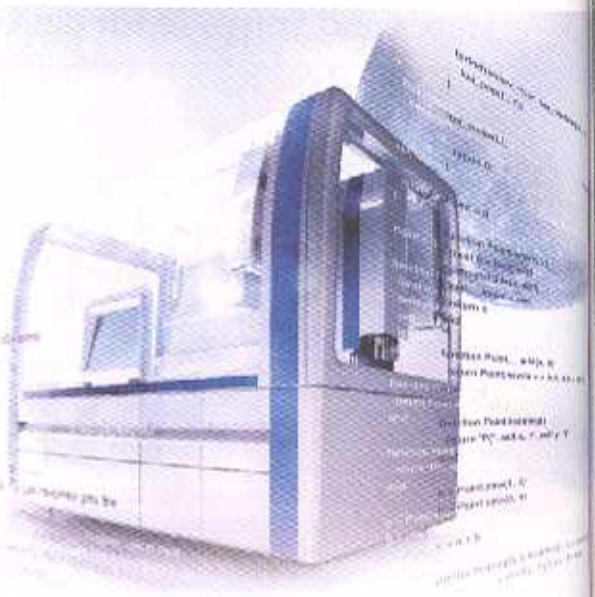
Sonne, Mond und Skripte

Die Skriptsprache Lua



Der Vorteil einer Skriptsprache ist, dass Inhalte von Variablen – ja sogar die gesamte Logik des Programms – jederzeit geändert werden können, da ja erst zur Laufzeit in Maschinenbefehle übersetzt wird. Außerdem entfällt während der Entwicklung der aufwendige Compile-Debug-Zyklus, was den Entwicklungsvorgang deutlich beschleunigt.

Von Dr. Claus Kühnel und Daniel Zwirner

Um den Entwicklungsvorgang komplexer mechatronischer Systeme zu beschleunigen, wird versucht, die Entwicklung von Hard- und Firmware und die der Applikations-Software zu parallelisieren. Ein Skript-Interpreter, mit dem beliebige Aktionen in den hardwarenahen Schichten ausgeführt werden können, kann hier sehr hilfreich sein. Allerdings ist die Entwicklung eines eigenen Interpreters sehr aufwendig – und auch überhaupt nicht nötig, denn es existieren bereits Lösungen wie z.B. die Skriptsprache Lua [4]. Die Erfahrungen, auf denen dieser Artikel beruht, entstanden im Zuge der Entwicklung eines Laborautomaten für die Medizintechnik (siehe **Kasten**). Lua kann für ein breites Feld von Embedded-Systemen, insbesondere im Bereich der Robotik, eingesetzt werden.

Lua – portugiesisch für Mond – ist eine Skriptsprache zum Einbinden in Programme, um diese leichter weiterentwickeln und warten zu können. Der Name Lua ist eine Anspielung darauf, dass die Sprache ein Nachfolger von „Sol" (Simple Object Language) ist, dem portugiesischen Wort für „Sonne". Eine der besonderen Eigenschaften von Lua ist die geringe Größe des compilierten Skript-Interpreters.

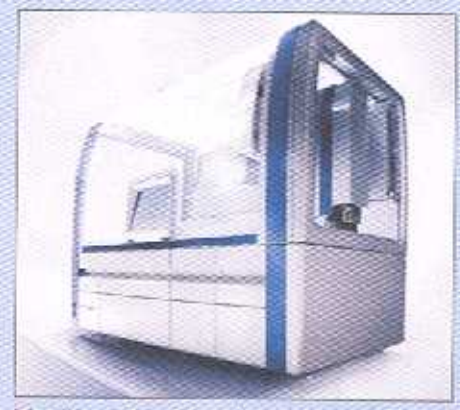
Lua wurde 1993 von der Computer Graphics Technology Group der Ponti-

Lua in Aktion: Geräteplattform QIAsymphony SP

QIAsymphony SP automatisiert die Aufreinigung von Proteinen, DNA oder RNA aus einer Vielzahl unterschiedlicher Probenmaterialien. Der Prozess basiert auf einer Weiterentwicklung der Qiagen Magnetic Particle-Chemie. Probenvolumina bis zu einem Milliliter und bis zu 96 Proben pro Lauf können bearbeitet werden. QIAsymphony automatisiert ganze Arbeitsabläufe von der Probe bis zum Ergebnis. Über Ethernet TCP/IP kann das Gerät in ein Firmen respektive Klinik-Netzwerk integriert werden.

