Heimautomatisierung für Hardwarebastler (Teil 3)

von Hans Müller

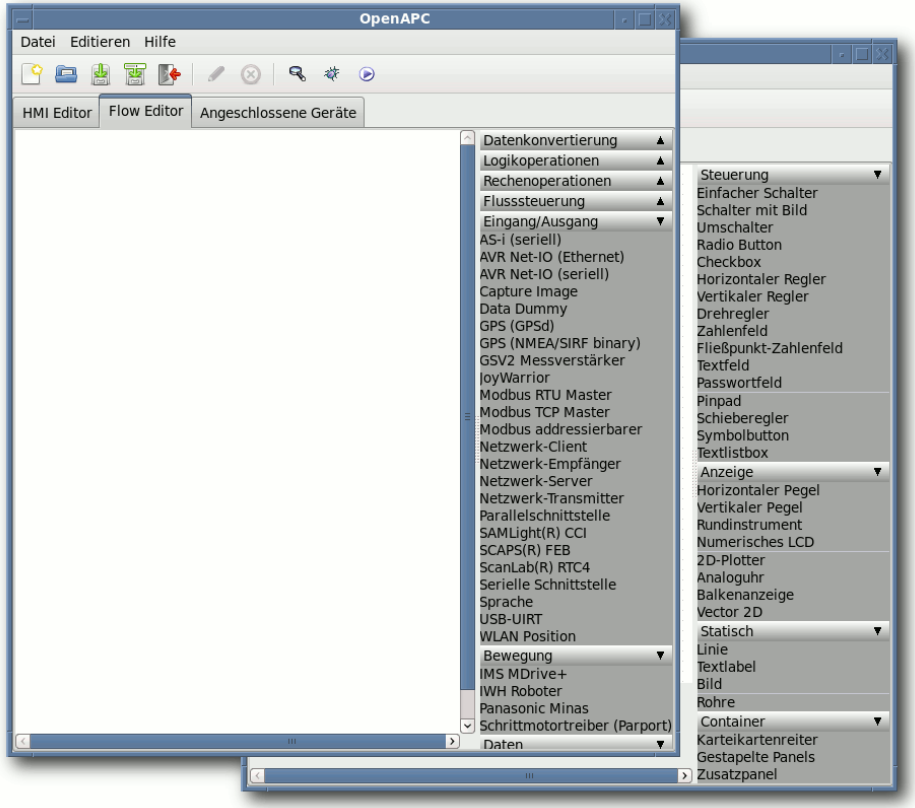
Eigene Hardware von einer möglichst einfach und leicht zu benutzenden Softwarelösung ansteuern zu lassen, ist eine feine Sache. Allerdings wird das in dem Moment schwierig, in dem exotische oder gar komplett selbst entwickelte Hardware zum Einsatz kommt, welche von der gewählten Prozesssteuerungssoftware nicht direkt unterstützt wird. Dann entscheidet sich, ob die bis dahin verwendete Visualisierungslösung so flexibel und erweiterbar ist, dass sie dennoch weiter benutzt werden kann, oder ob es sich um eine Sackgasse handelt, aus der man ohne einen kompletten Systemwechsel nicht mehr heraus kommt.

In den ersten beiden Teilen dieser Artikelserie, welche in den freiesMagazin-Ausgaben 01/2011 [1] und 03/2011 [2] zu finden sind, wurde gezeigt, wie es mittels der OpenAPC-Software möglich ist, kleine und auch größere Projekte samt des zugehörigen HMI zu erstellen. Die dort exemplarisch vorgestellte Rollladensteuerung hat mit der Außenwelt über die Parallelschnittstelle kommuniziert. Nachdem der Parallelport ein mittlerweile ziemlich veraltetes Stück Hardware ist, welches – wenn überhaupt – nur noch als Pfostenleiste auf dem Mainboard zu finden ist, liegt der Wunsch nahe, hier etwas anderes einzusetzen.

Alternativen gibt es viele, so kann die Kommunikation mit der Außenwelt per USB, per PCI-IO-Karte oder vielleicht sogar per Controller an der seriellen Schnittstelle erfolgen.

Egal welche Lösung bevorzugt wird, es ergibt sich das Problem, dass die OpenAPC-Software diese neue Hardwareschnittstelle möglicherweise nicht von Haus aus unterstützt. Dank des Plug-in-Konzeptes ist es aber leicht möglich, solche neuen Devices einzubinden und damit über die OpenAPC-Steuerungssoftware nutzbar zu machen.

Ganz in der Tradition der ersten beiden Teile soll auch hier nun wieder mit dem Software-Part, sprich der Implementierung eines neuen Plug-ins, der Teil eines Projektes im Vordergrund stehen, der für den Hardwareentwickler nicht nur ein fremdes Fachgebiet sondern meist auch ein ungeliebtes aber notwendiges Übel ist.



Leichter Zugriff auf HMI-und Flow-Elemente per Foldbar – Plug-ins inklusive.

Hinweis: Alle vorgestellten Dateien können über das Archiv Heimautomatisierung.zip heruntergeladen werden

Baukastenprinzip

Plug-ins sind ein elementarer Bestandteil der OpenAPC-Visualisierungssoftware. Innerhalb der Applikation ist das nicht so ohne weiteres zu erkennen, aber sehr viele Funktionalitäten sind bereits in externen Plug-ins implementiert und gar nicht Teil des Hauptprogrammes.

Ein Vorteil, der sich daraus ergibt, ist die Möglichkeit, auf dem Zielsystem nur die

Plug-ins zu installieren, welche auch wirklich benötigt werden. Das spart Platz und beschränkt eine solche Installation auf das absolut notwendige Minimum.

