XBee ZB Lib in Python3.2 (für XBee ZB Module im API-Modus)

Bislang sind meine Router und der Coordinator an ATMega's angeschlossen und die Software ist in C geschrieben. Zumindest der Coordinator soll in Zukunft direkt an der serielle Schnittstelle eines PC's oder RaspberryPi's seinen Dienst tun, damit dort Daten optisch ansprechend dargestellt und eine Benutzerschnittstelle realisiert werden kann.

Aus diesem Grund habe ich vorhandene C-Routinen in Python Objekte umstrukturiert und einige zusätzlich Features eingebaut, die den ATMega vermutlich überfordern würden.

Die beigefügten Module stellen zunächst das Grundgerüst für die Kommunikation zwischen XBee und PC im API-Modus zur Verfügung und erlauben
- das Empfangen von Frames in einem Hintergrundthread,
- das Versenden von lokalen und remote AT-Frames sowie von Datenframes.

Darüber hinaus ist eine "Sendungsverfolgung" entstanden.
Zur Kontrolle der Transaktionen wird für jedes vom Coordinator versandte Frame ein programminterner Listeneintrag erstellt, in dem die fortlaufende FrameId, der FrameTyp, die Id der Gegenstelle, der Sendezeitpunkt und die Referenz auf eine Funktion (rxHandler) gespeichert sind.
Sobald die zugehörige Rückmeldung bzw. Quittung eintrifft, wird die an den Eintrag gebundene Funktion ausgeführt - und der Eintrag selbst wieder gelöscht.
Dem rxHandler wird die Id der sendenden Station und der Inhalt des zurückgelieferten Frames übergeben, der Anwender kann damit eine geeignete Reaktion ausführen.

Mit anderen Worten:
Beim Versenden eines Frames wird eine Funktion referenziert, die für die Bearbeitung der Antwort auf die Transaktion zuständig ist.
Frames können versandt werden, ohne dass man sich um die Zuordnung der Rückmeldungen kümmern muss.
Die Rückmeldungen dürfen sogar in beliebiger Reihenfolge und mit zeitlicher Verzögerung eintreffen.

Vor dem Versenden eines neuen Frames kann die oben erwähnte Liste daraufhin befragt werden, ob von der Zielstation noch Rückmeldungen ausstehen.
Diese Information kann dazu genutzt werden, dass an nicht antwortende Stationen (vorerst) keine weiteren Nachrichten versandt werden.

Über einen längeren Zeittraum nicht quittierte Sendungen können aus der Liste gelöscht werden.
Vor dem Löschen wird wieder der rxHandler aufgerufen und ein Fehlercode übergeben.
Der Anwender kann den Code auswerten und eine angemessene Reaktion ausführen.

Schließlich ist noch die Option eingebaut, zeitverzögert bzw. in regelmäßigen zeitlichen Abständen Funktionen (wie z.B. das Absenden von Frames) ausführen zu lassen.

Eine detailliertere Beschreibung folgt im Anschluss.

Wie funktioniert das praktisch ?

Innerhalb der Anwendung wird eine Instanz der Klasse Xbee() eingefügt.
Diese Klasse lädt für spezielle Aufgaben weitere Klassen nach, Xbee_rx(), Xbee_adr(), Xbee_job() und Xbee_frame(). Der Anwender selbst arbeitet stets mit der Klasse Xbee().

Xbee()

Die Xbee-Klasse formuliert API-Frames, versieht sie mit einer Prüfsumme, verteilt dabei fortlaufende FrameId's, aktualisiert die 16-Bit-Netzwerkadressen, zählt aus Langeweile empfangene, versendete und fehlerhafte Übertragungen mit, registriert die Signalstärke der empfangenen Frames und führt bei Bedarf Protokoll via TTY und/oder File.
Die Klasse Xbee() wird mit folgenden Parametern initialisiert (die Parameter in [...] sind optional und vorbelegt wie beschrieben) :

```
class Xbee( ser [, callback = None, logToTTY = False, logToFile = False,
                                    readRSSI = False, filename = None, checkCTS = False] )
```

Die Übergabe der Seriellen Schnittstelle (ser), an der das XBee-Modem angeschlossen ist, ist zwingend notwendig.
Die Callback-Funktion muss 2 Parameter entgegennehmen können:

```
def callback( station, frame )
- station   = Index der sendenden Station
- frame     = kompletter Frame als Bytestring (ohne abschließendes Prüfsummenbyte)

Der FrameTyp des Frames kann aus dem Bytestring 'frame' extrahiert werden:
- frametyp = frame[3]
```

Wenn alle ein- und ausgehende Frames auf dem Bildschirm angezeigt werden sollen, dann muss logToTTY auf True gesetzt werden.
Um die Daten (auch) in eine Datei zu schreiben, muss logToFile auf True gesetzt werden.
Allerdings muss vorher der Name eines Logfiles angegeben werden.
Ein leerer Dateiname "" erstellt einen Namen aus der Zeit in Sekunden mit angehängter Dateierweiterung ".log".
Der Parameter checkCTS legt fest, ob vor dem Versenden von Frames die Empfangsbereitschaft des Modems geprüft werden soll (das Modem ist bereit, wenn CTS = low).

