dem Namen, z.B.

```
port = app.port[1]
```

Handelt es sich um ein Auswahldialog so wird der Index auf das entsprechende Feld im Tabellenfeld idx und die Werte in der Untertabelle alt abgelegt. Der Zugriff aus dem Script erfolgt dann in zwei Stufen:

```
pos = app.equipment[idx]
equipment = app.equipment[pos][1]
```

Arrays bzw. Tabellen die mit natürlichen Zahlen indiziert sind (Typ {}) können entsprechend verwendet werden. Der Zugriffe erfolgt analog zu oben:

```
idx = 3
locale.caminfo[idx][1]
```

liefert den Wert zu locale.caminfo.3

Die einzelnen Tabellen müssen zuerst im Scripteditor angelegt werden, einige Beispiele:

```
locale.caminfo    = { inf="CAM3 Meldungen",     {"Error 1",inf="Fehlertext"},
                                                {"Error 2",inf="Fehlertext"},
                                                {"Error 3",inf="Fehlertext"},
                                                {"Error 4",inf="Fehlertext"},
}

locale.version    = {"Version 1.0",             inf="Software Version"};

app.errorbeeps    = {idx=3,alt={1,2,3,4,5,6},   inf="Anzahl Beeps für Fehler"};

app.port          = {2001,                      inf="CAM3 Serverport"};
```

Details und die weitere Verwendung der Tabellen sind weiter oben im Abschnitt zu eLua beschrieben. Die Aufteilung in einzelne Module respektive Dateien, ermöglicht es die Konfiguration in Tabellenform vorzunehmen und auch vor dem Speichern bzw. Anwenden auf Konsistenz zu prüfen. Syntaxfehler und Fehleingaben die zu inkonsistentem Code führen lassen sich so nahezu vermeiden.

Die Funktionen apply() und verify() der einzelnen Module dienen dazu, falls vorhanden, die obige Funktionalität umzusetzen. Zusätzlich besteht die Möglichkeit die Daten in ein entsprechendes Format zur Weiterverarbeitung zu konvertieren und einem anderen Programm zu übergeben. Die zusätzlichen Funktionen sind nicht in den Moduldateien definiert sondern werden nur vom Webinterface aus der Konfigurationsdatei geladen, daher stehen sie auch nicht dem Anwendungsprogramm zur Verfügung. Typischerweise werden in der Datei „cfg.lua“ die leeren Modultabellen angelegt, die Tabellenwerte aus den Moduldateien mit „require“ geladen und die jeweiligen Prüffunktionen definiert. Ein kleiner Auszug aus der „cfg.lua“ für das Webinterface des

8“ Aligners. In apply() werden lediglich Hinweise angegeben wie weiter zu verfahren ist, die Ausgabe erfolgt als HTML im Textfeld unterhalb der Tabelle. Die Funktion validate() ist lediglich als Prototyp vorhanden.

```
---------------- Application module app configuration ---------------

app = {}
require "app"

function app.apply()
      print(
            "<br><b><center>Apply module app !</center></b><br>"..
            "<center>Damit die &Auml;nderungen wirksam werden, „
            „m&uuml;“ssen sie die Anwendung neu starten !</center><br>"
      );
end

function app.validate()
      print("<br><b><center>Validate modlue app !</center></b><br>");
end
```

Anwendungen definieren im Gegensatz zur Verwaltung Komfortfunktionen für den Zugriff auf einzelne Elemente aus Tabellen z.B.:

```
function app.getEquipment(idx)
  if (idx > 0) and (idx <= #app.equipment) then
    return app.equipment[idx][1]
  else
    return "ERROR"
  end
end
```

Typischerweise liegen diese Funktionen in der Datei „env.lua“ die die Umgebung für die Anwendung bereitstellt und um entsprechende Funktionen ergänzt. Hier ein kleiner Auszug aus dem Programm für den 8“ Aligner. Verfahren wird hier nach dem gleichen Prinzip, Definition der Tabelle, Werte laden mit „requiere“, Komfortfunktion „localeGetCamInfo“ definieren.

```
---------------- Application module locale configuration ---------------

locale = {};

require "locale";

function locale.getCamInfo(idx)
  if (idx > 0) and (idx <= #locale.caminfo) then
    return locale.caminfo[idx][1]
  else
    if idx == 0 then
        return "OK"
    else
        return "ERROR "..idx
    end
  end
end
```

Die Anwendung in „run.lua“ liefert dann die Nachricht zum Status in der entsprechenden Sprache:

```
require "env"
...
caminfo.status = 7
...
showError("CAM "..locale.getCamInfo(tonumber(caminfo.status))..." !")
```

Problem

```
mic@fabpro:~$ ssh root@pi_hmi
Warning: the ECDSA host key for 'pi_hmi' differs from the key for the IP address
'172.27.121.62'
Offending key for IP in /home/mic/.ssh/known_hosts:24
Matching host key in /home/mic/.ssh/known_hosts:35
Are you sure you want to continue connecting (yes/no)? ^C
mic@fabpro:~$ ssh-keygen -R pi_hmi
# Host pi_hmi found: line 35 type ECDSA
/home/mic/.ssh/known_hosts updated.
Original contents retained as /home/mic/.ssh/known_hosts.old
mic@fabpro:~$ ssh-keygen -R 172.27.121.62
# Host 172.27.121.62 found: line 24 type RSA
/home/mic/.ssh/known_hosts updated.
Original contents retained as /home/mic/.ssh/known_hosts.old
mic@fabpro:~$ ssh root@pi_hmi
The authenticity of host 'pi_hmi (172.27.121.62)' can't be established.
ECDSA key fingerprint is 92:19:74:96:28:d7:39:da:be:2f:4a:bb:6c:21:80:f4.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'pi_hmi,172.27.121.62' (ECDSA) to the list of known
hosts.
# exit
Connection to pi_hmi closed.
mic@fabpro:~$ ssh root@pi_hmi
# exit
```