SANE Backend für Canon USB Scanner

Kennen Sie das? Sie gehen frohen Mutes in ein Geschäft, erwerben nach langem Überlegen ein neues Gerät für Ihren Computer und zu Hause stellt sich heraus, dass die Hardware nicht mit Ihrem Computer läuft. Dem Autor dieses Artikels ist dies leider passiert. Dieser Artikel beschreibt, wie *Perl* dabei geholfen hat, ein SANE Backend zu realisieren [1].

Der Flachbettscanner

Meine Neuerwerbung war der USB Flachbettscanner *CanoScan LiDE 600F*. Dieser ist kompatibel zum Modell *CanoScan LiDE 600*, besitzt aber noch einen steckbaren Aufsatz zum Scannen von Negativen. Die Hardware ist sehr gut, nur an der Software für Linux hapert es. Aber hier kann man als Programmierer nachbessern. Man erinnere sich dabei an das wohlbekannte Zitat aus Wilhelm Tell: „Die Axt im Haus erspart den Zimmermann“.

Das SANE Projekt

Um einen Scanner unter Linux in Betrieb zu nehmen, kommt man um das *SANE* Projekt [2] nicht herum. *SANE* steht für „Scanner Access Now Easy“ und stellt eine einfache und vor allem komfortable API zwischen Scanner und Anwendung dar. Dabei beschränkt sich *SANE* nicht nur auf Scanner, sondern erlaubt es auch, auf Videokameras oder Webcams zuzugreifen.

Auf den Webseiten fand sich dann auch der Hinweis, dass der Scanner *CanoScan LiDE 600F* nicht unterstützt wird. Dass sich noch niemand der Hardware angenommen hat, liegt wohl an dem in diesem Scanner eingesetzten Chip (Philips CP2155BE). Nachfragen in China bzw. Taiwan bei verschiedenen Herstellern lieferte das Ergebnis, dass kein Datenblatt erhältlich ist, weil es sich um einen „custom chip“ handelt.

Keine Hilfe vom Hersteller

Man sollte meinen, dass man vom Hersteller eines Produktes zumindest ein klein wenig Hilfe bekommen könnte. Leider musste ich feststellen, dass die Firma *Canon* überhaupt nicht gewillt ist irgendeine Art von Hilfe oder Dokumentation anzubieten, die es ermöglichen könnte, einen Linux-Treiber zu schreiben.

Am Telefon sagte mir der Mitarbeiter vom Support, dass dies so in Japan beschlossen wurde und es keine Ausnahmen gäbe. Diese Unternehmenspolitik ist für mich nicht nachvollziehbar. Zumal der Linux-Treiber doch kostenfrei erstellt wird und der Hardwarehersteller dadurch problemlos einen weiteren Kundenkreis erschließen könnte.

Vielleicht scheuen sich einige Hersteller davor, den Quellcode der Treiber und der Software freizugeben, weil sie glauben das Risiko, einer abstrusen Patentverletzung bezichtigt zu werden, sei dann höher.

USB Daten aufzeichnen

Somit blieb nur noch der Weg selbst „in die Untiefen der Hardware hinabzusteigen“.

Da dieser Scanner ein USB Interface besitzt, bot es sich an ein paar spezifische Scans durchzuführen und die dabei übertra-

genen Daten mit einem sogenannten USB Sniffer aufzuzeichnen. Mit etwas Glück ist man dann in der Lage, das Protokoll zu entschlüsseln. Um die Log-Dateien aufzuzeichnen, verwendete ich *Win XP* und *Benoit's USB Sniffer* [3].

Es ist sehr verwunderlich, wie groß solche Log-Dateien werden können. Allein das Einstöpseln des Scanners dokumentierte das Log mit 7 MB. Der Scan einer Briefmarke in Farbe schlug mit 83 MB zu Buche. Klar, dass diese Dateien erst noch mit *Perl* geparst und verkleinert werden mussten. Ich erspare mir daher den Abdruck dieser Daten und zeige den Inhalt der ersten Zeilen einer geparsten Fassung. Siehe Listing 1.