Die Methoden der Klasse Xbee() sind:

```
sendLocalAtCmd( atcmd [,data = [], rxHandler = __doNix, frameId = True] )
    sendet ein lokales AT-Kommando
    - frameId = True    - sendet fortlaufende FrameId's von 1 .. 255
    - frameId = False   - sendet konstante FrameId = 0
sendRemoteAtCmd(dev, atcmd [,data = [] rxHandler = __doNix, frameId = True,
                                                            RemoteCmdOptions=2] )
    sendet ein remote AT-Kommando
sendDataFrame( dev [, data = [], rxHandler = __doNix, frameId = True, Options = 0,
                                                            BroadcastRadius = 0] )
    sendet ein Daten-Frame
config( [logToTTY = None, logToFile = None, readRSSI = None] )
    erlaubt die Änderung von Einstellungen
closeLogFile()
    schließt ein geöffnetes Logfile
exit()

- die nachfolgenden Methoden werden an Xbee_adr() weitergereicht und dort ausgeführt:
getMyAdr()
    ermittelt die Seriennummer, den Namen und die Id des lokalen XBee-Modules
    und trägt sie in Klassenvariablen von Xbee() ein.
getStation( [idx = None, name = None, adr64 = None, adr16 = None] )
    liefert durch Angabe eines der 4 Parameter die Daten der Station als Tuple zurück

- die nachfolgenden Methoden werden an Xbee_job() durchgereicht
getJobByStation( Id )
    gibt zurück, wie viele offene Jobs für Station 'Id' verzeichnet sind und wie lange
    der älteste Eintrag wartet (in Sekunden).
cleanJobs( [delay = 180] )
    löscht alle Jobs, die älter als 180 Sekunden sind
```

Die Auswahl der Gegenstelle (dev) kann wahlweise über eine Ganzzahl (= Index der Stationsliste) oder einen String (= Name der Station innerhalb der Stationsliste) erfolgen.
Die Auswahl über den Index ist allerdings programmtechnisch der einfachere Weg.

Xbee_adr()

Xbee_adr() ist für die Verwaltung der 64-Bit Seriennummer und die Aktualisierung der 16-Bit Netzwerkadressen der beteiligten Stationen zuständig.
Von hier holt Xbee() für zu sendende Frames die Seriennummer und die Netzwerkadresse der Zielstation.

Bei der Initialisierung werden aus der Datei "xbee.cfg" die 64-Bit Seriennummern aller beteiligten Devices eingelesen. Jedem Device kann ein (eindeutiger) Name vergeben werden.

Der Sinn der Übung ist, dass im späteren Programmablauf jede Station über eine Zahl (nämlich den Index innerhalb der Stationsliste) oder über einen Namen (den oben vergebenen) angesprochen werden kann.
Die Kenntnis der 64-Bit Adresse und der vom Coordinator vergebenen 16-Bit Netzwerkadresse ist nicht mehr erforderlich.

Die 16-Bit Netzwerkadressen lauten beim Programmstart "FFFE", bei jedem eingehenden Frame wird geprüft, ob die übermittelte Netzwerkadresse mit dem gespeicherten Eintrag übereinstimmt.
Falls nicht, wird die Netzwerkadresse aktualisiert.
Fällt eine Station aus, dann wird ihre Netzwerkadresse wieder auf "FFFE" geändert.

Das Format der Datei "xbee.cfg" sieht so aus:

```
0x00,0x13,0xA2,0x00,0x40,0x33,0x7B,0xB7,RaspberryPi,
0x00,0x13,0xA2,0x00,0x40,0x62,0x8D,0x12,Station 3,
0x00,0x13,0xA2,0x00,0x40,0x32,0x09,0x3B,Station 4,
0x00,0x13,0xA2,0x00,0x40,0x32,0x09,0x8B,Station 5,
0x00,0x13,0xA2,0x00,0x40,0x70,0xF4,0x7F,Station 6,
0x00,0x13,0xA2,0x00,0x40,0x64,0xF6,0xA4,Station 7,
0x00,0x13,0xA2,0x00,0x40,0x6A,0x4B,0x3F,Station 8,
0x00,0x13,0xA2,0x00,0x40,0x52,0xA0,0xD8,PC,
0x00,0x13,0xA2,0x00,0x40,0x52,0xA0,0xA8,Station 9,
#
# Die default-Adressen für den Coordinator /Broadcast = Zeile 0 / 1
# werden automatisch in der Stationsliste angelegt,
# es sind an dieser Stelle nur die Adressen der vorhandene Devices einzutragen !
#
# die Zeilen werden solange eingelesen, bis keine Adresse mehr gefunden wird.
# Das Programm prüft dazu, ob die Zeile genau 9 Kommata enthält,
# wenn nicht, dann wird das Einlesen beendet.
```

Auf die Methoden der Klasse Xbee_adr() wird nicht direkt zugegriffen.
Da einige Methoden unter Umständen zur Adressauflösung und zur Ausgabe der Stationsliste benötigt werden, reicht Xbee() die Funktionsaufrufe an Xbee_adr() weiter.

```
printStationList( dev = -1] )
   Druckt Stationsdaten aus:
   wenn dev = -1: Liste aller Stationen
   wenn dev >= 0: Status dieser Station als Tuple
getStation( [idx = None, name = None, adr64 = None, adr16 = None] )
   sucht in der Adressliste nach dem übergebenen Wert und gibt ein Tuple zurück:
   (Index, 64-Bit Seriennummer, 16-Bit Netwerkadresse, Stationsname)
setSelfStation(idx)
   übergibt den Index der am eigenen Rechner angeschlossenen Station
```

