Computing

Programmierung des TX-Controllers mit Python

Raphael Jacob

Im Sommer 2009 kam der TX-Controller auf den Markt. Unter Windows ist dieser mit ROBOPro sehr einfach anzusteuern, aber wenn ein Programm unter Linux oder gar einer anderen Architektur, wie zum Beispiel ARM (TXT-Controller) laufen soll, stößt man schnell an Grenzen. Das Problem lässt sich aber lösen – denn auch den TX-Controller kann man mit Python ansteuern.

Problemstellung

Alle Interfaces vor dem TX-Controller (im Folgenden kurz TX-C) und der TXT-Controller lassen sich auf allen Plattformen mit Python ansteuern, aber den TX-C kann man – zumindest „von Haus aus“ – nur unter Windows mit ROBOPro oder unter Verwendung einer bereits fertig kompilierten Bibliothek bspw. mit einem C-Programm ansteuern. Sollte es da nicht eine alternative Möglichkeit geben, auch den TX direkt mit Python zu steuern?

Idee

Daher begann ich mit Nachforschungen, welche Informationen über den TX von fischertechnik bereitgestellt werden oder von Fans entdeckt wurde. fischertechnik stellt auf seiner Homepage zu der oben genannten Bibliothek die Headerdateien und eine Anleitung sowie eine Beschreibung des Betriebssystems bereit [1].

Schnell stellte sich heraus, dass das Protokoll auf dem Erweiterungsbus (RS-485) identisch mit dem auf dem USB-Anschluss und einer Bluetooth-Verbindung ist. So boten sich drei weitere Quellen an:

- Christoph Nießen untersucht den Erweiterungsbus
- Thomas Kaiser analysiert noch etwas genauer die Kommandos des Protokolls
- Ad van der Weiden hat einen Konverter vom TX-C zur Robo-Interface-Extension gebaut

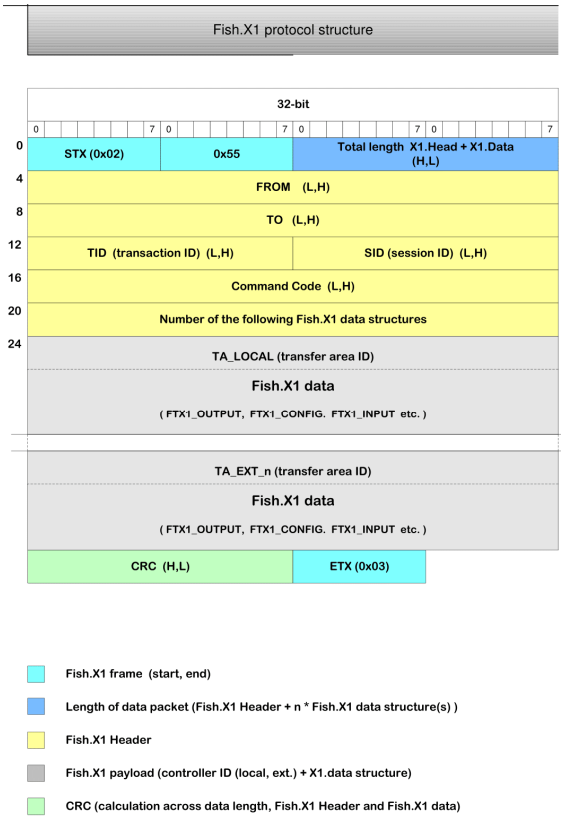

Abb. 1: Paketaufbau (Quelle: MSC, [1])

Analyse

Im zweiten Schritt habe ich den TX-C an einen Computer angeschlossen, ROBOPro geöffnet und den Interfacetest gestartet. Gleichzeitig habe ich im Hintergrund mit Wireshark die Kommunikation mitgeschnitten. Als Ergebnis hatte ich einen riesigen Berg von HEX-Zeichen. Zunächst habe ich das Ganze auf die Nutzdaten reduziert und dann mit meinen vier Quellen abgeglichen. MSC liefert für das X.1-Protokoll eine Grafik, in welcher der Paketaufbau sehr gut beschrieben ist (Abb. 1).

