1. Zusammenfassung

Um praktische Erfahrung mit XBee-Netzen zu sammeln, habe ich zwei unterschiedliche Wege der Realisierung getesten.

Beim erste Ansatz wird versucht, die Hardware der Module möglichst effizient zu nutzen und mit einem Minimum an zusätzlichen, externen Bauteilen / Controllern auszukommen.

Die Module stellen bekanntlich - neben ihrer Fähigkeit zum Aufbau von Netzwerken und zur Abwicklung des Datentransfers - einige zusätzliche Optionen bereit:

Es sind bis zu 12 IO-Pins verfügbar, die der Anwender als binären/analogen Eingang oder als Ausgang in seine Anwendung einbinden kann.

Der Pin-/ADC-Status kann über 'Funk' von aussen her abgefragt bzw. gesetzt werden.

Eingänge, die als "Digitale Eingänge" definiert werden, können darüberhinaus selbstständig den Pin_Status der IO-Pins versenden, sobald der Level eines einzelnen Pins (etwa durch Schließen eines Kontaktes) verändert wird.

Diese Eigenschaft macht es möglich, XBee-Module als Router oder Enddevice ohne zusätzlichen Controller einzusetzen.

Lediglich der Coordinator oder ein Router, an den die Nachrichten mit den Pin_States gesendet werden, muss die Fähigkeit besitzen, die übermittelten Informationen auszuwerten und Aktionen zu veranlassen.

Mit diesem Ansatz kann ein einfaches Netzwerk etwa für eine Hausautomation aufgebaut werden, in dem diverse binäre Sensoren den Input liefern um binäre Aktoren an beliebiger Stelle innerhalb des Netzes zu schalten.

Eine Beschreibung zu Aufbau und Funkionsweise dieses Netzwerkes folgt im Kapitel 3.

Der zweite Ansatz der Realisierung ignoriert die oben beschriebenen Fähigkeiten der XBee-Module und beschränkt deren Aufgabe auf den Aufbau und die Unterhaltung des Funknetzes sowie die Übertragung von Nachrichten.

Nachteilig ist, dass nun jedem XBee-Modul ein Controller beigestellt werden muss.
Entsprechend höher ist der Aufwand zur Entwicklung der notwendigen Software.

Vorteilhaft ist, dass an den Funkmodul die Anzahl und Art der Sensoren und Aktoren unbeschränkt ist.

Umgesetzt wurde zunächst ein Netzwerk mit der Aufgabenstellung, dass diverse Router Daten aquirieren (Temperatur, Helligkeit etc.) und an einen Datenkollektor senden.
Der protokolliert die Daten auf einem Datenträger und kann weitere Aktionen initieren.

Zusätzlich wurden XBee-Module als Enddevice konfiguriert, die im Sleep-Modus energiesparend im Batteriebetrieb arbeiten können.

Um ein Maximum an Flexibilität zu erreichen, verfügt jeder Controller an einem Router oder Coordinator über einen TWI-Bus, als dessen Master er Sensoren ausgelesen, Aktoren setzen, Daten speichern oder zu Debug-Zwecken ausgeben kann.

Freie Pins des verwendeten ATMega88 werden genutzt, um DS18B20 Temperatursensoren via 1-Wire-Bus anzuschließen.
Diese Sensoren werden jeweils einzeln an je einem Portpin angeschlossen und können dadurch nacheinander ausgelesen werden, ohne dass die Kenntnis/Verwaltung der ROM-Codes erforderlich ist.

Ein Light-to-Frequency-Wandler TSL235 ist an den 16-Bit-Counter1 angeschlossen und liefert Informationen zur Beleuchtungsstärke.

Der Datenkollektor erhält die aktuelle Zeitinformation über ein DCF77-Modul und kann dadurch alle Protokolleinträge mit einem Zeitstempel versehen.

Dieser Ansatz ist nicht Inhalt der nachfolgenden Beschreibung.