Dies ist der Ausdruck der Stationsliste, wenn die oben beschriebene Datei "xbee.cfg" eingelesen wurde:

```
Idx 64-Bit-Adresse ------------ 16-Bit -- Name -------------- RSSI ------ RX ---- TX --- Err
0   00 00 00 00 00 00 00 00 --- 00 00 --- COORDINATOR         -             0       0       0
1   00 00 00 00 00 00 FF FF --- FF FE --- BROADCAST           -             0       0       0
2   00 13 A2 00 40 33 7B B7 *** 00 00 *** RaspberryPi         -00dB         0      12       1
3   00 13 A2 00 40 62 8D 12 --- FF FE --- Station 3           ?             0       0       0
4   00 13 A2 00 40 32 09 3B --- DF C3 --- Station 4           -56db        35       0       0
5   00 13 A2 00 40 32 09 8B --- FF FE --- Station 5           ?             0       0       0
6   00 13 A2 00 40 70 F4 7F --- D6 AD --- Station 6           -71db        61       0       0
7   00 13 A2 00 40 64 F6 A4 --- D1 BA --- Station 7           -45db        10       0       0
8   00 13 A2 00 40 6A 4B 3F --- FF FE --- Station 8           ?             0       0       0
9   00 13 A2 00 40 52 A0 D8 --- FF FE --- PC                  ?             0       0       0
10  00 13 A2 00 40 52 A0 A8 --- FF FE --- Station 9           ?             0       0       0
```

Die Zeile mit den "***" verweist auf die Station, die am lokalen Rechner angeschlossen ist.
Wenn readRSSI = True, dann wird die Empfangsstärke von eingehenden Frames registriert.
Die Anzahl der versendeten und empfangen Frames sowie die Anzahl der dabei fehlerhaften Übertragungen wird stationsweise protokolliert.
Die Klassenvariablen cnt_tx, cnt_rx, cnt_data und cnt_err speichern die Gesamtzahl der versendeten und der quittierten Frames, der empfangen sonstigen (Daten-)Frames und der insgesamt registrierten Übertragungsfehler.

Xbee_rx()

Die Klasse Xbee_rx() startet einen Hintergrundthread, der in regelmäßigen Abständen prüft, ob über die Serielle Schnittstelle Daten empfangen wurden.
Wenn ja, dann wird auf den Eingang des vollständigen Frames gewartet und anschließend die Prüfsumme getestet.
Danach werden die Daten per Callback an Xbee() zurückgegeben.

In der Klasse Xbee() werden die empfangenen Daten "vorgeprüft", es wird in Xbee_adr() nachgesehen, ob dort die 16-Bit Netzwerkadresse aktuell ist, Zähler für empfangene, gesendete und fehlerhafte Frames werden stationsweise incrementiert und die Stärke des empfangenen Signals ermittelt und stationsweise gespeichert.

Danach wird das empfangene Frame per Callback an das Hauptprogramm weitergereicht - es sei denn, ein rxHandler übernimmt die Regie. Aber dazu kommen wir später.

Die Klasse Xbee_rx() verfügt über keine Methoden.

Xbee_job()

Diese Klasse hat nichts mehr zu tun mit der Übertragung der Daten, sie verwaltet versendete Frames und sorgt für die Zuordnung von empfangenen Frames.
Das Modul vereinfacht die Behandlung von eingehenden Frames, die als Antwort auf lokale oder remote AT-Kommandos oder auf versandte Datenframes eingehen.

Zum Erklärung muss weiter ausgeholt werden.
Es gibt im wesentlichen zwei Arten von Nachrichten, die von den Modulen empfangen werden.
Die einen sind Datenframes vom Typ 0x90, die von anderen Station via Frame 0x10 versendet wurden.
Sie treffen zu unvorhersehbaren Zeitpunkten ein und müssen in main() behandelt werden.

Die Frames 0x88, 0x8B und 0x97 sind Antworten auf von der eigenen Station versandte Anfragen der Frames 0x08, 0x10 und 0x17.
Für jedes versandte Frame wird mindestens eine Quittung (beim Datenframe 0x10), bei den AT-Frames eine Quittung ggf. mit zusätzlichen Daten zurückgeliefert.
Die Quittungen treffen der Regel relativ prompt ein. Aber nicht ohne Verzögerung.
Und nicht unbedingt in der Reihenfolge, in der sie versandt wurden.

Wenn die Gegenstelle offline ist, dann dauert es einige Sekunden, bis eine Fehlermeldung eingeht.
Wenn gar ein schlafendes Enddevice adressiert wurde, dann können Minuten bis zum Eingang einer Antwort vergehen.

Damit wird deutlich, dass - nach dem Absenden eines Frames - nicht einfach pollend auf die Antwort gewartet werden kann:
Denn das als nächstes eingehende Frame ist möglicherweise nicht die erwartete Antwort - sondern ein zufällig eingehendes Frame.
Wie kann man aber sicherstellen, dass das "richtige" Frame einer Anfrage zugeordnet wird ?