Offset	Length	Contents and meaning
0	1	STX (0x02) Marks the beginning of a Fish.X1 data packet with the following byte
1	1	0x55
2	2	Length (H,L) of a data packet Consists of the header length (20 bytes) and the payload length.
4	4	Bus address, 4 bytes (L,H), <FROM> (data packet transmitter) Possible values: BUS_ADR_WILDCARD=1, BUS_ADR_MASTER=2, BUS_ADR_SLAVE1=3 ... BUS_ADR_SLAVE8=10
8	4	Bus address, 4 bytes (L,H), <TO> (data packet receiver) Possible values, see above
12	2	Transaction ID (TID), 2 bytes (L,H) Each data packet is marked by the transmitter with a sequential number; the response to a request is returned by the message receiver with the same TID.
14	2	Session ID (SID), 2 bytes (L,H) The ROBO TX interface internally manages a session ID, which is reassigned every time there is a change in task from local to remote or remote to local. For example, the session ID is reassigned by the firmware if a timeout occurs during communication. The transmitter can then respond accordingly.
16	4	Fish.X1 command code (tele-code), 4 bytes (L,H) Defines a unique request or reply code. The command code thus describes the supplied data structure in the payload.
20	4	Number of subsequent Fish.X1 data structures, 4 bytes (L,H). (= n)
24	4	Transfer area ID, 4 bytes (L,H) The ID specifies for which ROBO TX Controller, and thus for which transfer area subsection, the subsequent data structure is addressed. The data structure specified via the command code is updated from the individual transfer area. The interfaces can be configured as master (always local) or as slaves (to a master). A corresponding ID is specified here via the transfer area ID: TA_LOCAL=0, TA_EXT_1=1, TA_EXT_2= 2, etc. The master (TA_LOCAL) manages the IO values of the individual slaves in its transfer area and can thereby assign the current IO data using the share memory ID.
28	m	Message payload, dependent upon the transmitted data structure. The payload, together with the share memory ID, can also have a length of 0, typically a request or a reply to a request. The structures defined in the header file ROBO_TX_FW.H are set as the payload. The command code identifies the supplied data structure.
24 + n*(4+m)	2	CRC, 2 bytes (H,L) The CRC is calculated from byte 2 (data length) to the end of the payload.
24 + 2 n*(4+m)	1	ETX (0x03) Marks the end of a Fish.X1 message

Abb. 2: Erläuterungen (Quelle: MSC)

Die Struktur der Pakete ist relativ einfach:

- Jedes Paket beginnt mit einem Präambel bestehend aus zwei Bytes (0x02 und 0x55)
- Jedes Paket endet mit 0x03
- Jedes Paket besteht aus Präambel, Länge, Header, Daten, einer Prüfsumme und dem Schluss
- Die Länge ist die Summe der Anzahl Bytes im Header (20) und der Anzahl Bytes in den Daten
- Der Header enthält Sender, Empfänger, die Transaction-ID, die Session-ID, den Command Code und die Anzahl übertragener Datenblöcke
- Die Sender- und Empfängeradresse sind bei der Übertragung via USB fest; bei Paketen vom PC zum TX-C ist der Sender 2 und der Empfänger 1
- Die Transaction-ID erhöht sich bei jedem gesendeten Paket
- Die Session-ID kann am Anfang auf 0 gesetzt werden, bis der TX-C eine neue Session-ID setzt
- Die Anzahl der übertragenen Datenblöcke gibt an, an wie viele TX-Cs Daten übermittelt werden
- Die Prüfsumme ist laut Spezifikation eine CRC-Prüfsumme, die aus allen Bytes von der Länge bis zum Ende des Datenpakets berechnet wird

Der Command Code war das größte Problem. Dieser beschreibt, was hier überhaupt übertragen wird und somit, wie die Struktur in den Daten aussieht. In den offiziellen Quellen wird dies nicht beschrieben, und in den Analysen zum Erweiterungsbus fehlen leider einige benötigte Codes. Beim Senden ist der Code eine Zahl (hier: 1, 2, 5, 6, 7) und in umgekehrter Richtung ist es diese Zahl plus 100.