2.Grundlagen

2.1 API-Modus

Voraussetzung für den Aufbau eines flexiblen und effizienten XBee-Netzes ist die Arbeit im API-Modus.

Im API-Modus findet die Datenübermittlung zwischen Controller und dem angeschlossenen XBee-Modul über sogenannte 'Frames' statt.
Frames kann man sich wie Formulare vorstellen, in denen alle Informationen an genau vorgeschriebener Positionen eingetragen werden müssen.

Allen Frames gemeinsam ist, dass sie

- mit dem Start Delimiter 0x7F beginnen,
- danach folgen 2 Byte, die die Gesamtlänge des Frames beschreiben,
- das 4. Byte gibt den Frametyp an,
- das allerletzte Byte eines Frames stellt eine Prüfsumme dar.

Die Daten zwischen Byte 4 (dem Frametyp) und dem letzten Byte (der Prüfsumme) differieren in Abhängigkeit vom Frametyp.

Innerhalb der Frames werden neben den bis zu 84 Bytes an Nutzdaten, die 64-Bit-Seriennummer und die 16-Bit-Netzwerkadresse des Absenders sowie diverse Statusinformationen mitübertragen.

Das Formulieren von Frames von Hand ist möglich - aber umständlich und fehleranfällig.

Daher wurde zur 'Erforschung' der XBee's zunächst ein Programm auf einem PC entwickelt, das zu sendende Daten in Frames verpackt bzw. die Daten eines empfangenen Frames strukturiert ausgeben kann.
Das Projekt ist unter dem Betreff "XBee for NewBees .." in der Codesammlung veröffentlicht.

Das Tool ist hilfreich, um XBee-Module zu konfigurieren, Daten auszulesen und allgemein für Funktionstests jeder Art.

2.2 Kommunikation über Frames

Um Daten zwischen Controller und XBee zu übertragen, müssen die Daten - wie oben beschrieben - in einem genau definierten Format (=Frame) an das Modul gesendet werden.

Welches Frameformat zu verwenden ist, das hängt davon ab, welchen Zweck die Sendung erfüllen soll.

Die Antworten - seien es lediglich die Bestätigungen erfolgreicher Übertragungen oder seien es Daten, die zurückgeliefert werden - sind ebenfalls in Frames verpackt.

Der zurückgelieferte Frametyp hängt ab vom Typ des gesendeten Frames.

Zur Kommunikation zwischen Controller und XBee-Modul stehen folgende Frametypen zu Verfügung:

Frame0x08	Sendet ein lokales AT-Kommando (also ein AT-Kommando an das direkt an der seriellen Schnittstelle angeschlossene Modul)
Frame0x17	Sendet ein Remote-AT-Kommando (also ein AT-Kommando an ein beliebiges Moduls innerhalb des Netzwerkes)
Frame0x10	Sendet 'formlose' Daten (deren Formatierung der Benutzer festlegt und bei denen lediglich die Anzahl der Bytes begrenzt ist auf 84) an ein beliebiges Modul innerhalb des Netzwerkes.

Nach dem Senden einer Nachricht wird eine Bestätigung und ggf. eine Antwort zurückgeliefert:

Wenn etwa mit einem Frame0x17 das Remote AT-CMD "D1" versandt wurde, dann wird über den Erfolg der Übertragung sowie über den aktuelle Status des IO-Pin DIO1 via Frame0x97 informiert.

Die Framekombinationen sind:

Versendet wird Frame0x08 - Rückmeldung erfolgt über Frame0x88
Versendet wird Frame0x17 - Rückmeldung erfolgt über Frame0x97
Versendet wird Frame0x10 - Rückmeldung erfolgt über Frame0x8B - Auslieferung am Ziel über Frame 0x90

Manche Frames sendet das XBee-Modul unaufgefordert an den Controller:

Frame0x08 - Das lokale XBee-Modem meldet seine Status (connect, disconnect etc.)
Frame0x92 - Ein Remote Modem sendet seinen Pin-Status
Frame0x95 - Node Identification Indicator, wenn irgendwo der commissioning push button gedrückt wurde

Weitere Frames werden im Zusammenhang mit dem Routing verwendet - das soll aber nicht Thema dieser Beschreibung sein.