Doch zuerst ein Blick in die neue Version 1.5: Auffälligste Änderung sind Foldbars jeweils am rechten Rand des Fensters, in der die vorhandenen Funktionen und Elemente verfügbar sind und in denen man auch leicht erkennen kann, welche dieser Elemente als externe Plug-ins realisiert sind. Sowohl im HMI- als auch im Flow-Editor sind all diejenigen Objekte, welche sich in den Foldbars unterhalb der kleinen, horizontalen Trennlinie befinden, als externe Softwarekomponenten realisiert. In den Flowelementgruppen „Daten“, „Bewegung“, „Eingang/Ausgang“ sind das sogar alle. Hier existieren gar keine Funktionen, welche vom Hauptprogramm ausgeführt werden.

An dieser Stelle ist bereits zu erahnen, dass zwischen den einzelnen Typen von Flow-Plug-ins kein großer Unterschied besteht. Die Schnittstelle nach oben hin ist für alle gleich. Es ist nur eine Frage der Implementierung, was diese genau tun.

Aus diesem Grund soll hier in einem ersten Schritt ein einfaches, allgemein nutzbares Plug-in entwickelt werden, welches gar nicht zur Kategorie „Eingang/Ausgang“ gehört, sondern in der Kategorie „Daten“ Zufallszahlen erzeugt. Dies ist eine Funktionalität, welche dem OpenAPC-Paket noch fehlt. Das fertige Plug-in wird dem Projekt anschließend zur Verfügung gestellt. Die Weiterentwicklung auf ein Plug-in, welches dann tatsächlich irgendeine sehr spezielle Hardware ansteuert (und damit zu einer anderen Kategorie gehört), bedarf dann nur noch weniger Änderungen.

Die Werkzeuge

Um mit der Entwicklung eines eigenen Plug-ins zu beginnen, werden einige Tools und Pakete benötigt. Das wären zum einen gcc und make, welche Teil jeder Linux-Distribution sind und im Zweifelsfall leicht nachinstalliert werden können. Zum anderen ist das das OpenAPC-Entwicklungspaket. Dieses findet sich z. B. auf der OpenAPC-Downloadseite [3] im SDK [4]. Alternativ kann der aktuelle Stand aller Sourcen und Header aber auch per git von der Projektseite auf fedorahosted.org beschafft werden. Ein Kommandozeilenaufruf

```
$ git clone http://git.fedorahosted.org/git/OpenAPC.git
```

erzeugt dabei eine lokale Kopie des Sourcenarchives inklusive der benötigten Headerfiles. Kleiner Nachteil hier: Das Handbuch, welches alle Programmierschnittstellen beschreibt, ist dort nicht enthalten. Dieses ist nur im SDK zu finden.

In der mittels **git clone** erzeugten Verzeichnisstruktur finden sich im Unterverzeichnis **plugins/** gleich eine ganze Sammlung von Sourcen, welche innerhalb des OpenAPC-Paketes diverse Plug-ins realisieren. Das ist auch ein guter Platz, um ein eigenes Verzeichnis **libio_random/** anzulegen und ein Makefile zu erzeugen. Dabei ist es das einfachste, sich eines der existierenden Makefiles der anderen Plug-ins zu bedienen und lediglich die Namen der Sourcedateien anzupassen.

Zwei Eigenheiten dieser Makefiles sind jedoch zu beachten: Zum einen wird ein Verzeichnis **../../flowplugins** erwartet, in dem das Plug-in erzeugt wird, welches aber nicht existiert und deswegen von Hand mittels

```
$ mkdir ../../flowplugins
```

angelegt werden muss. Und zum anderen wird als letzter Schritt im Makefile ein

```
$ sudo cp $(EXECUTABLE) /usr/lib/openapc/flowplugins/
```

aufgerufen, was dazu führt, dass das eben generierte Plug-in in das **flowplugins**-Verzeichnis der aktuellen OpenAPC-Installation kopiert wird, sodass man es nach einem Neustart der Applikation sofort testen kann. Damit dass klappt, muss sudo [5] allerdings so eingerichtet sein, dass der lokale Benutzer auch **cp** mit root-Rechten ausführen darf. Ein Eintrag

```
username ALL = NOPASSWD: /bin/cp
```

in der Datei **/etc/sudoers** sorgt dafür, dass das auch klappt. Hier muss **username** natürlich durch den Namen des Benutzers ersetzt werden, welcher **cp** per **sudo** ausführen können soll.

Die Basis

Prinzipiell wird ein Plug-in immer in zwei verschiedenen Umgebungen ausgeführt. Das ist zum einen der Editor, in welchem hauptsächlich Konfigurationsfunktionalitäten gefragt sind, und zum anderen der Player,in welchem das Plug-in dann seine eigentliche Arbeit verrichtet. Entsprechend dieser unterschiedlichen Umgebungen müssen vom Plug-in auch unterschiedliche Funktionen bereit gestellt werden.

Technisch gesehen ist so ein Plug-in nichts weiter als eine Shared Library, welche unter Linux durch die Dateiendung **.so** gekennzeichnet wird. Da diese Libraries aber nicht beim Programmstart gelinkt sondern zur Laufzeit dynamisch geladen werden, kommt eine Besonderheit zum Tragen: In so einem Fall muss die Library ihre internen Daten selbst trennen, da sie bei mehrfacher Verwendung nicht mehrfach private Speicherbereiche zugewiesen bekommt. Praktisch heißt das, dass die OpenAPC-Software zuerst immer eine Funktion **oapc_create_instance2()** aufruft, in welcher ein Datenbereich angelegt werden muss, der die aktuelle Instanz dieses Plugins ausschließlich verwenden darf. Eine andere Funktion **oapc_delete_instance()** kann verwendet werden, um diesen Speicherbereich wieder freizugeben.