Die Daten selbst bestehen außer bei I²C aus einem Block pro angesteuerter Extension. Innerhalb eines Blocks kommen erst die Transfer Area ID und dann die Nutzdaten. Die Transfer Area ID ist für den Master 0 und zählt dann hoch bis 8 für die achte Extension. Der Aufbau der Nutzdaten ist bei jedem Command Code unterschiedlich, darauf gehe ich gleich ein.

Schließlich konnte ich die Datenpakete genauer analysieren und für einige Pakete herausfinden, wie die Daten aufgebaut sind. Wie auch von Thomas Kaiser festgestellt, ist die Prüfsumme kein CRC, sondern eine

Subtraktion aller Bytes von der Länge bis zum Ende der Daten von der Zahl 0x00.

Leider konnte ich nicht alles identifizieren, weshalb ich fischertechnik um Mithilfe bat. fischertechnik stellte mir alle erforderlichen Daten zur Verfügung – ein großes Dankeschön an Herrn Knecht, ohne den diese Herausforderung nicht zu meistern gewesen wäre. Leider waren auch diese Daten nicht zu 100% aktuell, aber irgendwann konnte ich das Puzzle lösen. Jetzt wusste ich, wann welches Paket mit welchem Command Code gesendet werden muss und wie die Daten darin aufgebaut sein müssen.

Command Codes

Durch die Analyse der Daten, die zwischen ROBOPro und TX-C ausgetauscht werden, fand ich heraus, dass die Command Codes 1, 2, 5, 6, 7, 101, 102, 105, 106 und 107 genutzt werden. Folgende Übersicht beschreibt die Bedeutung dieser Codes:

Echo (1)

- Enthält keine Daten
- Wird beim Öffnen der Verbindung gesendet

Echo Reply (101)

- Antwort auf Echo
- Enthält keine Daten
- Wird ignoriert

Remote IO (2)

- Sendet Ausgangsdaten und Daten zur Steuerung der Zähler an die TX-Cs und dient der Abfrage von Eingangsdaten
- Sendet einen Datenblock an den Master und jede verbundene Extension
- Die Counter Reset ID wird bei jedem Zurücksetzten des Zählers auf 0 um eins erhöht
- In das Feld Motor Sync wird die ID des Motors geschrieben, mit dem synchronisiert wird (0: Synchronisierung deaktiviert)
- Die Motor Command ID muss bei jeder Änderung der Entfernung oder des Synchronisationspartners für einen Motor um eins erhöht werden; anschließend bewegt sich der Motor erneut um die vorgegebene Strecke

	4 Bytes	
0	Transfer Area ID	
4	Counter Reset ID 1	Counter Reset ID 2
8	Counter Reset ID 3	Counter Reset ID 4
12	Motor Sync 1 / Motor Sync 2	Motor Sync 3 / Motor Sync 4
16	Output Duty 1	Output Duty 2
20	Output Duty 3	Output Duty 4
24	Output Duty 5	Output Duty 6
28	Output Duty 7	Output Duty 8
32	Motor Distance 1	Motor Distance 2
36	Motor Distance 3	Motor Distance 4
40	Motor Command ID 1	Motor Command ID 2
44	Motor Command ID 3	Motor Command ID 4

Abb. 3: Byteschema Remote IO Send

Remote IO Reply (102)

- Antwort auf Remote IO
- Enthält die Eingangswerte, Informationen der Zähler und Informationen zur erweiterten Motorsteuerung
- Enthält einen Datenblock an den Master und jede verbundene Extension
- Die Felder Input Value enthalten den fertig berechneten Eingangswert. Es muss keine Umwandlung für die verschiedenen Modi durchgeführt werden
- In den Feldern Counter State ist gespeichert, ob der entsprechende Zählereingang geöffnet (0x01) oder geschlossen (0x00) ist
- Counter Count ist der Wert des Zählers
- Aus der Counter Reset ID des zuletzt fertiggestellten Zählerresets, ergibt sich, ob der letzte Reset bereits durchgeführt wurde
- Aus der Motor Command ID des zuletzt fertiggestellten Motorkommandos ergibt sich, ob das letzte Kommando bereits fertiggestellt wurde