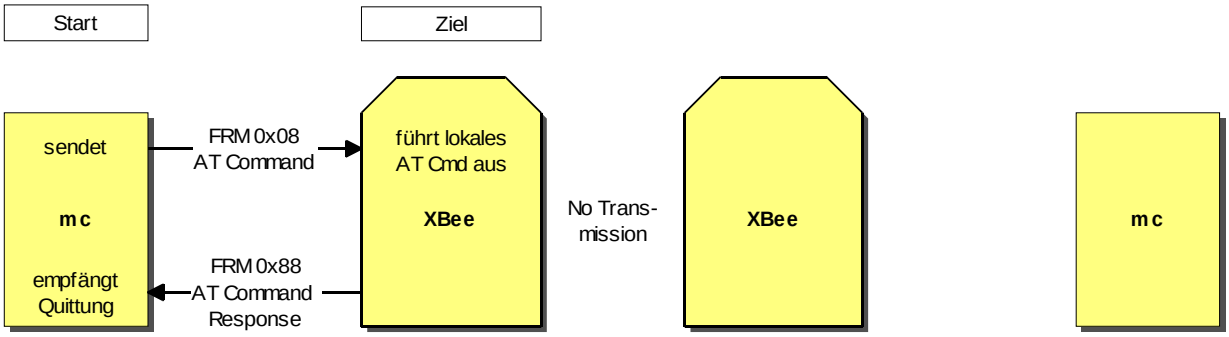
Der exakte Aufbau der einzelnen Frametypen ist in der Dokumentation (90000976_F.pdf) ausführlich, anschaulich und erschöpfend beschrieben und kann dort nachgeschlagen werden (Kapitel 9 - API Frames).

In der Datei "xbee_api.c" sind alle Funktionen bereitgestellt, um Frames aufzubauen bzw. die enthaltenen Informationen gezielt abzuholen.

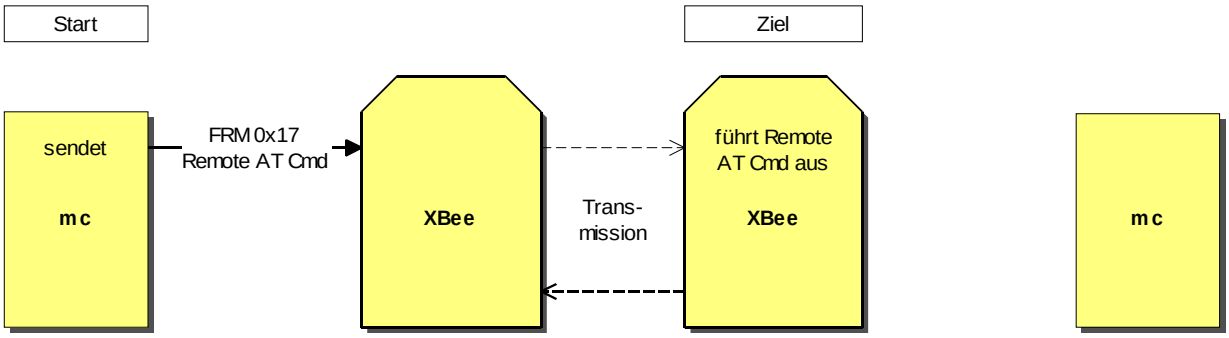
In den nachfolgenden Skizzen sind die Kommunikation zwischen Controller und XBee sowie die dabei zum Einsatz kommenden Frametypen grafisch dargestellt:

1.) KONFIGURATION ETC.