Da ein USB Host bei einem neuen USB-Gerät nur weiß, dass soeben ein Gerät angeschlossen wurde, aber nicht welche Eigenschaften es besitzt, muss er diese Informationen zuerst abfragen. Dies geschieht mit dem Kommando GET_DESCRIPTOR. Danach wird meist eine bestimmte Konfiguration des Gerätes mit dem Kommando SELECT_CONFIGURATION ausgewählt. Da es für die Dekodierung der Scannerbefehle unrelevant war, habe ich diese Sequenzen in der geparsten Form entfernt und durch Kommentarzeilen ersetzt.

Kommen wir nun zu den Kommandos, die Daten übertragen. Diese werden mit dem Wort BULK (bulk transfer) gekennzeichnet. Mit > und < wird die Übertragungsrichtung zum und vom Scanner angegeben. Dahinter steht in hexadezimaler Schreibweise die Adresse des USB Endpoints. Es folgt die fortlaufende Nummer des URB und der Zeitpunkt seit dem Start des Log in Millisekunden. Die übertragenen Bytes werden als Hexdump in der nachfolgenden Zeile aufgeführt.

```
# GET_DESCRIPTOR
# SELECT_CONFIGURATION
# GET_DESCRIPTOR
BULK > 02 (URB 6) [134 ms]
    00000000: 01 d0 01 00

BULK < 83 (URB 7)
    00000000: 81

BULK > 02 (URB 8) [134 ms]
    00000000: 01 46 01 00

BULK < 83 (URB 9)
    00000000: 00

BULK > 02 (URB 10) [135 ms]
    00000000: 00 72 01 00 1c

BULK > 02 (URB 11) [148 ms]
    00000000: 00 73 01 00 00
```

Listing 1: Geparste Fassung einer USB Log-Datei

Reverse Engineering mit Perl

Nun kann man sich daran machen, die Bedeutung dieser Daten zu entschlüsseln. Dies erweist sich je nach Scanner mehr oder weniger schwer. Glücklicherweise ist es beim *CanoScan LiDE 600F* sehr einfach. Der Abstraktionsgrad des Protokolls ist sehr hoch. Die Sequenzen könnte man als Funktionsaufrufe einer speziellen Scanner-Sprache ansehen. Ich versuche dies kurz zu verdeutlichen.

Es fällt auf, dass die Zugriffe nur auf die Endpoints 02 und 83 gehen, wobei über 02 die Ausgabe und über 83 die Eingabe von Daten erfolgt und nach einer statistischen Analyse über die Häufigkeit einzelner Bytes kommt man der Bedeutung auf die Spur. Die Datensequenz 01 d0 01 00 besteht aus vier Teilen: dem Befehlscode, der Registeradresse und einer Länge als 16Bit-Wert (LSB first).

URB 6 bedeutet somit folgendes: „Frage den Wert des Registers d0 ab". Der Rückgabewert (URB 7) ist ein einzelnes Byte. Analoges gilt für URB 8 und 9. Für URB 10 bis 11 gibt es keine Rückgabewerte. Diese Befehle setzen ein Register. Die Datensequenz 00 72 01 00 1c von URB 10 bedeutet daher sinngemäß: „Setze das Register 72 auf den Wert 1c".

Insgesamt lässt sich die in Listing 1 gezeigte Sequenz in einer einfachen Pseudosprache schreiben (siehe Listing 2).

Variablenzuweisungen gibt es hier nicht. Die Werte sind daher als Kommentar angegeben.

Damit ist die Dekodierung des Protokolls schon abgeschlossen. Es gibt außer get und set noch ein paar weitere Befehle, die ich aber wegen der Kürze des Artikels hier nicht aufführen möchte.

An diesem Punkt haben wir z.B. die oben erwähnte 7 MB große Log-Datei auf etwa 7 KB reduziert! Dies entspricht etwa 300 Zeilen in obiger Pseudosprache.