	4 Bytes			
	Transfer Area ID			
0	Input Value 1		Input Value 2	
4	Input Value 3		Input Value 4	
8	Input Value 5		Input Value 6	
12	Input Value 7		Input Value 8	
16	Counter 1 State	Counter 2 State	Counter 3 State	Counter 4 State
20	Counter 1 Count		Counter 2 Count	
24	Counter 3 Count		Counter 4 Count	
28	Counter Reset ID 1		Counter Reset ID 2	
32	Counter Reset ID 3		Counter Reset ID 4	
36	Motor Command ID 1		Motor Command ID 2	
40	Motor Command ID 3		Motor Command ID 4	
44	Empty			

Abb. 4: Byteschema Remote IO Reply

Remote Config Write (5)

- Senden von der Eingangskonfiguration an die TX-Cs
- Sendet einen Datenblock an den Master und jede verbundene Extension
- In die Felder Input Config wird je nach Modus ein Byte geschrieben. Soll ein Analogwert gemessen werden, so muss der Wert mit 128 multipliziert werden:

Modus	Digital	Analog
Spannung	0x00	0x80
Widerstand 5k	0x01	0x81
Widerstand 15k	0x02	0x82
Ultraschall	0x03	0x83

	4 Bytes			
0	Transfer Area ID			
4	0x01	0x01	0x01	0x01
8	Input 1 Config	Input 2 Config	Input 3 Config	Input 4 Config
12	Input 5 Config	Input 6 Config	Input 7 Config	Input 8 Config
16	0x01	0x01	0x01	0x01
20	0x00	0x00	0x00	0x00
24	0x00	0x00	0x00	0x00
28	0x00	0x00	0x00	0x00
32	0x00	0x00	0x00	0x00

Abb. 5: Byteschema Remote Config Write

Remote Config Write Reply (105)

- Antwort auf Remote Config Write
- Enthält keine Daten
- Wird ignoriert

Info (6)

- Abfrage von Metadaten von Master und Extensionen
- Sendet einen leeren Datenblock an den Master und jede verbundene Extension

Info Reply (106)

- Antwort auf Info
- Enthält für den Master und jede angeschlossene Extension einen Datenblock
- Bei der Bluetooth MAC-Adresse sind die Doppelpunkte bereits in den Daten enthalten

	4 Bytes		
0	Transfer Area ID		
4	Name des TX-Controller		
8			
12			
16			
20	Name des TX-Controller	Bluetooth MAC-Adresse	
24	Bluetooth MAC-Adresse		
28			
32			
36	Bluetooth MAC-Adresse	Empty	
40	Empty		
44			
48			
52			
56	Empty	Version 1	Version 2
60	Empty		
64			

Abb. 6: Byteschema Info Write

State (7)

- Abfrage nach verbundener Extensionen
- Sendet einen leeren Datenblock an den Master

State Reply(107)

- Antwort auf State
- Enthält einen Datenblock vom Master
- Die 8 Bytes Extension State geben an, ob die entsprechende Extension verbunden ist. Ist das Byte 0, so ist diese getrennt; ist es 1, so ist diese Extension verbunden

	4 Bytes			
0	Transfer Area ID			
4	Extension 1 State	Extension 2 State	Extension 3 State	Extension 4 State
8	Extension 5 State	Extension 6 State	Extension 7 State	Extension 8 State
12	Empty			
16				
20				
24				

Abb. 7: Byteschema State Reply

Software

Die Entscheidung, die Software in Python zu entwickeln, war eigentlich schon vorher klar, da die Community Firmware von Anfang an auf Python gesetzt hat und ich die Kompatibilität dazu wahren wollte. Um die Ein- und Ausgänge des TXT-Controllers anzusteuern hat Torsten Stuehn die Bibliothek *ftrobopy* geschrieben. Um auch hier möglichst kompatibel zu bleiben hatte ich mir vorgenommen, auf der höchsten Abstraktionsebene die Funktionsaufrufe gleich zu gestalten. In Anlehnung an *ftrobopy* habe ich die Software *fttxpy* getauft.