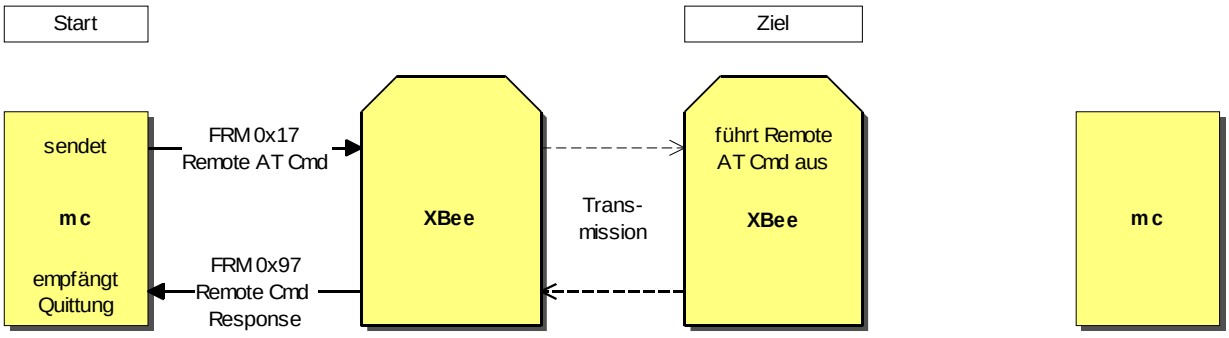
1.1
Ein AT-Command wird an ein lokales Modem versandt.



1.2
Ein Remote-AT-Command wird an ein entferntes Modem versandt
(Frame_Id = 0).

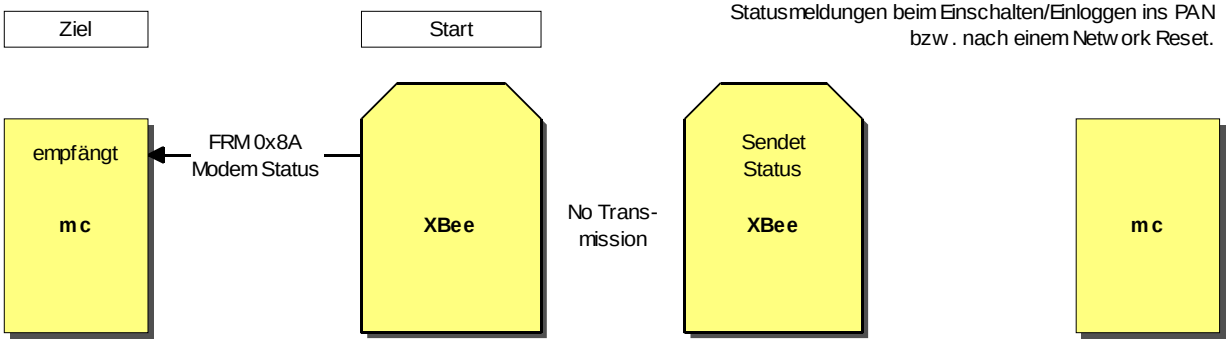


1.3
Ein Remote-AT-Command wird an ein entferntes Modem versandt
(Frame_Id <> 0).



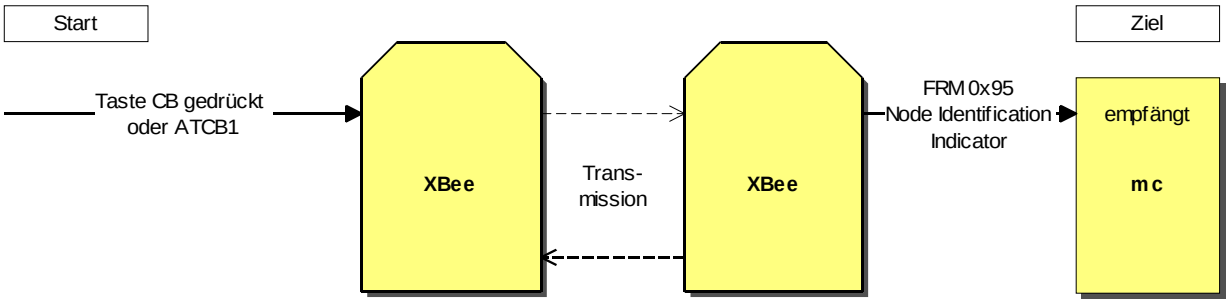
Eine Quittung via FRM 0x97 erfolgt nur dann,
wenn die Frame_Id <> 0 ist !