Zur Abarbeitung des biochemischen Prozesses sind zahlreiche Sensoren, Aktoren und Steuerungsfunktionen nötig, die auf einem Steuerungsrechner mit grafischem User-Interface (Touchscreen) zusammengeführt werden. Über CAN/CANopen ist eine Reihe von kleinen Controllern abgesetzt, die die zahlreich vorhandenen Antriebe steuern und Sensoren bis hin zu 2D-Barcode-Kameras abfragen.

Der abzuarbeitende Prozess liegt als XML-File vor und wird in der Applikationsebene der Software abgearbeitet. In dieser Schicht befinden sich zahlreiche weitere Komponenten, die eine störungsfreie Abarbeitung der Proben sichern. Die Grundfunktionen der Software wurden in Funktionsmustern und Prototypen implementiert, bevor die Applikation und das GUI für Inbetriebnahme und Tests zur Verfügung standen. Dadurch entstand der Bedarf nach einem Tool, mit dem der Test der Grundfunktionen möglich wurde. Der Lua-Interpreter wurde versuchsweise in die Low-Level-Testsoftware integriert. Ein Nachmittag reichte, um Lua zu integrieren und den ersten Befehl zur Roboteransteuerung zu implementieren. Es folgten schnell Lua-Anbindungen für weitere Komponenten. Heute haben wir für alle Schichten unterhalb des User-Interfaces (GUI) spezifische Lua-Packages.



Im QIAsymphony SP werden zu Testzwecken alle Funktionen unterhalb der Benutzeroberfläche mit Lua-Skripts gesteuert. Der Prozessablauf wird als XML-Datei übergeben.

Danach verwendete das Testcenter Lua intensiv, um einzelne Module zu verifizieren. Es entstanden erste Lua-Skripts, um den Roboter zu kalibrieren. In der Qualitätskontrolle (QC) werden nun mit Hilfe von Lua-Skripts die fertig gebauten Roboter kalibriert und getestet.

```
a = 1.5 print(type(a), a)       --> number      1.5
b = "2"; print(type(b), b)      --> string      2
c = a + 0; print(type(c), c)    --> number      3.5
b = b .. "a"                    -> stdin:1: attempt to perform
                                   arithmetic on a string value
d = print; d(type(d))           -> function
```

Listing 1. Variablen werden in Lua automatisch erzeugt und sind nicht typgebunden. Der Typ wird automatisch erzeugt.

fikalen Katholischen Universität von Rio de Janeiro in Brasilien entwickelt. Lua-Programme werden vor der Ausführung in Bytecode übersetzt. Obwohl man mit Lua auch eigenständige Programme schreiben kann, ist Lua vorrangig als Skriptsprache von C-Programmen konzipiert. Der Lua-Interpreter kann über eine C-Bibliothek angesprochen werden, die auch ein API für die Laufzeitumgebung des Interpreters für Aufrufe vom C-Programm aus enthält. Mittels des Programmier-Interfaces können verschiedene Teile des Programms in C und Lua geschrieben werden, während Variablen und Funktionen in beiden Richtungen erreichbar bleiben, d.h., eine Funktion in Lua kann eine Funktion in C aufrufen und umgekehrt. Lua ist in ANSI-C implementiert und unterstützt sowohl funktionale als auch objektorientierte Programmierung.