Abstraktionsebenen

Um den Überblick nicht zu verlieren habe ich die Software in fünf Abstraktionsebenen unterteilt:

TXSerial: Hier werden die serielle Verbindung und das X.1-Protokoll abstrahiert. Nach dem Aufbau der Verbindung können hier Pakete gesendet und Empfangen werden. Die Informationen werden in einem einheitlichen Datenformat gespeichert, welches Sender, Empfänger, Command Code und die Datenblöcke für den Master und die Extensionen enthält.

ftTX: Hier werden die Daten aller Ein- und Ausgänge verwaltet. Ein Hintergrundprozess kümmert sich um das Aufrechterhalten der Verbindung, indem möglichst dauerhaft alle Ausgangsdaten an den TX-C gesendet und alle Eingangsdaten eingelesen werden. Falls eine Eingangskonfiguration geändert wurde, wird diese automatisch an den TX-C übertragen. Des Weiteren werden hier die verbundenen Extensionen verwaltet, um zum Beispiel die Firmwareversion und den Namen der Extension auslesen zu können.

Um die gespeicherten Daten leichter editieren zu können, werden hier Funktionen bereitgestellt, mit welchen beispielsweise die Motorgeschwindigkeit gesetzt oder ein Eingang konfiguriert werden kann. Diese Funktionen werden in den folgenden Ebenen genutzt und benutzerfreundlicher bereitgestellt.

fttxpy: Dies ist die erste Ebene, mit der der Benutzer in Berührung kommt; sie stellt den Verbund aus dem Master und den Extensionen dar. Um die Bibliothek zu benutzen ruft der Benutzer diese Klasse auf. Dabei wird automatisch der erste verbundene TX-C ausgewählt. Nach dem Aufruf der Klasse werden automatisch der Name und die Firmwareversionen des Masters sowie aller verbundener Extensionen ausgegeben.

robotx: Diese Klasse stellt einen einzelnen TX-C im Verbund dar. Hier stehen Klassen zur Abstraktion verschiedener Peripheriegeräte wie Motoren, Lampen, Tastern und anderen Sensoren bereit. Die Funktion *robotx* wird aufgerufen, um einen bestimmten TX-C aus dem Verbund anzusprechen. Ohne Argumente erhält man ein Objekt des Masters, mit einer Zahl als Argument erhält man die entsprechende Extension.

Die *Sensoren* und *Aktoren*: Die wohl wichtigsten Klassen sind für die verschiedenen Sensoren und Aktoren verantwortlich. Eine genaue Beschreibung zu diesen Klassen erfolgt weiter unten.

Programmierung

Zur Benutzung der Software braucht ihr einen Rechner mit Linux und Python 3 oder einen TXT-Controller mit der Community Firmware. Auf der CFW ist *fttxpy* bereits vorinstalliert, auf anderen Geräten muss man sich *fttxpy* herunterladen und dann wie jedes andere Python Modul installieren. Dazu gibt man ein:

```
python3 setup.py install
```

Für die ersten Versuche öffnet man am besten den interaktiven Python Interpreter. Um die Bibliothek zu importieren und die Verbindung aufzubauen gibt man folgendes ein:

```
from fttxpy import fttxpy
tx = fttxpy()
```

Wenn nur ein TX-C ohne Extensionen verbunden ist, so steht dort zum Beispiel:

```
Found 1 TX-Controller(s).
Automatically selecting the first
one!
Connected to ROBO TX-308 Version
1.30
```

Sind Extensionen verbunden, dann steht dort zusätzlich:

```
Found Extensions: Ext 1 (ROBO TX-
309, 1.30), Ext 2 (ROBO TX-309,
1.30)
```

Um nun auf den Master oder eine Extension zuzugreifen gibt man folgendes ein:

```
master = tx.robotx()
ext1 = tx.robotx(1)
ext2 = tx.robotx(2)
...
```

Die Objekte `master, ext1, …` bieten nun die Klassen zur Ansteuerung der Ein- und Ausgänge. Für jeden TX-C können mehrere Objekte erstellt werden, für die Ein- und Ausgänge jedoch nicht. Wenn man ein Ein- bzw. Ausgangsobjekt nicht mehr benötigt, so muss dieses mit `del(<Objekt>)` gelöscht werden. Danach kann ein neues Objekt erstellt werden.