Die gewählte Lösung verzichtet auf das Pollen und sieht wie folgt aus:

Die Klasse Xbee() lädt bei der Initialisierung die Klasse Xbee_job() nach.
Hier wird eine Liste aller versandten Nachrichten der Frames 0x08, 0x10 und 0x17 geführt.
Beim Aufruf der Methoden sendLocalAtCmd(), sendRemoteAtCmd() und sendDataFrame() kann als Parameter eine Funktion (rxHandler = ...) übergeben werden, von der die zurückgelieferten Daten bearbeitet werden sollen.
Fehlt der Parameter, dann wird per default die Methode __doNix() eingefügt.
Die macht ihrem Namen alle Ehre - und macht nur bei Übertragungsfehlern auf sich - bzw. den Fehler aufmerksam.

Objektintern wird im Array jobs[] ein Datensatz angelegt, in dem die FrameId des erwarteten Frames (die ist identisch mit der FrameId des gesendeten Frames), der Frametyp und eine Referenz auf die übergebene Funktion abgelegt werden.
Zusätzlich wird noch der Index der Gegenstelle, von der die Antwort erwartet wird sowie ein Zeiteintrag vorgenommen.

Alle eingehende Daten (der Frames 0x88, 0x8B und 0x97) werden nun daraufhin überprüft, ob innerhalb des Arrays jobs[] ein Datensatz mit der FrameId und dem Frametyp des eingegangen Frames vorhanden ist.
Wenn ja, dann wird die dort referenzierte Funktion mit der Id der sendenden Station und den Daten des empfangenen Frames als Parameter aufgerufen und der Eintrag in jobs[] gelöscht.

Was ist damit erreicht ?

Den Methoden, die ein Frame versenden, wird als Parameter eine Referenz auf eine für die Bearbeitung der Quittung zuständige Funktion mitgegeben.
Nach dem Motto "Senden und vergessen" werden nun Frames abgeschickt, ohne dass man sich selbst noch weiter um die Behandlung der Antwort kümmern muss.
Das Hauptprogramm kann sich sofort nach dem Absenden anderen Aufgabe widmen und blockiert nicht pollend in einer Warteschleife.

Die Jobliste kann befragt werden, ob eine Station Sendungen noch nicht beantwortet hat. GetJobsByStation liefert für eine Station zurück, wieviele Sendung noch "on air" sind und wie lange auf die älteste Sendung gewartet wird.
Aus diesen Informationen kann man seine Rückschlüsse ziehen: ggf. ist die Gegenstelle offline - und weitere Sendungen sollten unterbleiben.

Damit die Liste durch verwaiste Einträge nicht unnötig verlängert wird, sollten in regelmäßigen Zeitabständen veraltete (= innerhalb eines definierten Zeitraumes nicht beantwortete) Einträge gelöscht werden.
Im Normalfall werden fehlgeschlagene Übertragungen durch ein Frame "gemeldet" und sauber aus der Liste ausgetragen.
Nur in extremen Fällen bleiben Antworten aus oder treffen erst mit sehr großer zeitlicher Verzögerung ein.
Für das Löschen verwaister Einträge ist die Methode cleanJobs( ) zuständig.

Sie löscht per default alle Einträge, die älter als 180 Sekunden sind.
Schlafende Enddevices können von der Aufräumaktion ausgenommen werden, indem ihr Index innerhalb einer optionalen Liste als zusätzlicher Parameter übergeben wird.
Vor dem Löschen jedes Eintrages wird die referenzierte Funktion im rxHandler aufgerufen.
Da in diesem Falle kein empfangenes Frame vorliegt, wird als Parameter die StationsId und ein Tuple in folgendem Format übergeben: (0x5A, FrameId, Frametyp).
Die Konstante 0xA5 steht als erstes Byte und signalisiert einen Fehler (bei jedem fehlerfrei empfangenen Frame steht der "start delimiter" 0x7E im ersten Byte).
Die auswertende Funktion sollte daher frame[0] testen, wenn hier nicht 0x7E gefunden wird, dann liegt ein Fehler vor.

Verwaiste Einträge entstanden während meiner Tests, wenn in kurzer Folge viele Frames an nicht präsente Stationen geschickt wurden:
Hier wurden nur die ersten Frames mit einer Fehlermeldung quittiert, die Quittierung der nachfolgenden verzögert sich um mehrere Minute bzw. unterblieb vollständig.

Zu beachten sind mögliche Seiteneffekte.
Ein Eintrag im Array jobs[] wird gelöscht, sobald eine Nachricht mit passender FrameId /Frametyp eingegangen ist.
Nun gibt es Broadcastes und AT-Kommandos, die mehrere Stationen dazu veranlassen, eine Sendung an den Absender abzusetzen.
Hier wird nur die erste Nachricht berücksichtigt und "behandelt", die nachfolgend gehen als 'normal' empfangene Frames durch und werden an das Hauptprogramm weitergereicht.
Ergo: Bei Broadcasts und einigen AT-Kommandos wie "ND" kann dieser Mechanismus nicht funktionieren.