Skripte – noch zur Laufzeit modifizierbar

Zu Beginn entdeckten die Spielehersteller die Vorzüge von Lua, weil sie damit künstliche Intelligenz in Skripte auslagern und diese noch zur Laufzeit modifizieren konnten. Aber auch in Anwendungen außerhalb dieses Bereichs, in denen Abläufe oder Konfigurationen durch Skripte gesteuert werden sollen, kam und kommt Lua immer häufiger zum Einsatz.

Da der Lua-Interpreter extrem schnell und hochgradig portabel ist und sich leicht in C-Programme einbetten lässt, ist er gerade für Embedded-Systeme eine attraktive Alternative zu anderen Skript-Interpretern. Obwohl er nur wenige Kilobyte umfasst, passt noch eine vollständige Garbage Collection hinein, die anfallenden Datenmüll automatisch aus dem Speicher wirft.

Lua ist freie Software und wurde bis zur Version 5 unter einer BSD-Lizenz, ab der heute aktuellen Version 5 unter der MIT-Lizenz veröffentlicht. Die MIT-Lizenz erlaubt den Einsatz auch in kommerziellen Produkten, ohne dass man den eigenen Quellcode veröffentlichen muss. Jede Kopie der Software oder substanzielle Teile davon müssen mit einem Copyright und einem Erlaubnishinweis ausgestattet werden. Hinweise hierzu sind den Links unter Open Source Initiative OSI zu entnehmen.

```
function pow(a, n)
  n = n or 2
  b = 1
  for i=1,n do b = b * a end
  return b
end

print(pow(2, 8))          --> 256
print(pow(2))             --> 4
print(pow(2, 3, 4))       --> 8
```

Listing 2. Funktionen können beliebig viele Parameter haben, sie können Variablen zugewiesen werden und sie können auch einer Funktion wieder als Parameter übergeben werden.

```
a = {}                    -- erzeugt eine leere Tabelle
                          und speichert die Referenz in 'a'
a["x"] = 11
print(a["x"])             --> 11
print(a.x)                --> 11

farbe = {"rot", "grün"}
print(farbe[1])           --> rot
for i=1, #farbe do
   print(farbe[i]) end    --> rot  grün

de_en = {};
   de_en["rot"] = "red";
   de_en["grün"] = "green"
for i=1, #farbe do
   print(de_en[farbe[i]]) end     --> red  green
for key, value in pairs(de_en) do
   print(key, value) end          --> grün  green  rot  red
```

Listing 3. Tabellen haben in Lua keine feste Größe. Wird eine Tabelle wie ein Array verwendet, kann man sie mit einer for-Schleife durchlaufen. Der Operator # liefert die Länge des Arrays zurück.

„Hello World" sagen mit Lua

Der Quellcode von Lua kann von der Lua-Homepage (www.lua.org/) bezogen werden. Unter Linux kann der Quellcode mit *make linux* übersetzt werden. Binäre Distributionen für Windows, Mac und Linux können von Lua-Binaries (http://luabinaries.luaforge.net/) heruntergeladen werden. Zum Kennenlernen von Lua reicht der mitgelieferte Standalone-Interpreter aus. Im Verzeichnis *src* des Lua-Quellcodes befindet sich nach dem Compilieren der Standalone-Interpreter lua. In einer Shell wird der Interpreter mit *./lua* gestartet. Jetzt kann an der Lua-Eingabeaufforderung der Befehl *print("Hello World")* eingegeben und mit „Enter" bestätigt werden. Die Funktion *print()* gibt einfach den String "Hello World" über die Standardausgabe aus.

Variablen und Datentypen

Variablen müssen in Lua nicht deklariert werden. Sie werden bei Bedarf automatisch erzeugt. Die Variablen sind nicht typgebunden. Der Typ ist implizit vom Wert, den sie vertreten, definiert. Ändert sich der Wert einer Variablen, passt sich der Typ automatisch an. Alle Variablen sind global, außer sie werden als lokal definiert. Lua kennt die Datentypen String (Zeichenketten), Zahl (Gleitkommazahl doppelter Genauigkeit), Boolean (wahr oder falsch), Nil (die Variable hat kei-

nen Wert), Funktion und Tabelle. Uninitialisierte Variable haben den Wert Nil.