Um nun mit *Perl* den Scanner über USB steuern zu können, müssen wir nur die obigen Pseudobefehle als Basisfunkti-

```
( get d0 )      # -> 81
( get 46 )      # -> 00
( set 72 1c )
( set 73 00 )
( set 41 80 )
...
```

Listing 2: Dekodierte Fassung der USB-Übertragung

onen implementieren. Siehe Listing 4. Baut man sich noch einen Parser für obige Sprache, kann man den Scanner sogar direkt in dieser Sprache programmieren. Dies ist nicht sehr kompliziert. Für Interessierte sei erwähnt, dass obige Syntax an *Lisp* bzw. *Scheme* angelehnt ist.

Ich will nun anhand von zwei Listings zeigen, wie ein Zugriff auf ein USB-Gerät mit *Perl* aussehen kann. Dabei reduziere ich die Befehle auf das Nötigste.

Man braucht zuerst das *Perl*-Paket `Device::USB` [4], um damit auf *libusb* zugreifen zu können. Es kann vom *CPAN* [5] heruntergeladen werden.

Um das USB-Gerät anzusprechen, erzeugt man zuerst ein neues `Device::USB` Objekt. Dann sucht man das Gerät über `find_device` und öffnet es anschließend mit `open`. Ein `close` wird nicht benötigt. Natürlich sollte man in einer robusten Implementation noch Prüfungen einbauen, ob die Befehle erfolgreich waren.

Die weiteren Befehle zeigen, wie man mit *Perl* die USB Konfiguration des geöffneten Gerätes ausliest. Man kann noch mehr auslesen, aber das Beispiel zeigt das Wesentliche.

Die Methode *find_device* sucht hier gezielt nach dem *CanoScan LiDE 600F*. Für andere Geräte muss man die beiden Parameter anpassen. Der erste Parameter ist die *Vendor ID* und der zweite Parameter die *Produkt ID*.

Sollte man diese noch nicht kennen kann man sich z.B. mit dem Befehl `sane-find-scanner -v -v` eine ähnliche Ausgabe anzeigen lassen wie mit obigen Skript. Ansonsten sind die IDs auch in der Ausgabe des Kommandos `cat /proc/bus/usb/devices` zu finden.

```
#!/usr/bin/perl
use Device::USB ;

my $usb = Device::USB -> new() ;
my $dev = $usb -> find_device ( 0x04a9 , 0x2224 ) ;
$dev -> open() ;

display ( 'bcdUSB' , $dev -> bcdUSB() ) ;
display ( 'bDeviceClass' , $dev -> bDeviceClass() ) ;
display ( 'bDeviceSubClass' , $dev -> bDeviceSubClass() ) ;
display ( 'bDeviceProtocol' , $dev -> bDeviceProtocol() ) ;
display ( 'bMaxPacketSize0' , $dev -> bMaxPacketSize0() ) ;
display ( 'idVendor'  , sprintf('0x%04X',$dev -> idVendor()) ) ;
display ( 'idProduct' , sprintf('0x%04X',$dev -> idProduct()) ) ;
display ( 'bcdDevice' , $dev -> bcdDevice() ) ;
display ( 'iManufacturer' , $dev -> iManufacturer() , 1 ) ;
display ( 'iProduct' , $dev -> iProduct() , 1 ) ;
display ( 'iSerialNumber' , $dev -> iSerialNumber() , 1 ) ;
display ( 'bNumConfigurations' , $dev -> bNumConfigurations() ) ;

for ( $i = 0 ; $i < $dev -> bNumConfigurations() ; $i ++ )
{
  display ( ' <configuration ' . $i . '>' ) ;
  $cfg = $dev -> get_configuration ( $i ) ;
  display ( ' wTotalLength' , $cfg -> wTotalLength() ) ;
  display ( ' bNumInterfaces' , $cfg -> bNumInterfaces() ) ;
  display ( ' bConfigurationValue' , $cfg -> bConfigurationValue() ) ;
  display ( ' iConfiguration' , $cfg -> iConfiguration() ) ;
  display ( ' bmAttributes' , $cfg -> bmAttributes() , 'Remote Wakeup' ) ;
  display ( ' MaxPower' , $cfg -> MaxPower() . ' mA' ) ;
}

sub display
{
  my ( $key , $val , $txt ) = @_ ;
  my $pad = ' ' x 22 ;
  $key = substr ( $key . $pad , 0 , 22 ) ;
  if ( $txt == 1 ) { $txt = $dev -> get_string_simple ( $val ) ; }
  if ( $txt ) { $txt = ' (' . $txt . ')' ; }
  print ( $key , $val , $txt , "\n" ) ;
}
```