Motor

Ein Motor kann viel mehr, als viele denken. Er kann nicht nur in der Richtung und Geschwindigkeit gesteuert werden, sondern auch definierte Distanzen fahren und zu einem anderen Motor synchronisiert werden.

Zunächst möchten wir einen Motor nur in Richtung und Geschwindigkeit variieren:

```
# Zunaechst erzeugen wir ein
Motorobjekt
m1 = master.motor(1)
# warten bis Nutzer Enter drueckt
input()
# Geschwindigkeit 128 rechts
m1.setSpeed(128)
input()
# Geschwindigkeit 512 rechts
m1.setSpeed(512)
input()
# Geschwindigkeit 128 links
m1.setSpeed(-128)
input()
# Geschwindigkeit 512 links
m1.setSpeed(-512)
input()
# Stop
m1.setSpeed(0)
# Oder
m1.stop()
```

Eine vorgegebene Distanz zu fahren und dann automatisch zu stoppen ist auch ganz einfach:

```
# Distanz auf 1000 setzen
m1.setDistance(1000)
# Geschwindigkeit auf 300
m1.setSpeed(300)
# Abfragen, ob Position erreicht
ist
while not m1.finished():
      pass
# Geschwindigkeit und Distanz auf
0 setzen
m1.stop()
print("Fertig")
```

Die Abfrage `finished()` gibt `True` oder `False` zurück. Wenn die Position erreicht ist, so ist der Wert `True`, andernfalls `False`.

Der Befehl `stop()` setzt Geschwindigkeit und Distanz auf 0. Will man den Motor, bevor er sein Ziel erreicht hat, anhalten und dann wieder weiterfahren lassen, ohne diese Distanzmessung zu unterbrechen, sollte man `setSpeed(0)` verwenden.

Die Kunst in der Robotik ist es, einen fahrenden Roboter exakt geradeaus fahren zu lassen. Dies ist auch mit der `setDistance`-Funktion möglich:

```
# Zunaechst erzeugen wir zwei
Motorobjekte
m1 = master.motor(1)
m2 = master.motor(2)
# Distanz für M1: 1000
m1.setDistance(1000)
# Distanz für M2: 1000
# M2 zu M1 synchronisieren
m2.setDistance(1000, m1)
# Geschwindigkeit beide 512
m1.setSpeed(512)
m2.setSpeed(512)
# Warten auf Position erreicht
while not m1.finished() and not
m2.finished():
      pass
print("Fertig")
m1.stop()
```

```
m2.stop()
```

Der Befehl `stop()` deaktiviert die Synchronisierung. Will man die Motoren erneut synchronisieren, so muss man mit `setDistance()` erneut den Synchronisationspartner übergeben. Wie auch in ROBOPro kann ein Motor nur zu einem anderen Motor am gleichen TX-C synchronisiert werden.

Ausgang

Um zum Beispiel Lampen, Magnetventile oder Summer anzusteuern genügt ein ein „O“-Ausgang. Dieser lässt sich nur in der Intensität, aber nicht in der Richtung steuern:

```
# Ausgangsobjekt erzeugen
o5 = master.output(5)
# Intensitaet 128
o5.setLevel(128)
input()
# Intensitaet 255
o5.setLevel(255)
input()
# Ausschalten
o5.setLevel(0)
input()
```

Taster

Der wohl wichtigste, aber auch simpelste Sensor ist der Taster. So wird er eingelesen:

```
from time import sleep
# Eingangsobjekt generieren
i1 = master.input(1)
while True:
      # Taster ein Mal pro Sekunde
einlesen und ausgeben
      print(i1.state())
      sleep(1)
```