Weiterhin ist zu bedenken, dass bei Wahl des Parameters setFrameId = False (default ist setFrameId = True) die konstante FrameId '0' gesendet wird.
Was bedeutet, dass die Gegenstelle keine Quittung senden wird.
Ohne eine Rückmeldung ist aber keine Auswertung möglich.
Darum bleibt Xbee_job() bei setFrameId = False untätig und legt keinen Eintrag in jobs[] an.

Ein potentielles Problem ergibt sich die Tatsache, dass die FrameId nur von 1 bis 255 zählt.
Vergeht ein längerer Zeitraum bis zum Eingang der Rückmeldung und werden zwischenzeitlich mehr als 254 weitere Frames versandt, dann wiederholen sich die Id's und es könnten mehrere Einträge in jobs[] mit identischer FrameId und FrameTyp vorliegen. Dann wäre keine eindeutige Zuordnung mehr möglich.

Darum wird das Array jobs[] vom Anfang bis zum Ende nach der Id und dem Frametyp des empfangenen Frames durchsucht.
Wird dabei mehr als ein passender Eintrag gefunden, dann liegt ein Fehler vor (Überlauf der FrameId) und alle Einträge mit der gesuchten Id/Frametyp werden gelöscht.

Vor dem Löschen jedes Eintrages wird wie oben die referenzierte Funktion im rxHandler aufgerufen und dabei als Parameter die StationsId und ein Tuple in folgendem Format übergeben:
(0xA5, FrameId, Frametyp).

Das Auftreten des beschriebenen Problem ist im Normalbetrieb sehr unwahrscheinlich - aber nicht grundsätzlich auszuschließen.

Damit sich der Überlauf der FrameId so selten wie möglich ereignet, wird für jeden der drei versendenden Frametypen eine individuelle FrameId geführt.

Die Klasse Xbee_jobs() kennt folgende Methoden:

```
addJob( frameId, frametyp, dev, rxHandler )
    fügt einen neuen Job in die Jobliste ein.
checkJob( frameId, frametyp )
    liefert den Index des Eintrages in der Liste oder
    -1 wenn kein passender Eintrag gefunden wurde.
doJob( idx, frame )
    führt die Funktion aus, die in Joblist[idx] hinterlegt ist und übergibt ihr den kompletten
    Frameinhalt zur Auswertung.
getJobsByStation( station )
    prüft, wie viele Jobs für die genannte Station noch in Arbeit sind,
    gibt als Tuple zurück:
    - Anzahl der Jobs
    - Wartezeit des ältesten Jobs in Sekunden
cleanJobs( [ delay = 180, enddevices = [] ])
    entfernt verwaiste Jobs, die älter sind als delay in Sekunden,
    aber dann nicht, wenn die zugehörige Station in der Liste der Enddevices enthalten ist.
    Gibt eine Liste mit den Indizes der zugehörigen Station zurück.
printJobs()
    druckt eine Liste aller noch wartenden Jobs.
```

Eingehenden Frames der Typen 0x88, 0x8B und 0x97 werden nicht an die Callback-Funktion von main() weitergegeben ! In den Protokolldaten werden sie aber aufgeführt.
Erscheinen diese Frames wider Erwarten dennoch in main(), dann muss der Eintrag im Array Jobs[] gelöscht worden sein.
Ursache kann sein, dass der Eintrag wegen Zeitüberschreitung aufgeräumt wurde oder ein Überlauf in der FrameId zum Löschen mehrdeutiger Einträge geführt hat.
Oder dass ein Broadcast oder AT-Kommando ausgeführt wurde, auf das hin mehr als eine Station antwortet.

rxHandler(station, frame)

Einige Anmerkungen zur Rückgabe des rxHandlers.
Dem rxHandler wird immer die Id der die Quittung sendenden Station sowie ein Iterable als Parameter übergeben.
Im Normalfall handelt es sich bei dem Iterable um den Inhalt des empfangenen Frame als Bytestring ohne abschließendes Prüfsummenbyte.
Zuerst sollte der rxHandler das erste Byte des Bytestrings prüfen.
Steht hier der "Start Delimiter" 0x7E, dann ist wurde ein fehlerfreies Frame empfangen.
In diesem Frame sollte nun das Byte "Command Status" überprüft werden.
Ein Wert von "0" deutet auf Erfolg hin, jeder andere Wert ist ein Hinweis auf einen Übertragungsfehler (Fehlercodes siehe Usermanual).

Steht in frame[0] nicht der "Start Delimiter", dann ist das ebenfalls ein schlechtes Zeichen.

Der Wert 0xA5 weist darauf hin, dass in der Jobliste mehrere identische Einträge enthalten waren, für jeden Eintrag wird eine separate Fehlermeldung ausgegeben. Ursache könnte ein Überlauf der FrameId's sein.

Der Wert 0x5A weist darauf hin, dass der Eintrag wegen Zeitüberschreitung gerade aus der Jobliste entfernt wird.
Wenn für eine Transaktion der Eintrag in der Jobliste entfernt worden ist - und dann wider Erwarten doch noch eine Rückmeldung eintrifft, dann kann dieser Meldung kein rxHandler mehr zugeordnet werden.
Die Meldung wird dann genauso behandelt wie eingehende Frames 0x90.
Sie wird per callback() an main() durchgereicht, ist dort über den Frametyp als "verwaister" Eintrag identifizierbar.