Listing3: USB Geräte-Informationen ausgeben

```
#!/usr/bin/perl
use Device::USB ;

my $timeout = 500 ;
my $usb = Device::USB -> new() ;
my $dev = $usb -> find_device ( 0x04a9 , 0x2224 ) ;
$dev -> open() ;
print 'libusb:xxx:' , $dev -> filename() , "\n" ;

dispreg ( 'd0' ) ;
dispreg ( '46' ) ;
canon_set ( '02' , '01' ) ;
canon_set ( '02' , '00' ) ;
canon_set ( '01' , '00' ) ;
canon_set ( '01' , '28' ) ;
canon_set ( 'a0' , '04' ) ;
canon_set ( 'a0' , '05' ) ;
canon_set ( '01' , '28' ) ;
canon_set ( '04' , '0c' ) ;
canon_set ( '05' , '00' ) ;
canon_set ( '06' , '00' ) ;
canon_set ( '90' , '27' ) ;
canon_set ( '92' , 'f7' ) ;
canon_set ( '94' , 'f7' ) ;
canon_set ( '93' , '00' ) ;
canon_set ( '91' , '1f' ) ;

while ( 1 )
{
  dispreg ( '91' ) ;
  sleep ( 1 ) ;
}

1 ;

sub canon_set
{
  my ( $reg , $val ) = @_ ;
  my $data = '00 ' . $reg . ' 01 00 ' . $val ;
  my $cnt = $dev -> bulk_write ( hex('02') , hex2bin($data) , $timeout ) ;

  if ( $cnt < 0 )
  {
    die ( join ( ' ' , 'ERROR: canon_set' , $cnt , $reg , '=' , $val ) ) ;
  }

  print 'SET ' , $reg , ' ' , $val , "\n" ;
}

sub canon_get
{
  my ( $reg ) = @_ ;
  my $data = '01 ' . $reg . ' 01 00' ;
  my $cnt = $dev -> bulk_write ( hex('02') , hex2bin($data) , $timeout ) ;

  if ( $cnt < 0 )
  {
    die ( join ( ' ' , 'ERROR: canon_get' , $cnt , $reg ) ) ;
  }

  #--- warten bis Scanner die Daten bereit hat ---
  select ( undef , undef , undef , 0.001 ) ;

  my $data = ' ' ;
  my $cnt = $dev -> bulk_read ( hex('83') , $data , 1 , $timeout ) ;

  if ( $cnt < 0 )
  {
    error ( 'canon_get' , $cnt , $reg , '(bulk_read 0x83)' ) ;
  }

  $data = bin2hex ( $data ) ;
  return ( $data ) ;
}
```

```
sub dispreg
{
  my ( $reg ) = @_ ;
  my $data = canon_get ( $reg ) ;
  print $reg , ' = ' , $data , '  ' ,
        sprintf('%08b',hex($data)) , "\n" ;
}

sub hex2dez
{
  my ( $txt ) = @_ ;
  $txt =~ s![\da-f]+:!!ig ;
  $txt =~ s!\A\s+!! ;
  $txt =~ s!\s+\Z!! ;
  my @data = split ( m!\s+! , $txt ) ;
  map { $_ = hex ( $_ ) } ( @data ) ;
  return ( @data ) ;
}

sub bin2hex
{
  my ( $data ) = @_ ;
  my @data = unpack ( 'C*' , $data ) ;
  my $anz = @data ;
  my $hex = sprintf ( '%02x ' x $anz ,
                      @data ) ;
  return ( $hex ) ;
}

sub hex2bin
{
  my ( $txt ) = @_ ;
  my @data = hex2dez ( $txt ) ;
  my $data = pack ( 'C*' , @data ) ;
  return ( $data ) ;
}
```