In der ersten Zeile von **Listing 1** wird der Variablen a die Zahl 1,5 zugewiesen, was den Typ *number* zur Folge hat. Danach werden mit der Funktion print() der Typ und der Wert der Variablen a ausgegeben. Aufeinanderfolgende Anweisungen müssen nicht mit einem Separator getrennt werden. Lua erlaubt aber die Verwendung des Semikolons am Ende einer Anweisung zur besseren Lesbarkeit.

In der zweiten Zeile wird der Variablen b der String „2" zugewiesen und der Typ ist folglich *string*. In der dritten Zeile sollen nun die numerische Variable a und die String-Variable b addiert werden. Lua versucht den String in eine Zahl zu konvertieren, um die arithmetische Operation auszuführen. Das Ergebnis ist vom Typ *number*. Kann die String-Variable nicht konvertiert werden wie in Zeile 4, dann erfolgt eine Fehlermeldung. Diese automatische Typkonvertierung wird Coercion genannt. In der letzten Zeile wird der Variablen a eine Referenz auf die Funktion print zugewiesen, danach wird die von a referenzierte Funktion mit dem Parameter type(a) aufgerufen.

Funktionen

Funktionen (**Listing 2**) können beliebig viele Parameter haben, sie können Variablen zugewiesen werden und sie können auch einer Funktion wieder als Parameter übergeben werden. Wird in Listing 2 der zweite Parameter der Funktion pow() nicht übergeben, dann ist *n == nil*. Der Ausdruck *n = n or 2* weist n den Wert 2 zu, wenn kein Wert übergeben wurde. Werden einer Funktion zu viele Parameter übergeben, dann werden diese ignoriert. Eine interessante Eigenschaft von Lua ist, dass Funktionen mehrere Rückgabewerte haben können.

Kontrollstrukturen

Lua kennt die üblichen Kontrollstrukturen – *if* für bedingte Ausführung sowie *while*, *repeat* und *for* für Schleifen. Zusätzlich kennt Lua noch ein generisches *for*, mit dem alle Werte einer Iteratorfunktion durchlaufen werden.

Tabellen und assoziative Arrays

Tabellen in Lua sind assoziative Arrays. Ein assoziatives Array ist ein Array, das als Index nicht nur Zahlen, sondern auch Strings oder alle anderen Datentypen (außer *nil*) verwendet. Die Tabellen haben keine feste Größe. Man kann zur Laufzeit beliebig viele Elemente hinzufügen. Einmal erzeugte Tabellen müssen nicht explizit freigegeben werden. Der in Lua eingebaute Garbage Collector löscht automatisch alle nicht mehr benötigen Variablen und Tabellen.

Lua realisiert Records oder Structs, indem man den Index als Feldname verwendet. Der Ausdruck a["x"] ist identisch mit a.x. (**Listing 3**). Wird eine Tabelle wie ein Array verwendet, kann man sie mit einer for-Schleife durchlaufen. Dies zeigt Listing 3 am Beispiel des Arrays *farbe*. Es ist mit den Zeichenketten "rot" und "grün" initialisiert. Der erste Index ist 1. Das Array farbe kann mit einer *for*-Schleife durchlaufen werden. Der Operator # liefert (ab Lua 5.1) die Länge des Arrays zurück. Die Variable *de_en* initialisiert eine Tabelle mit der englischen Über-

```
Point = {}

function Point:new(x, y)
  local o = {x=x, y=y}
  setmetatable(o, self)
  self.__index = self
  return o
end

function Point.__add(a, b)
  return Point:new(a.x + b.x, a.y + b.y)
end

function Point:tostring()
  return "P("..self.x.."/"..self.y..")"
end

a = Point:new(1, 2)
b = Point:new(3, 4)

c = a + b

print(a:tostring(), b:tostring(), c:tostring())
  --> P(2/2)   P(3/4)   P(4/6)
```