Das klingt jetzt komplizierter als es ist. Praktisch sind das nur wenige Zeilen Code. Um diesen Datenbereich anzulegen, muss zuerst definiert werden, was das Plug-in tun soll. In diesem Beispiel soll es zwei Digitaleingänge besitzen, welche immer dann auf dem gegenüberliegenden Ausgang einen Zufallswert ausgeben, wenn ein HIGH-Signal gesetzt wurde. Die Ausgänge bestehen dementsprechend aus einem Digital- und einem numerischen Ausgang. Der Wertebereich des numerischen Ausganges soll konfigurierbar sein. Damit ist die Datenstruktur klar, welche beim Erzeugen einer Instanz angelegt werden muss:

```
struct libio_config
{
  unsigned short version,length;
  int            numRange;
};

struct instData
{
  struct libio_config config;
  int                 m_callbackID;
  unsigned char       m_digi;
  double              m_num;
};
```

Die erste Struktur **libio_config** enthält die Konfigurationsdaten, welche auch geladen und gespeichert werden sollen, weswegen sie in eine eigene Struktur ausgelagert wurden. **version** und **length** werden hier eingesetzt, um spätere kompatible Erweiterungen der Funktionalität zu ermöglichen, **numRange** speichert den maximal zulässigen Wertebereich des Zufallswertes.

Diese Struktur ist Teil von **instData**, der eigentlichen Instanzdatenstruktur. Hier kommen noch einige, nur während der Laufzeit benutzte Member hinzu, welche später näher erklärt werden. Die Erzeugung einer solchen Plug-in-Instanz ist damit denkbar simpel:

```
OAPC_EXT_API void* ↷
oapc_create_instance2(unsigned long↷
 /*flags*/)
{
   struct instData *data;

   data=(struct instData*)malloc(↷
   sizeof(struct instData));
   if (!data)
     return NULL;
   memset(data,0,sizeof(struct ↷
   instData));
   data->config.numRange=500;

   return data;
}
```

Es wird ein Speicherbereich alloziert und mit 0 initialisiert, in welchem die oben definierte Struktur passt. Der einzige Parameter **numRange** wird auf seinen Vorgabewert gesetzt. Dieser Speicherbereich ist gleichzeitig der Rückgabewert der Funktion. Geht während der Initialisierung irgend etwas schief, so muss **NULL** zurück gegeben werden.

Noch simpler geht es beim Freigeben der Instanzdaten zu:

```
OAPC_EXT_API void ↷
oapc_delete_instance(void* ↷
instanceData)
```

```
{
   if (instanceData) free(↷
   instanceData);
}
```

Damit ist die gesamte Funktionalität im Zusammenhang mit den Plug-in-Instanzen bereits bereitgestellt. Bei allen folgenden Aufrufen wird der oben angelegte Speicherbereich immer mitgeliefert. Wenn das Plug-in dann ausschließlich diese Daten verwendet, ist es egal, wie oft es innerhalb eines Projektes verwendet wird. Es arbeitet automatisch immer mit den richtigen Daten.

Funktionen für den Editor

Innerhalb des Editors muss ein Plug-in folgende Aufgaben erfüllen:

- mitteilen, was es kann, wie es angesprochen werden soll und welche Ein-/ Ausgänge es besitzt;
- Daten zur Verfügung stellen, welche das Layout des Konfigurationsdialoges festlegen;
- die vom Konfigurationsdialog zurückgegebenen Daten empfangen;
- aus einem Projektfile geladene Daten intern verarbeiten;
- die intern gehaltenen Daten so zur Verfügung stellen, dass sie in einem APCP-Projektfile abgespeichert werden können.

Für all diese Aufgaben werden wieder Funktionen erwartet, welche vom Plug-in bereit gestellt werden müssen.

```
static char   libname[]="Random Generator";

OAPC_EXT_API char *oapc_get_name(void)
{
   return libname;
}
```

Diese Funktion teilt dem Hauptprogramm mit, wie das Plug-in heißt. Dieser Name wird dann überall dort angezeigt, wo das Plug-in verwendet wird. Wichtig hier: der Name ist in einer globalen Variablen **libname** gespeichert, sodass sichergestellt ist, dass diese Daten auch nach Verlassen der Funktion noch gültig sind.

```
OAPC_EXT_API unsigned long oapc_get_capabilities(void)
{
   return OAPC_HAS_INPUTS|OAPC_HAS_OUTPUTS|
          OAPC_HAS_XML_CONFIGURATION|OAPC_ACCEPTS_PLAIN_CONFIGURATION|
          OAPC_ACCEPTS_IO_CALLBACK|
          OAPC_FLOWCAT_DATA;
}
```

Diese Funktion liefert verschiedene Flags zurück, welche festlegen, was das Plug-in kann und welche Features es besitzt. Die ersten beiden Konstanten **OAPC_HAS_INPUTS** und **OAPC_HAS_OUTPUTS** legen fest, dass es sowohl Ein- als auch Ausgänge hat. Von diesen hängt dann ab, ob die im Folgenden beschriebenen zwei Funktionen vorhanden sein müssen oder nicht. Das Flag **OAPC_HAS_XML_CONFIGURATION** ist ein wenig seltsam, zumindest in der aktuellen Softwareversion muss dieses eigentlich immer vorhanden sein. Es legt fest, dass die Konfiguration per XML-Struktur übermittelt wird, Alternativen dazu existieren derzeit keine. Ähnlich scheint es sich mit **OAPC_ACCEPTS_PLAIN_CONFIGURATION** zu verhalten. Hierüber wird festgelegt, wie die im Konfigurationsdialog eingegebenen Daten zurückgeliefert werden. Auch hier ist derzeit nur diese eine Variante vorgesehen. Anders die nächste Konstante **OAPC_ACCEPTS_IO_CALLBACK**. Ist diese gesetzt, werden die Ausgänge des Plug-ins nicht zyklisch abgefragt, vielmehr kann das Plug-in dem OpenPlayer per Callback-Funktion mitteilen, dass neue Daten vorhanden sind, welche von den Ausgängen abgeholt werden können. Dieses Flag ist somit für die Verwendung des Plug-ins in Player oder Debugger relevant.