Listing4: Lesen/schreiben der Scannerregister

Hier nun noch ein umfangreicheres Beispiel mit dem man die Funktionstasten am Scanner abfragen kann. Wer möchte, kann hiermit leicht ein Skript schreiben, das je nach gedrückter Scannertaste ein bestimmtes Kommando aufruft.

In den Funktionen `canon_set` und `canon_get` sieht man sehr schön, wie die zuvor beschriebenen Datensequenzen als Hex-String zusammengebaut werden. Für die Übertragung zum Scanner muss dieser String ins Binärformat umgewandelt werden. Hierzu dient die Funktion `hex2bin`.

Umgekehrt werden mit `bin2hex` die Daten vom Scanner in ein lesbares Format umgewandelt.

Daten überträgt man mit den Methoden `bulk_write` und `bulk_read`. Bei `bulk_write` muss man keine Länge angeben.*Perl* kennt die Länge des Strings. Aber bei `bulk_read` muss man angeben, wieviele Bytes man lesen möchte. **ACHTUNG**: Man muss einen String als Puffer bereitstellen, der schon die nötige Größe besitzt. Andernfalls kann `bulk_read` die Daten nicht in den Puffer schreiben. Daher wurde `$data` vor dem `bulk_read` mit einem Leerzeichen initialisiert.

Erhält man als User keinen Zugriff auf das USB-Gerät, müssen die Skripte unter dem User *root* laufen. Dies liegt an den eingeschränkten Berechtigungen für normale User. Natürlich kann man kurzfristig die Berechtigungen in `/proc/bus/usb/00?/00?` ändern. Das ist aber nicht besonders sinnvoll, da jedes neue Einstöpseln den Dateinamen ändert und alle Berechtigungen sind dahin.

Geschickter ist es, dies automatisch über *hotplug* einstellen zu lassen. Voraussetzung ist dabei, dass der User Mitglied der Gruppe „Scanner“ ist. Bei den neusten Linuxsystemen sollte dies schon per Default so eingestellt sein.

Von Perl nach C

Nachdem die grundlegenden Funktionen in *Perl* vorlagen, war es sehr einfach die Steuerung nach *C* zu portieren. Hätte ich das Reverse Engineering mit *C* realisieren müssen, wäre sicher alles wesentlich umständlicher geworden.

Das Backend ist nun soweit, dass ich erste Scanversuche mit dem *Xsane* Frontend machen konnte. Es bedarf aber noch weiterer Arbeit, um damit korrekte Scans mit unterschiedlichen Auflösungen zu erzeugen.

Noch ein weiterer Punkt ist mir aufgefallen. Mit *Device::USB* muss man die Nummer (hier 02 und 83) des Endpoints angeben über den man Daten transferieren möchte. Ähnliche Funktionen der *SANE API* in *C*, wie sie mit `sanei_usb_write_bulk` aufgerufen werden, erfordern jedoch die Angabe des Endpoint-Index. Die *sanei*-Funktionen suchen dann über den Index erst die Endpoint-Nummer heraus. Für dieses Backend war der Index immer Null. Ein Lesen oder Schreiben über diesen Index wurde dann automatisch den beiden Endpoints zugeordnet. Dies ist für den Anfänger möglicherweise verwirrend und kostet Zeit bei der Fehlersuche.

Jürgen Ernst

Referenzen

[1] http://www.juergen-ernst.de/info_sane.html
[2] http://www.sane-project.org
[3] http://benoit.papillault.free.fr/usbsnoop/
[4] http://search.cpan.org/~gwadej/Device-USB-0.21/lib/Device/USB.pm
[5] http://www.cpan.org/