Ein Test zeigt, wie die rxHandler arbeiten (siehe main_e.py)
Die nachfolgende Schleife versendet 18 Frames, es werden lokale und remote AT Kommandos sowie Datenframes verschickt. Die einzige eingebaute "Bremse" ist das Handshake mit der Abfrage von CTS.
Für jeden Frametyp wird eine eigene Funktion festgelegt, die die Rückmeldungen bearbeiten soll.

```
for i in ("SL", "SH", "MY", "D0", "D1", "D3", "D4", "D5", "D6" ):
    self.X.sendLocalAtCmd( atcmd = i, rxHandler = self.func88 )
    self.X.sendRemoteAtCmd( atcmd = i, dev = 9, rxHandler = self.func97 )
    self.X.sendDataFrame( dev = 10, data = [1, 2, 3], rxHandler = self.func8B )
```

Es folgt das Logfile der versendeten (<-) und empfangenen (->)Frames, es zeigt die Datenbytes der Frames (bei den empfangenen Frames fehlt jeweils als letztes Byte die Prüfsumme !)

```
<- 13:40:51.9 7E 00 04 08 01 53 4C 57
<- 13:40:51.9 7E 00 0F 17 01 00 13 A2 00 40 52 A0 D8 FF FE 02 53 4C 8A
<- 13:40:51.9 7E 00 11 10 01 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 5C
<- 13:40:51.9 7E 00 04 08 02 53 48 5A
<- 13:40:51.9 7E 00 0F 17 02 00 13 A2 00 40 52 A0 D8 FF FE 02 53 48 8D
<- 13:40:51.9 7E 00 11 10 02 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 5B
-> 13:40:51.9 7E 00 09 88 01 53 4C 00 40 33 7B B7
<- 13:40:51.9 7E 00 04 08 03 4D 59 4E
<- 13:40:51.9 7E 00 0F 17 03 00 13 A2 00 40 52 A0 D8 FF FE 02 4D 59 81
<- 13:40:51.9 7E 00 11 10 03 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 5A
<- 13:40:51.9 7E 00 04 08 04 44 30 7F
-> 13:40:51.9 7E 00 13 97 01 00 13 A2 00 40 52 A0 D8 87 BD 53 4C 00 40 52 A0 D8
<- 13:40:51.0 7E 00 0F 17 04 00 13 A2 00 40 52 A0 D8 FF FE 02 44 30 B2
-> 13:40:51.0 7E 00 09 88 02 53 48 00 00 13 A2 00
<- 13:40:51.0 7E 00 11 10 04 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 59
<- 13:40:51.0 7E 00 04 08 05 44 31 7D
-> 13:40:51.0 7E 00 07 8B 01 CA 37 00 00 00
<- 13:40:52.0 7E 00 0F 17 05 00 13 A2 00 40 52 A0 D8 87 BD 02 44 31 69
-> 13:40:52.0 7E 00 07 88 03 4D 59 00 00 00
-> 13:40:52.0 7E 00 13 97 02 00 13 A2 00 40 52 A0 D8 87 BD 53 48 00 00 13 A2 00
<- 13:40:52.0 7E 00 11 10 05 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 58
-> 13:40:52.0 7E 00 07 8B 02 CA 37 00 00 00
<- 13:40:52.0 7E 00 04 08 06 44 33 7A
<- 13:40:52.0 7E 00 0F 17 06 00 13 A2 00 40 52 A0 D8 87 BD 02 44 33 66
-> 13:40:52.0 7E 00 06 88 04 44 30 00 01
<- 13:40:52.1 7E 00 11 10 06 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 57
-> 13:40:52.1 7E 00 11 97 03 00 13 A2 00 40 52 A0 D8 87 BD 4D 59 00 87 BD
-> 13:40:52.1 7E 00 07 8B 03 CA 37 00 00 00
<- 13:40:52.1 7E 00 04 08 07 44 34 78
-> 13:40:52.1 7E 00 10 97 04 00 13 A2 00 40 52 A0 D8 87 BD 44 30 00 01
<- 13:40:52.1 7E 00 0F 17 07 00 13 A2 00 40 52 A0 D8 87 BD 02 44 34 64
<- 13:40:52.1 7E 00 11 10 07 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 56
-> 13:40:52.1 7E 00 06 88 05 44 31 00 00
-> 13:40:52.1 7E 00 07 8B 04 CA 37 00 00 00
<- 13:40:52.1 7E 00 04 08 08 44 35 76
-> 13:40:52.1 7E 00 06 88 06 44 33 00 00
<- 13:40:52.2 7E 00 0F 17 08 00 13 A2 00 40 52 A0 D8 87 BD 02 44 35 62
<- 13:40:52.2 7E 00 11 10 08 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 55
-> 13:40:52.1 7E 00 07 8B 05 CA 37 00 00 00
-> 13:40:52.2 7E 00 10 97 05 00 13 A2 00 40 52 A0 D8 87 BD 44 31 00 00
<- 13:40:52.2 7E 00 04 08 09 44 36 74
-> 13:40:52.2 7E 00 10 97 06 00 13 A2 00 40 52 A0 D8 87 BD 44 33 00 00
<- 13:40:52.2 7E 00 0F 17 09 00 13 A2 00 40 52 A0 D8 87 BD 02 44 36 60
-> 13:40:52.2 7E 00 06 88 07 44 34 00 00
<- 13:40:52.2 7E 00 11 10 09 00 13 A2 00 40 52 A0 A8 FF FE 00 00 01 02 03 54
-> 13:40:52.2 7E 00 07 8B 06 CA 37 00 00 00
-> 13:40:52.2 7E 00 06 88 08 44 35 00 01
-> 13:40:52.2 7E 00 10 97 07 00 13 A2 00 40 52 A0 D8 87 BD 44 34 00 00
-> 13:40:52.3 7E 00 07 8B 07 CA 37 00 00 00
-> 13:40:52.3 7E 00 10 97 08 00 13 A2 00 40 52 A0 D8 87 BD 44 35 00 01
-> 13:40:52.3 7E 00 06 88 09 44 36 00 00
-> 13:40:52.3 7E 00 07 8B 08 CA 37 00 00 00
-> 13:40:52.3 7E 00 10 97 09 00 13 A2 00 40 52 A0 D8 87 BD 44 36 00 00
-> 13:40:52.4 7E 00 07 8B 09 CA 37 00 00 00
```