Die letzte Konstante legt fest, innerhalb welcher Kategorie das Plug-in im Editor aufgelistet werden soll. Hiermit wurde dem Zufallszahlen-Plug-in die Kategorie „Daten“ zugeordnet. Andere Varianten existieren mit **OAPC_FLOWCAT_CONVERSION** für „Datenkonvertierung“, **OAPC_FLOWCAT_LOGIC** für „Logikoperationen“, **OAPC_FLOWCAT_CALC** für

„Rechenoperationen“, **OAPC_FLOWCAT_FLOW** für „Flusssteuerung“, **OAPC_FLOWCAT_IO** für „Eingang/Ausgang“ und **OAPC_FLOWCAT_MOTION** für die Flow-Elemente-Kategorie „Bewegung“.

Mit den nächsten beiden Funktionen wird der Applikation mitgeteilt, dass die Digitaleingänge 0 und 1 sowie die Ausgänge 0 (digital) und 1 (numerisch) verwendet werden sollen:

```
OAPC_EXT_API unsigned long ↷
oapc_get_input_flags(void)
{
   return OAPC_DIGI_IO0|↷
   OAPC_DIGI_IO1;
}

OAPC_EXT_API unsigned long ↷
oapc_get_output_flags(void)
{
   return OAPC_DIGI_IO0|↷
   OAPC_NUM_IO1;
}
```

Hier ist darauf zu achten, dass jeder Ausgang nur einmal mit einem entsprechenden Flag für einen Datentyp belegt wird. Eine Angabe wie z. B. **OAPC_CHAR_IO7|OAPC_BIN_IO7** wäre unzulässig, da ein Ein- bzw. Ausgang nur exakt einen Datentyp unterstützen und nicht gleichzeitig für Text- und Binärdaten benutzt werden kann.

Die nächste Funktion – bzw. die Daten, welche von ihr erzeugt und zurückgegeben werden – haben es in sich. Hier wird der Applikation mitgeteilt, welches Symbol zur Verwendung im Flow-Editor benutzt werden soll, wie das Layout des Konfigurationsdialoges aussehen soll und welche Parameter mit welchen Wertebereichen dort angezeigt werden sollen:

```
OAPC_EXT_API char *↷
oapc_get_config_data(void* ↷
instanceData)
{
   struct instData *data;

   data=(struct instData*)↷
   instanceData;
   sprintf(xmldescr,xmltempl,
     flowImage,
     data->config.numRange);
   return xmldescr;
}
```

Wie zu sehen ist, passiert hier nicht all zu viel. Die Informationen selbst stecken in einer XML-Struktur, welche an dieser Stelle zusammengesetzt und zurückgegeben wird. Diese besteht im Beispiel aus drei Teilen: der Definition des Bildes für den Floweditor, der Definition des Panels für die Parameter sowie der Definition des Hilfe-Panels, in welchem die Ein- und Ausgänge sowie deren Funktion erklärt werden:

```
<?xml version="1.0" encoding="UTF↷
-8" standalone="yes"?>
<oapc-config>
 <flowimage>%s</flowimage>
 <dialogue>
  <general>
   <param>
    <name>numrange</name>
    <text>Numeric Range</text>
    <type>integer</type>
    <default>%d</default>
    <min>2</min>
    <max>10000</max>
   </param>
  </general>
  <helppanel>
   <in0>CLK - generate random ↷
   digital value</in0>
   <in1>CLK - generate random ↷
   numeric value</in1>
   <out0>RND - random digital value↷
   </out0>
   <out1>RND - random numeric value↷
   </out1>
  </helppanel>
 </dialogue>
</oapc-config>
```

An Stelle des Platzhalters **%s** im XML-Tag **<flowimage />** wird das Bild eingefügt. Hier wird ein Base64-Encodiertes PNG-Bild in der Größe 106x50 Pixel erwartet. PNG-Bilder lassen sich mit quasi jedem Zeichenprogramm erzeugen, die Base64-Codierung kann beispielsweise online gemacht werden [6]. Das Ergebnis dieser Codierung ist dann ein Textstring, welcher nur noch aus druckbaren Zeichen besteht, die sich als normales Character-Array in die XML-Struktur einfügen.

Mit Hilfe des **<general />**-Tags wird eine eigene Tab-Pane erzeugt, in der alle Elemente angeordnet werden, welche sich zwischen die-

sen befinden. Hier in diesem Beispiel wird ein Eingabefeld (**<param></param>**) für Ganzzahlen (**<type>integer</type>**) angelegt, welches Werte im Bereich von 2 (**<min>2</min>**) bis 10000 (**<max>10000</max>**) akzeptiert und mit dem Wert aus **data->config.numRange** vorbelegt ist (**<default>%d</default>**). An dieser Stelle wird der Wert aus der Variablen **numRange** für den Default-Wert verwendet, damit sichergestellt ist, dass bereits geänderte und möglicherweise aus einem Projektfile geladene Werte auch korrekt angezeigt werden.

Alles, was vom **<helppanel>**-Tag eingeschlossen ist, landet wiederum in einer eigenen Tab-Pane „Beschreibung“, in der Sinn und Zweck der verschiedenen Ein- und Ausgänge noch einmal kurz erklärt sind. Diese Pane sollte in keinem Plug-in fehlen, da es als Gedächtnisstütze hilfreich ist.