Listing 4. Mit Lua kann man Klassen, Datenkapselung, Vererbung und Mehrfachvererbung realisieren. Teilweise ist die Umsetzung aber eher etwas umständlich.

setzung der Tabelle *farbe*. Die Tabelle *farbe[i]* liefert den Text an der i-ten Position zurück (z.B. rot) und übergibt diesen Wert der Tabelle de_en als Index, diese gibt dann die englische Übersetzung der Farbe zurück. Mit einer normalen for-Schleife kann man ein Array nur dann durchlaufen, wenn die Indizes numerisch und fortlaufend sind und keine „Löcher" haben. Möchte man die Tabelle de_en durchlaufen, kann man einen Iterator verwenden – siehe die letzte *for*-Schleife in Listing 3. Die Funktion pairs(...) liefert für jeden Aufruf ein Index/Wert-Paar zurück. Sind keine Werte mehr vorhanden, wird nil zurückgegeben. Die Reihenfolge, in der pairs(...) die Index/Wert-Paare zurückgibt, ist nicht spezifiziert.

Objektorientiertes Programmieren mit Lua

Lua ist eigentlich keine objektorientierte Programmiersprache, aber mit Hilfe der Tabellen kann man Klassen nachbilden. Das Beispiel in **Listing 4** zeigt, dass mit *Point = {}* eine Tabelle erzeugt wird, die die Klasse *Point* darstellen soll. Die drei Funktionen *new()*, *__add()* und *tostring()* werden der Tabelle *Point* zugewiesen, wobei der jeweilige Index gleich dem Funktionsnamen ist.

Der „:" bei *Point:new(x, y)* und *Point:tostring()* bewirkt, dass die Funktion einen versteckten Parameter *self* (entspricht *this* in C++) hat. Die Funktion *new()* erzeugt eine neue Tabelle mit den Feldern *x* und *y* und gibt diese Tabelle zurück. Der neuen Tabelle wird die Tabelle *Point* als Metatabelle hinterlegt. Wenn die beiden Punkte *a* und *b* addiert werden, schaut der Interpreter in der Metatabelle nach, ob eine Funktion *__add()* vorhanden ist, und ruft diese dann auf. Auf diese Weise kann der Operator „+" überladen werden.

Lua-Bibliotheken

Die Lua-Standard-Bibliotheken bieten sehr nützliche Funktionen, die über das C-Programmier-Interface implementiert sind und als separate C-Module zur Verfügung stehen. Obwohl Erweiterungen in Lua selbst implementiert werden könnten, bietet eine Implementierung in C verschiedentlich Performance-Vorteile. Gegenwärtig bietet Lua Standard-Bibliotheken für:

- string: Reguläre Ausdrücke und Funktionen zur Stringmanipulation,
- table: Tabellen sortieren und Elemente einfügen und löschen,
- math: Mathematische Funktionen,
- io: Funktionen zum Lesen und Schreiben von Dateien,
- os: Funktionen zur Berechnung von Datum und Zeit sowie zum Ausführen von Befehlen des Betriebssystems.

Lua und C verbinden

Lua wurde vom Anfang an entworfen, um erweitert zu werden. Daher ist es sehr einfach, den Leistungsumfang von Lua zu erweitern. Die Sprache bietet dafür ein universelles C-Interface an. Mit der Präprozessor-Anweisung *#include "luaxlib.h"* werden dem Compiler alle Lua-Definitionen bekannt gemacht. Alle Funktionen, die in C implementiert werden, haben als Parameter einen Zeiger auf den aktuellen Lua-Interpreter. Alle Variablen, die einer Lua-Funktion übergeben werden, werden auf den Lua-Stack kopiert.