Nach ca. 0.5 Sekunden sind alle Frames versandt und die Antworten vollständig eingegangen.

Wie man oben sieht:
Die Reihenfolge der eingehenden Daten ist nicht identisch mit der Reihenfolge der versendeten Frames.
Es folgt die Ausgabe der jeweiligen rxHandler (Func88(), Func8B(), Func97()).
Wie man unten sieht:
Die Frametypen werden jeweils von der zugeordneten Funktion mit den jeweiligen Daten des Frames bearbeitet.

```
Func88(): empfange FrameId   1, Frametyp: 88, AtCmd: SL, Status: 0, Data: b'@3{\xb7'
Func97(): empfange FrameId   1, Frametyp: 97, AtCmd: SL, Status: 0, Data: b'@R\xa0\xd8'
Func88(): empfange FrameId   2, Frametyp: 88, AtCmd: SH, Status: 0, Data: b'\x00\x13\xa2\x00'
Func8B(): empfange FrameId   1, Frametyp: 8B, Status: 00 00
Func88(): empfange FrameId   3, Frametyp: 88, AtCmd: MY, Status: 0, Data: b'\x00\x00'
Func97(): empfange FrameId   2, Frametyp: 97, AtCmd: SH, Status: 0, Data: b'\x00\x13\xa2\x00'
Func8B(): empfange FrameId   2, Frametyp: 8B, Status 00 00
Func88(): empfange FrameId   4, Frametyp: 88, AtCmd: D0, Status: 0, Data: b'\x01'
Func97(): empfange FrameId   3, Frametyp: 97, AtCmd: MY, Status: 0, Data: b'\x87\xbd'
Func8B(): empfange FrameId   3, Frametyp: 8B, Status: 00 00
Func97(): empfange FrameId   4, Frametyp: 97, AtCmd: D0, Status: 0, Data: b'\x01'
Func88(): empfange FrameId   5, Frametyp: 88, AtCmd: D1, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   4, Frametyp: 8B, Status: 00 00
Func88(): empfange FrameId   6, Frametyp: 88, AtCmd: D3, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   5, Frametyp: 8B, Status: 00 00
Func97(): empfange FrameId   5, Frametyp: 97, AtCmd: D1, Status: 0, Data: b'\x00'
Func97(): empfange FrameId   6, Frametyp: 97, AtCmd: D3, Status: 0, Data: b'\x00'
Func88(): empfange FrameId   7, Frametyp: 88, AtCmd: D4, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   6, Frametyp: 8B, Status: 00 00
Func88(): empfange FrameId   8, Frametyp: 88, AtCmd: D5, Status: 0, Data: b'\x01'
Func97(): empfange FrameId   7, Frametyp: 97, AtCmd: D4, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   7, Frametyp: 8B, Status: 00 00
Func97(): empfange FrameId   8, Frametyp: 97, AtCmd: D5, Status: 0, Data: b'\x01'
Func88(): empfange FrameId   9, Frametyp: 88, AtCmd: D6, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   8, Frametyp: 8B, Status 00 00
Func97(): empfange FrameId   9, Frametyp: 97, AtCmd: D6, Status: 0, Data: b'\x00'
Func8B(): empfange FrameId   9, Frametyp: 8B, Status: 00 00

Gesendet:  18, quittiert: 18, empfangene Datenframes: 0
```

Xbee_frame()

Diesem Objekt vertraue ich in main() empfangenen (Daten-)Frames zur Auswertung an.
Hier wird der Frametyp bestimmt und in das für seine Behandlung zuständige Unterprogramm verzweigt.

Wenn bei der Analyse der eingegangen Daten entschieden wird, dass Aktionen auszuführen sind, dann können die an dieser Stelle veranlasst werden.

Terminarbeit

Manche Jobs wollen in regelmäßigen zeitlichen Abständen oder zur einem bestimmten Zeitpunkt ausgeführt werden:
So könnte es Sinn machen, die Betriebsspannung der Router jede Stunde zu überprüfen.
Oder die Treppenhausbeleuchtung - nachdem sie eingeschaltet wurde - 120 Sekunden später wieder auszuschalten.