Ein besonderes Augenmerk soll hier noch auf das Tag **<name>numrange</name>** für das Integer-Eingabefeld gelegt werden. Der vergebene Name muss innerhalb eines Plug-ins eindeutig sein, da er benötigt wird, um zu ermitteln, welchen Wert der Benutzer hier eingegeben hat:

So bald das Konfigurationspanel eines Plug-ins mit „*OK*“ verlassen wurde, kommt diese Funktion zum Zug. Das passiert mehrfach – für jedes in der XML-Struktur definierte Eingabefeld je einmal. Dabei wird im Parameter **name** der eindeutige Name des Eingabefeldes übergeben und in **value** der Wert, der vom Benutzer gewählt wurde. Dieser liegt dabei in jedem Fall als Character-Array vor und muss gegebenenfalls entsprechend konvertiert werden. Die Umwandlung in eine Ganzzahl geschieht hier über die Funktion **atoi()**.

Noch kurz ein Wort zur oben aufgeführten XML-Struktur: Hier wurden zwei Paneltypen und ein Typ für Dateneingaben vorgestellt. Tatsächlich existieren wesentlich mehr Möglichkeiten. Neben zusätzlichen Panels, welche jeweils eigene Namen haben können, gibt es auch Auswahlfelder, statische Texte, Eingabefelder für Texte und Fließpunkt-Zahlen, Dateiauswahldialoge, Buttons zur Auswahl einer Farbe und anderes mehr. Damit ist im Prinzip jede Art von Konfigurationsdialog machbar.

Was jetzt noch fehlt, sind Möglichkeiten, die lokal gehaltenen Konfigurationsdaten in ein gemeinsames Projektfile zu bringen bzw. die dort gespeicherten Daten zurück zu erhalten, wenn so eine **.apcp**-Projektdatei geladen wird. Auch das ist nicht wirklich kompliziert, allerdings gilt es, ein paar Kniffe zu beachten.

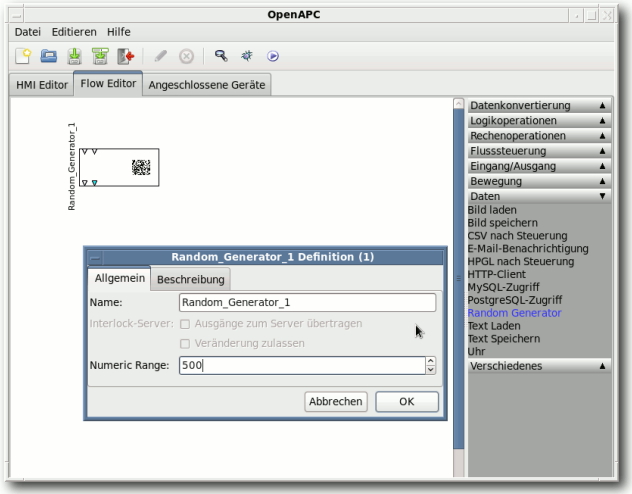
```
OAPC_EXT_API void oapc_set_config_data(void* instanceData,const char *name↷
,const char *value)
{
   struct instData *data;

   data=(struct instData*)instanceData;
   if (strcmp(name,"numrange")==0) data->config.numRange=atoi(value);
}
```

```
static struct libio_config   ↷
save_config;

OAPC_EXT_API char *↷
oapc_get_save_data(void* ↷
instanceData,unsigned long *length)
{
   struct instData *data;

   data=(struct instData*)↷
   instanceData;
   *length=sizeof(struct ↷
   libio_config);
   save_config.version =htons(1);
   save_config.length  =htons(*↷
   length);
   save_config.numRange=htonl(data↷
   ->config.numRange);

   return (char*)&save_config;
}
```

Die obige Funktion erwartet als Rückgabewert einen Pointer auf den Speicherbereich, in welchem die zu speichernden Daten liegen und in der Variablen, auf die **length** zeigt, deren Größe. Im Prinzip ist hier der Inhalt der zuvor angelegten Struktur **libio_config** zurückzugeben – in dieser befindet sich der zu speichernde Parameter **numRange**. Da das Plug-in zum einen kompatibel

zu möglichen zukünftigen Änderungen bleiben soll und zum anderen auch auf anderen Rechnerarchitekturen funktionieren muss, sind allerdings einige Maßnahmen erforderlich.

So werden zusätzlich eine Versionsnummer für die Datenstruktur und deren Länge gespeichert und in den zu übergebenden Daten abgelegt.



Das erste eigene Plug-in.

Des Weiteren werden alle Member der Struktur **libio_config** mittels **htons()** und **htonl()** in das plattformunabhängige Netzwerk-Datenformat umgewandelt. Dieses dient eigentlich dazu, die direkte Datenübertragung zwischen Rechnern unterschiedlicher Architektur zu ermöglichen, kann aber auch dann problemlos verwendet werden, wenn diese Daten „nur“ gespeichert werden. Es ist lediglich zu beachten, das beim Laden die Rückkonvertierung mittels **ntohs()** und **ntohl()** stattfindet:

```
OAPC_EXT_API void oapc_set_loaded_data(void* instanceData,unsigned long ↷
length,char *loadedData)
{
   struct instData *data;

   data=(struct instData*)instanceData;
   if (length>sizeof(struct libio_config)) length=sizeof(struct ↷
   libio_config);
   memcpy(&save_config,loadedData,length);
   data->config.version =ntohs(save_config.version);
   data->config.length  =ntohs(save_config.length);
   if ((data->config.version!=1) || (data->config.length!=sizeof(struct ↷
   libio_config)))
   {
      // do conversion from earlier versions here
   }
   data->config.numRange=ntohl(save_config.numRange);
}
```

Die obige Funktion handhabt die umgekehrte Richtung: In **loadedData** werden die geladenen Daten mit der Länge **length** übergeben und anschließend konvertiert. An dieser Stelle ist bei späteren Versionen des Plug-ins dann auch zu überpüfen, ob die geladenen Daten eventuell konvertiert werden müssen.