Listing 5 zeigt einige Quelltext-Zeilen, aus denen eine Lua-Erweiterung besteht. Die Funktion *lua_get-*

```
00  static int mylib_sum(lua_State *L){
01    double s = 0;
02    int i;
03    for(i=1; i<=lua_gettop(L); i++)
04      s = s + luaL_checknumber(L, i);
05
06    lua_pushnumber(L, s);
07    return 1;
08  }
09
10  static const struct luaL_reg mylib_functions[] = {
11    { "sum",      mylib_sum },
12    { NULL,NULL } };
13
14  int luaopen_mylib (lua_State *L){
15    luaL_openlib(L,"mylib", mylib_functions,0);
16    return 1;
17  }
18  print(mylib.sum(1, 2.2, "3"))     --> 6.2
```

Listing 5. Lua lässt sich einfach mit C erweitern. Alle Variablen, die einer Lua-Funktion übergeben werden, werden auf den Lua-Stack kopiert.

top() liefert die Anzahl der Werte auf dem Stack zurück. Mit der Funktion *luaL_checknumber(L, i)* kann auf das i-te Element des Lua-Stacks zugegriffen werden. Gleichzeitig überprüft die Funktion *luaL_checknumber()*, ob der Wert auf dem Stack eine Zahl ist bzw. in eine Zahl konvertiert werden kann. Das Resultat der Berechnung wird mit der Funktion *lua_pushnumber()* wieder auf den Stack kopiert. Diese Funktion hat als Returnwert die Anzahl Elemente, die auf den Stack kopiert wurden (hier also 1).

Alle Funktionen einer neuen Lua-Bibliothek werden in einer Liste mit Funktionsnamen und dazugehöriger C-Funktion abgelegt. Die Liste wird mit einem NULL-Eintrag beendet (Zeile 12 in Listing 5). Jede Bibliothek muss eine Funktion exportieren, die *luaopen_NAME()* heißt. In dieser Funktion wird die Liste mit den neuen Funktionen beim Lua-Interpreter registriert.

Die neue Bibliothek wird unter Linux mit *gcc -I~/lua-5.1.3/ mylib.c -o mylib.so -shared* in eine Programmbibliothek compiliert. Beim Start des Lua-Interpreters ist die neue Bibliothek dem Interpreter noch nicht bekannt. Die Bibliothek muss zuerst mit dem Befehl *require("mylib")* geladen werden. Jetzt kann die neue Bibliothek verwendet werden: Im Argument der Funktion *sum()* sind alle Typen enthalten, die Lua als Zahl interpretiert oder in eine Zahl konvertiert.

Lua-Interpreter in eine C-Anwendung integrieren

Um Lua kennenzulernen, ist der mitgelieferte Standalone-Interpreter sehr hilfreich, aber Lua wurde entwickelt, um in ein Anwendungsprogramm integriert zu werden. Wie der letzte Abschnitt gezeigt hat, kann der Lua-Interpreter so erweitert werden, dass z.B. ein Roboter damit gesteuert werden kann. Lua kann aber auch in ein Anwendungsprogramm integriert werden, um z.B. Benutzereingaben auszuwerten.

Das Programmbeispiel in **Listing 6** zeigt eine Taschenrechner-Anwendung auf der Befehlszeile. Benötigt werden nur die Bibliothek *math* und die eigene Bibliothek *mylib*. Das Programm liest alle über die Standardeingabe empfangenen Zeichen bis zum Zeilenende und kopiert diese in den Buffer *line*. Damit aus der Eingabe ein gültiger Lua-Befehl wird, wird noch ein „ *__res=*" vorangestellt.

Dem Lua-Interpreter wird die Zeichenkette *line* übergeben. Der Interpreter wertet den Ausdruck aus und kopiert das Resultat in die Lua-Variable *__res*. Die Funktion *lua_getglobal()* sucht im Lua-Interpreter die globale Variable *__res* und kopiert diese auf den Stack. Mit der Funktion *lua_tonumber(L, -1)* wird der oberste Wert im Stack der Funktion *printf()* übergeben. Der zuletzt auf den Stack kopierte Wert wird mit *lua_pop(L, 1)* vom Stack entfernt. Schließlich wird der Lua-Interpreter beendet und der Speicher freigegeben.