Widerstände

Der Helligkeitssensor und der NTC-Temperatursensor sind Widerstände. Dieses können sehr einfach ausgewertet werden. Zudem kann die Temperatur des NTC-Temperatursensors in Grad Celsius umgewandelt werden:

```
# Eingangsobjekt generieren
i2 = master.resistor(2)
# Rohwert auslesen
print(i2.value())
# Temperatur in Grad Celsius
ausgeben
print(str(i2.ntcTemperature()),
"Grad Celsius")
```

Spannung

Will man nun einen Farbsensor verwenden oder die Akkuspannung prüfen geht man wie folgt vor:

```
# Eingangsobjekt generieren
i3 = master.colorsensor(3)
# Spannung in Millivolt ausgeben
print(str(i3.voltage()), "mV")
# Farbe ausgeben - experimentell!
print(i3.color())
```

Der Aufruf `color()` ist sehr experimentell. Ich habe die Werte nur aus *ftrobopy* kopiert. Ob die Werte auch bei euch funktionieren, weiß ich nicht. Aufgabe für die Profis: Schreibt euch eine Funktion um die Farbe unter euren Bedingungen auszugeben.

Spursensor

Beim RoboCup Junior muss ein Roboter in der Disziplin Rescue Line u. a. einer Linie folgen. fischertechnik bietet dazu den Spursensor an. Dieser liefert am Ausgang 0V oder 9V. Die Auswertung ist ganz einfach:

```
# Eingangsobjekt generieren
i4 = master.trailfollower(4)
# Aktuellen Zustand ausgeben
print(i4.state())
```

Um die Spur auszuwerten muss man natürlich zwei Eingangsobjekte für die beiden Eingänge erzeugen und mit ein wenig Logik auswerten.

Distanzsensor

Will man eine Distanz mit dem Ultraschall-Abstandssensor von fischertechnik messen, muss man wie folgt vorgehen:

```
# Eingangsobjekt generieren
i5 = master.ultrasonic(5)
# Aktuelle Distanz ausgeben
print(i5.distance())
```

Schlusswort

Ich weiß nicht, für welchen Anwendungszweck das X.1-Protokoll ursprünglich entwickelt wurde, aber für eine möglichst schnelle Datenübertragung zwischen einem PC und einem TX mit bis zu neun Erweiterungen ist es nicht geeignet. An vielen Stellen wurde sehr viel Platz verschwendet, angefangen bei der Sender- und Empfängeradresse, die mit vier Byte völlig überdimensioniert ist (vier Bit hätten genügt), über den Command Code mit vier Bytes (es gibt noch ein paar Codes mehr, als hier dargestellt, aber ein Byte hätte ausgereicht) und schließlich der Anzahl Datenstrukturen mit vier Bytes (ein Master und acht Erweiterungen, also maximal neun Datenstrukturen; dafür wären fünf Bit ausreichend gewesen) sieht man, wie ineffizient alleine der Paketheader ist, der bei jedem Paket übertragen werden muss. An vielen Stellen in den Paketen hätte man auch viele ungenutzte Bytes entfernen können. Das von fischertechnik versprochene 10 ms-Raster ist mit nur fünf Erweiterungen nicht mehr zu halten. Hier sind pro Buszyklus 14 ms erforderlich [2]; Für acht Erweiterungen benötigt das Protokoll 22,4 ms.

Bei der Analyse ist mir irgendwann aufgefallen, dass man die Transaction-ID frei wählen kann. Hier darf auch während der Verbindung frei „gesprungen“ werden, also hätte man diese zwei Bytes auch einsparen können.

Egal, ob man den Distanzsensor als digital oder analog abruft, Werte liefert er immer, und wenn man den Sensor während der Verbindung trennt, bleibt der alte Wert erhalten.

Quellen

[1] fischertechnik GmbH: *Zusammenstellung der wichtigsten Dateien*.

[2] Ad van der Weiden: *Logic analyser trace of the RS485 communication between a Master and 5 Slaves*.