Damit ist ein wesentlicher Teil des Plug-ins fertiggestellt. Nachdem es erfolgreich mittels eines simplen Aufrufes von

```
$ make
```

in der Konsole kompiliert wurde, kann es im Flow-Editor des OpenEditors erstmalig getestet werden.

Funktionen für den Player

Nachdem bis eben auch einige Funktionen beschrieben wurden, welche in Player und Editor verwendet werden, geht es jetzt an die eigentliche Implementierung der Plug-in-Funktionalität. Die folgenden Funktionen werden im Player bzw. Debugger ausschließlich dazu benutzt, um einem Plug-in überhaupt „Leben“ einzuhauchen.

Wird ein Plug-in im Player oder Debugger gestartet, ist der erste Aufruf vom Player an das Plug-in auch wieder der zum Erzeugen einer neuen Instanz per **oapc_create_instance2()**. Anschließend werden die gespeicherten Werte mittels eines Aufrufs der Plug-in-Funktion **oapc_set_loaded_data()** übergeben. Beide Funktionen existieren bereits. Im nächsten

Schritt versucht der Player nun, das Plug-in zu initialisieren:

```
OAPC_EXT_API unsigned long ↷
oapc_init(void* instanceData)
{
   struct instData *data;

   data=(struct instData*)malloc(↷
   sizeof(struct instData));
   srand(time(NULL));
   return OAPC_OK;
}
```

Die einzig erforderliche Initialisierung für dieses Beispiel ist der Aufruf von **srand()** für den Zufallsgenerator. Wäre dies ein Plug-in, mit dem Hardware angesteuert werden müsste, so wäre diese Funktion ein guter Ort, um alle notwendigen Maßnahmen wie das Öffnen des zugehörigen Geräte-Devices, das Senden der Firmware oder von Initialisierungsparametern zu erledigen. Diese Werte sollten dann wieder aus der Struktur **instData** kommen und konfigurierbar sein. Wichtig in diesem Zusammenhang sind auch die Returncodes dieser Funktion, diese geben darüber Auskunft, was schiefgegangen ist und sorgen dafür, dass der Benutzer durch den Player auch gleich eine aussagekräftige Fehlermeldung angezeigt bekommt. Hier existieren folgende vordefinierte Codes, welche – sofern sie zum aufgetretenen Problem passen – im Prinzip für fast alle Funktionen verwendet werden können:

- **OAPC_OK** – die Operation war erfolgreich, alles ist in Ordnung
- **OAPC_ERROR_CONNECTION** – es ist ein Verbindungsproblem aufgetreten (z. B. bei Netzwerkkommunikation)
- **OAPC_ERROR_DEVICE** – ein benötigtes Geräte-Device steht nicht zur Verfügung bzw. konnte nicht erfolgreich geöffnet werden
- **OAPC_ERROR_NO_DATA_AVAILABLE** – es wurden Daten abgefragt, obwohl momentan keine zur Verfügung stehen (dieser Returncode ist bei den Funktionen zum Holen von Ausgangsdaten wichtig)
- **OAPC_ERROR_NO_SUCH_IO** – der spezifizierte Ein-/Ausgang existiert nicht (dieser Returncode ist ebenfalls für die I/O-Funktionen wichtig und würde etwas signalisieren, was so eigentlich nicht auftreten kann, wenn OpenAPC-Software und Plug-in fehlerfrei sind)
- **OAPC_ERROR_NO_MEMORY** – es ist nicht genügend Speicher vorhanden
- **OAPC_ERROR_RESOURCE** – eine sonstige benötigte Ressource steht nicht zur Verfügung
- **OAPC_ERROR_AUTHENTICATION** – es ist ein Fehler bei einer Authentifizierung aufgetreten, die für eine Operation benötigten Zugangsdaten sind falsch
- **OAPC_ERROR_CONVERSION_ERROR** – es war nicht möglich, eine erforderliche Datenkonvertierung durchzuführen
- **OAPC_ERROR_STILL_IN_PROGRESS** – es ist nicht möglich, neue Daten entgegenzunehmen, da die vorhergehende Operation noch läuft (ist in erster Linie bei den Funktionen zum Setzen von Eingangsdaten sinnvoll)
- **OAPC_ERROR_RECV_DATA** – der Empfang von Daten ist fehlgeschlagen
- **OAPC_ERROR_SEND_DATA** – das Senden von Daten ist fehlgeschlagen
- **OAPC_ERROR_PROTOCOL** – Fehler in einem (Kommunikations-)Protokoll
- **OAPC_ERROR_INVALID_INPUT** – die Eingangsdaten sind ungültig (z. B. außerhalb eines zulässigen Bereiches)
- **OAPC_ERROR** – es ist ein sonstiger Fehler aufgetreten, der durch die anderen Codes nicht abgedeckt wird

Auch Teil der Plug-in-Initialisierung ist der Aufruf einer Funktion, welcher nur dann erfolgt, wenn das Capability-Flag **OAPC_ACCEPTS_IO_CALLBACK** gesetzt war:

```
OAPC_EXT_API void ↷
oapc_set_io_callback(void* ↷
instanceData,lib_oapc_io_callback ↷
oapc_io_callback,unsigned long ↷
callbackID)
{
   struct instData *data;

   data=(struct instData*)↷
   instanceData;
   m_oapc_io_callback=↷
   oapc_io_callback;
   data->m_callbackID=callbackID;
}
```

Hier teilt der Player mit, über welche Funktion er darüber informiert werden möchte, wenn neue

Daten vorhanden sind, welche von ihm anschließend abgeholt werden können. Der entsprechende Funktionspointer **oapc_io_callback** sowie der ebenfalls benötigte Callback-Code **callbackID** werden für die spätere Verwendung gespeichert. Der Funktionspointer darf dabei in einer globalen Variable landen, da er keiner spezifischen Plug-in-Instanz zugeordnet ist, sondern offenbar für alle Plug-ins immer auf den gleichen Einsprungpunkt zeigt.