Die Umsetzung sieht so aus:
main() läuft in einer Endlosschleife und prüft im Sekundenabstand, ob Aktionen zur Ausführung anstehen.
Anstehende Aktionen sind in Listen eingetragen.

Zwei Listen sind vorbereitet:

timelist[]: hier sind Aktionen eingetragen, die einmalig zu einem bestimmten Zeitpunkt ausgeführt werden.
cronlist[]: hier sind Aktionen eingetragen, die in regelmäßigen Zeitabständen wiederholt ausgeführt werden.

Die Einträge bestehen aus einer Referenz auf eine auszuführende Funktion/Methode sowie ggf. weiteren Parametern, die in einem Tuple zusammengefasst sind.
Um die Standardframes zu versenden, wird folgendes Format verwendet:

```
für Frame 0x08: sendLocalAtCmd, ( atcmd[, datenliste[, rxHandler]] )
für Frame 0x17: sendRemoteAtCmd, ( device, atcmd[, datenliste[, rxHandler]] )
für Frame 0x10: sendDataFrame, ( device, datenliste[, rxHandler] )
```

Die Einträge müssen mindestens die Referenz auf eine ausführbare Funktion enthalten - sonst passiert nichts.
Da die Parameter als "positional parameter" übergeben werden, ist ihre Reihenfolge innerhalb des Tuples von absoluter Bedeutung.
Die Parameterliste darf entfallen, wenn keine Argumente zu übergeben sind.
Bei einem einzelnen Parameter ist Vorsicht geboten: ("MY") wird zu zwei Parametern ("M", "Y") aufgelöst.
Abhilfe schafft ein Komma ("MY",), dann wird anstelle eines Strings ein Tuple mit einem einzigen Element übergeben.

In den Listen können nicht nur die Methoden für das Versenden von Frames verwendet werden.
Jede beliebige Funktion mit beliebigen Parametern kann eingefügt werden.

Einträgen für die timelist[] muss die Angabe vorangestellt werden, nach wievielen Sekunden die Funktion ausgeführt werden soll.

```
(60, sendLocalAtCmd, ("%V, [], readVolt") )
wird in 60 Sekunden das lokale AT-Cmd "%V" ohne Parameter senden und die Antwort an die
Funktion readVolt(station, frame) übergeben.
```

Einträgen in der cronlist[] müssen zwei Werte vorangestellt werden:
Um den Zeitpunkt der Ausführung zu bestimmen, wird die interne Systemzeit (int(time.time()) in Sekunden) mit dem ersten Wert "moduliert" und der verbleibende Rest mit dem zweiten Wert verglichen.

```
int(time.time()) % 60  == 0 trifft immer bei einer vollen Minute zu
int(time.time()) % 60  == 5 trifft immer 5 Sekunden nach einer vollen Minute zu
int(time.time()) % 300 == 1 trifft immer 1 Sekunde nach vollen 5 Minuten zu

( 60, 1, sendRemoteAtCmd, (6, "P0", [], testFunc) ) sendet 1 Sekunde nach einer vollen Minute
an die Station 6 das RemoteAtCmd "P0" ohne weitere Daten, die Rückmeldung mit dem Status von
P0 wird von der Funktion testFunc bearbeitet.
( 60, 10, sendDataFrame, (6, [1,2,3,4] ) ) sendet 10 Sekunden nach einer vollen Minute an die
Station 6 die 4 Bytes 1,2,3,4 via DataFrame0x10.
```

Die Methoden addTimeList( ) bzw. addCronList() fügen die Aktionen in die Wartelisten ein.
Aktionen in der timelist[] werden sofort nach ihrer Ausführung aus der Liste gelöscht.
Es sollte bei Bedarf kein Problem darstellen, eine Funktion zu entwickeln, die während ihrer Ausführung einen neuen Eintrag anlegt.

GUI

Wenn eine graphische Oberfläche realisiert werden soll, dann muss das Hauptprogramm mit seiner Endlosschleife in einen Thread verbandt werden (siehe main_gui.py).
Denn sonst würde top.mainloop() nicht zur Ausführung kommen.
Zuerst wird der Thread App() initialisiert, der wiederum initialisiert einen weiteren Thread zum Empfangen von Daten sowie die erforderlichen sonstigen Klassen.
Danach wird die Graphische Oberfläche als separate Klasse Gui() gestartet..
Um zwischen beiden Bereichen Nachrichten übertragen zu können, werden Callbacks ausgetauscht:

die Klasse App() erhält eine Referenz auf die Callback-Methode von Gui() und umgekehrt.
Auf diesem Wege können empfangene Daten an die graphische Oberfläche durchgereicht und im Gegenzug Benutzeraktionen in der Klasse App() in Aktionen umgesetzt werden.
Ganz zum Schluss wird top.mainloop() gestartet.

In main_gui() in Verbindung mit Xbee_gui() ist das Prozedere im Prinzip ausgetestet.

Xbee_help()

Es ist noch ein Modul Xbee_help() beigefügt, ein Fragment aus einer älteren Anwendung.
Es verfügt über eine einzige Funktion: x_print(frame).
Wenn dieser Funktion der empfangene Frame übergeben wird, dann wird dieser Datensatz in seine Komponenten zerlegt und (relativ) verständlich auf dem Bildschirm ausgegeben.

Michael S.

06. Dezember 2013