Das Taschenrechner-Programm wird unter Linux mit *gcc mylib.c mylua.c ~/lua-5.1.3/src/liblua.a -o mylua -lm* compiliert. Das Programm berechnet z.B. den Ausdruck folgendermaßen:

```
echo "(2+3) / 2" | ./mylua --> 2.5
echo "math.sqrt(mylib.sum(4 * 2,
      math.pow(3, 2)))" | ./mylua --> 5
```

Mit dem Befehl *echo* wird das übergebene Argument an die Taschenrechner-Anwendung *mylua* gesendet, die dann die ge-

```
#include <string.h>
#include <lauxlib.h>
#include <lualib.h>
#include "mylib.h"

int main (void)
{
  char line[1024];
  strcpy(line, " res=");
  fgets(line + strlen(line), sizeof(line),
  stdin);  // reads characters from stdin

// neuen Lua Interpreter starten
  lua_State *l = lua_open();
  luaopen_math(l);
  luaopen_mylib(l);

// Lua-Interpreter liest Zeile ein
  if (luaL_dostring(l, line) == 0)
  {
    //  res auf Stack kopieren
    lua_getglobal(l, "res");
    // TopOfStack wird mit printf ausgegeben
    fprintf(stdout, "%g\n", lua_tonumber(l, -1));
    lua_pop(l, 1);  // TopOfStack löschen
  }
else
{
  fprintf(stderr, "%s", lua_tostring(l, 1));
    lua_pop(l, 1);
}

lua_close(l); // Lua-Interpreter schließen

  return 0;
}
```

Listing 6. Hier wird der Lua-Interpreter in ein C-Programm eingebunden, um einen Befehlszeilen-Taschenrechner zu implementieren.

wünschten Berechnungen durchführen und das Ergebnis über stdout ausgeben kann.

Die hier vorgestellten Programmbeispiele sind nur rudimentäre Auszüge dessen, was mit Lua möglich ist. Die Dokumentation, die von der Homepage des Lua-Projekts www.lua.org erreichbar ist, gibt einen ausführlichen Einblick in die Leistungsfähigkeit der Skriptsprache Lua. Im vorliegenden Projekt QIAsymphony SP lagen die Vorteile in der Parallelisierbarkeit der Hardware- und Firmware-Entwicklung. Mittels der interpretierten Skriptsprache können Variablenwerte und Programmablauf jederzeit vor Ort geändert werden – ein Vorgehen, das angesichts ansonsten komplizierter Cross-Entwicklungszyklen sehr einfach zu bewerkstelligen ist. *jk*

Literatur

[1] Lua-Homepage: www.lua.org

[2] *Ierusalimschy, R.:* Programmieren mit Lua. Open Source Press, 2006.

[3] *Ierusalimschy, R.; De Figueiredo, L.H.; Celes, W.:* Lua 5.1 Reference Manual. Erschienen im Eigenverlag der Autoren, Rio de Janeiro 2006.

[4] *Streicher, M.:* Embeddable scripting with Lua. Lua offers high-level abstraction without losing touch with the hardware. www-128.ibm.com/developerworks/linux/library/l-lua.html

Dr.-Ing. Claus Kühnel

studierte und promovierte an der Technischen Universität Dresden auf dem Gebiet der Informationselektronik und bildete sich später in Biomedizintechnik weiter. Seit 2004 ist er bei der Qiagen Instruments AG in Hombrechtikon (CH) als Associate Director Electronic Engineering für die Entwicklung von Elektronik-Hardware und hardwarenaher Software verantwortlich.

Claus.Kuehnel@qiagen.com

Dipl.-Ing. HTL Daniel Zwirner

studierte an der Hochschule für Technik Rapperswil Elektrotechnik und bildete sich mit einem Nachdiplom-Studium in Software Engineering weiter. Seit 2005 ist er bei der Qiagen Instruments AG in Hombrechtikon (CH) als Software-Ingenieur beschäftigt und dort an der Entwicklung einer neuen Geräteplattform für die molekulare Diagnostik beteiligt. Er arbeitet im Bereich der Betriebssystem- und Gerätesteuerungsentwicklung.