Wichtig ist auch die Deinitialisierung, welche beim Beenden des Players erfolgt, hierfür wird ebenfalls eine eigene Funktion benötigt:

```
OAPC_EXT_API unsigned long ↷
oapc_exit(void* instanceData)
{
   return OAPC_OK;
}
```

Für dieses Beispiel-Plug-in muss an dieser Stelle nichts unternommen werden, da keine Speicherbereiche freizugeben oder Geräte-Devices zu schließen sind. Das wäre wieder anders, wenn Hardware im Spiel ist, die vom Plug-in angesprochen wird.

Alle folgenden Funktionen dienen der Datenkommunikation und existieren als „set“-Funktion zum Setzen von Eingangsdaten, als „get“-Funktion zum Abholen von Ausgangsdaten. Diese gibt es dann dann jeweils in Varianten für Digital-, numerische, Binärdaten und Zeichenketten. Ob und welche dieser Funktionen vorhanden sein müssen und vom Player erwartet werden, hängt von den oben gesetzten IO-Flags **OAPC_xxx_IOy** für Ein- und Ausgänge ab.

```
OAPC_EXT_API unsigned long  oapc_set_digi_value(void* instanceData,↷
unsigned long input,unsigned char value)
{
   struct instData           *data;

   if (value!=1) return OAPC_OK;
   data=(struct instData*)instanceData;
   if (input==0)
   {
      if (rand()>RAND_MAX/2) data->m_digi=1;
      else data->m_digi=0;
      m_oapc_io_callback(OAPC_DIGI_IO0,data->m_callbackID);
   }
   else if (input==1)
   {
      double factor;

      factor=(1.0*data->config.numRange)/RAND_MAX;
      data->m_num=(int)(rand()*factor);
      m_oapc_io_callback(OAPC_NUM_IO1,data->m_callbackID);
   }
   else return OAPC_ERROR_NO_SUCH_IO;
   return OAPC_OK;
}
```

An dieser Stelle findet die Implementierung der Zufallsfunktion für die Eingänge 0 (**if (input==0)**) und 1 (**else if (input==1)**) statt. Mittels der Funktion **rand()** wird ein neuer Zufallswert ermittelt und in der zugehörigen Variable **data->m_digi** bzw. **data->m_num** gespeichert. Wird irrtümlich ein Wert für einen anderen Eingang übergeben, so quittiert die Funktion das mit Hilfe des Rückabewertes **OAPC_ERROR_NO_SUCH_IO**.

An dieser Stelle wird nun auch Gebrauch von der Callback-Funktion gemacht: Der Funktionspointer **m_oapc_io_callback()** dieser Funktion wird jetzt verwendet, um dem Player bzw. Debugger mitzuteilen, dass neue Daten am Digitalausgang 0 (**OAPC_DIGI_IO0**) bzw. am numerischen Ausgang 1 (**OAPC_NUM_IO1**) zur Verfügung stehen. Die eindeutige Zuordnung zu dieser spezi-

ellen Instanz des Plug-ins geschieht dabei über den zweiten Parameter **data->m_callbackID** welcher der Callback-Funktion mitgegeben wird.

Infolge dieses Funktionsaufrufes wird jetzt die jeweilige „get“-Funktion des Plug-ins vom Debugger bzw. Player aufgerufen, um die neuen Daten abzuholen:

```
OAPC_EXT_API unsigned long  oapc_get_digi_value(void* instanceData,↷
unsigned long output,unsigned char *value)
{
   struct instData *data;

   data=(struct instData*)instanceData;
   if (output==0) *value=data->m_digi;
   else return OAPC_ERROR_NO_SUCH_IO;
   return OAPC_OK;
}

OAPC_EXT_API unsigned long  oapc_get_num_value(void* instanceData,unsigned↷
 long output,double *value)
{
   struct instData *data;

   data=(struct instData*)instanceData;
   if (output==1) *value=data->m_num;
   else return OAPC_ERROR_NO_SUCH_IO;
   return OAPC_OK;
}
```

Beide überprüfen, ob der Aufruf für den richtigen Ausgang erfolgt (**if (output==...**) und geben dann den gespeicherten Zufallswert **data->m_digi** bzw. **data->m_num** zurück. Wäre echte Hardware im Spiel, wäre das der Ort, an dem von der Hardware gelesene Daten an den Player übergeben werden könnten. Ähnlich die „set“-Funktionen: Diese würden die Schnittstelle von der Software hin zum Gerät darstellen.

Ich habe fertig!

Auch wenn das auf den ersten Blick etwas kompliziert aussieht, so geht die Implementierung von eigenen Plug-ins nach einer Weile recht flüssig von der Hand. An gewisse Merkwürdigkeiten wie Capability-Flags, zu denen eigentlich gar keine Alternativen existieren, gewöhnt man sich und der Rest erscheint einem irgendwann logisch.

Auch ist der umfangreiche Fundus an Sourcen und fertigen Plug-ins durchaus hilfreich – sei es, um aus deren Programmierung zu lernen oder sei es, um sie als Basis für eigene Plug-ins zu verwenden. Die Lizenz der allermeisten Plug-ins lässt es sogar zu, dass diese verändert werden, ohne dass die Sourcen anschließend veröffentlicht werden müssen [7].

Es bleibt noch zu erwähnen, dass dieser kleine Artikel wirklich nur als Einstieg in die Plug-in-Programmierung gedacht ist. Die Möglichkeiten innerhalb der OpenAPC-Software sind weit umfassender.

So existieren nicht nur wesentlich mehr Konfigurationselemente, welche sich per XML erzeugen lassen, auch können visuelle Plug-ins entwickelt werden, welche dem HMI ganz neue grafische Elemente hinzufügen. Auch zu erwähnen wäre die mitgelieferte Bibliothek **liboapc.so**. Diese bietet viele zum Teil recht einfache Funktionalitäten, diese aber in einer plattformunabhängigen Version. So ist es damit beispielsweise nicht notwendig, Zugriffe auf die serielle Schnittstelle für jedes Betriebssystem separat zu implementieren. Hier bietet diese Bibliothek bereits fertige Funktionalitäten.

Auch bei eher trivial anmutenden Features wie Netzwerkzugriffen, Threading-Funktionalitäten oder Timern sollten möglichst immer die Funktionen der **liboapc.so** verwendet werden, um einen möglichen Portierungsaufwand auf andere Plattformen klein zu halten. Denn auch wenn ein **pthread_create()** [8] unter Linux ganz simpel

ist – unter Windows existiert es schlichtweg nicht und würde an dieser Stelle schon wieder diverse **`#ifdef`**'s erforderlich machen.

LINKS

[1] http://www.freiesmagazin.de/freiesMagazin-2011-01

[2] http://www.freiesmagazin.de/freiesMagazin-2011-03

[3] http://www.openapc.com/download.php

[4] http://www.openapc.com/download/index.php?c=1&f=opensdk.zip

[5] https://secure.wikimedia.org/wikipedia/de/wiki/Sudo

[6] http://www.motobit.com/util/base64-decoder-encoder.asp

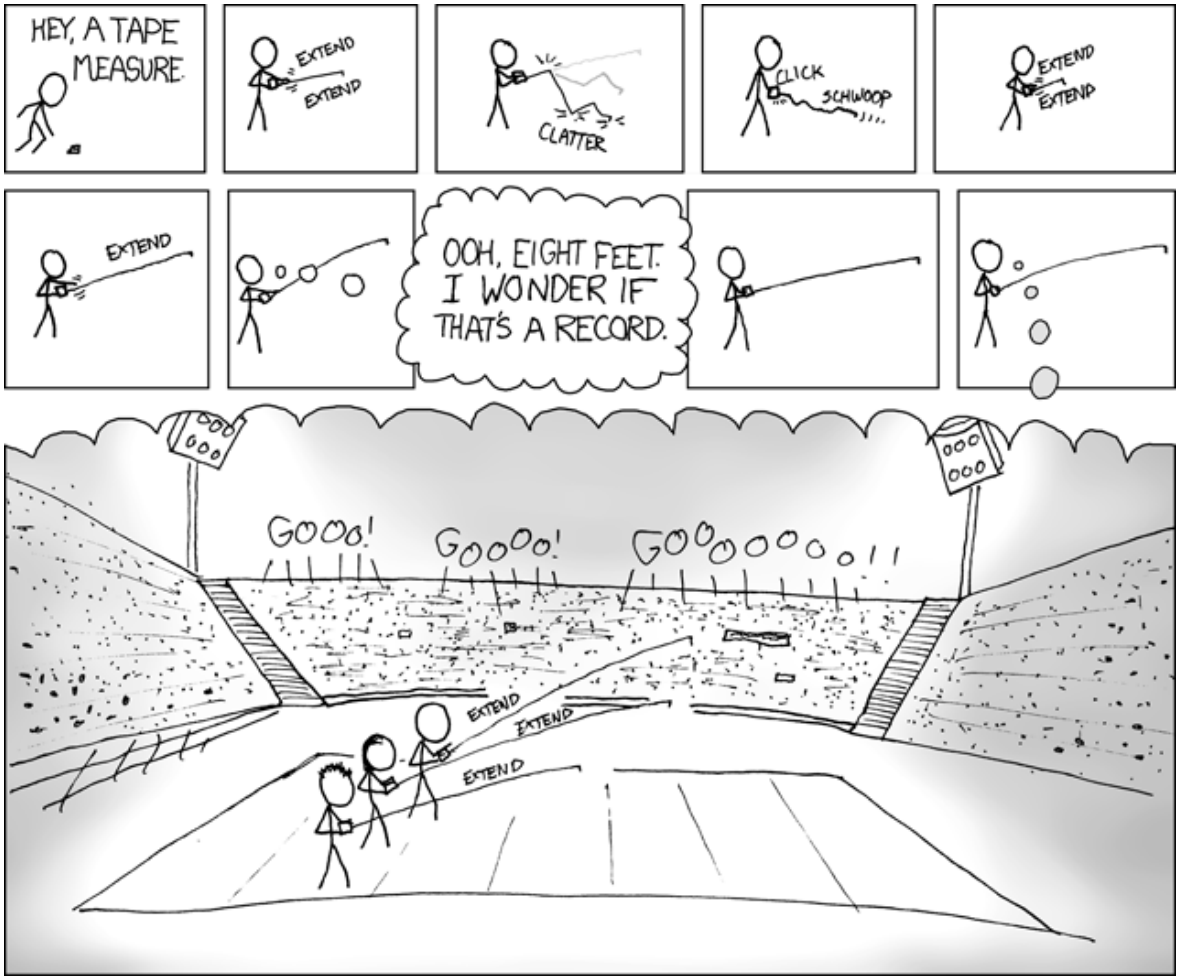
[7] http://www.openapc.com/oapc_license.php

[8] http://www.manpagez.com/man/3/pthread_create/

Autoreninformation

Hans Müller ist als Elektroniker beim Thema Automatisierung mehr der Hardwareimplementierung zugeneigt als dem Softwarepart und hat demzufolge auch schon das ein oder andere Gerät in der privaten Wohnung verkabelt.

Diesen Artikel kommentieren



„Tape Measure“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/284