1.4
Das Modem sendet eine
Statusmeldungen beim Einschalten/Einloggen ins PAN
bzw. nach einem Network Reset.



1.5

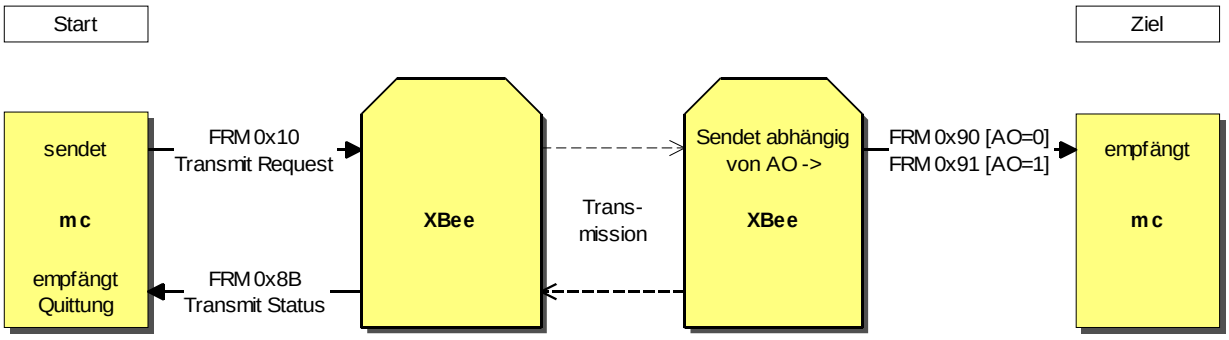
Der Commissioning Button wird gedrückt.
Alle anderen Modems im PAN geben einen FRM 0x95 aus.



2.) DATENÜBERTRAGUNG

2.1

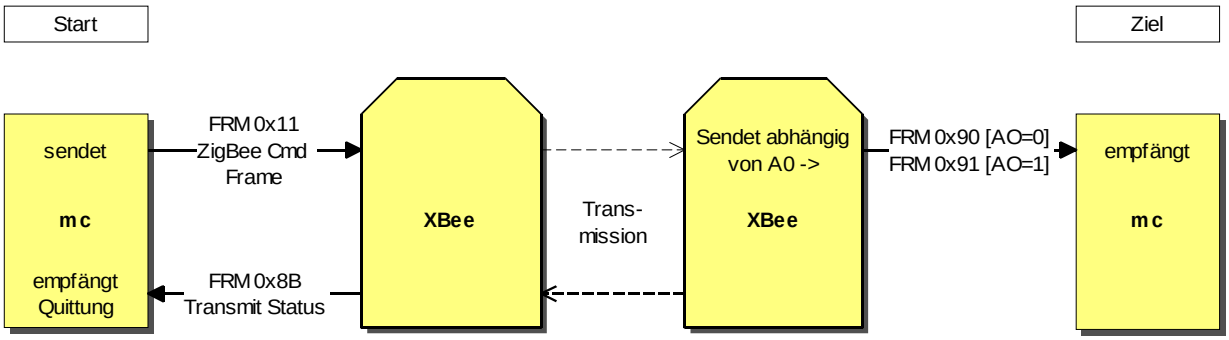
Ein ZigBee Transmit Request wird an ein entferntes Modem versandt.
Die Payload kann bis zu 84 Byte betragen (abhängig von Routing und Security)



Ein FRM 0x8B wird nur zurückgegeben
wenn die Frame_ID <> 0 ist.

2.2

Ein Explicit Adressing ZigBee Command Frame wird an ein entferntes Modem versandt.
Alternativ zu 2.1, wenn ZIgBee Device Objects (ZDO) Commands versandt werden.

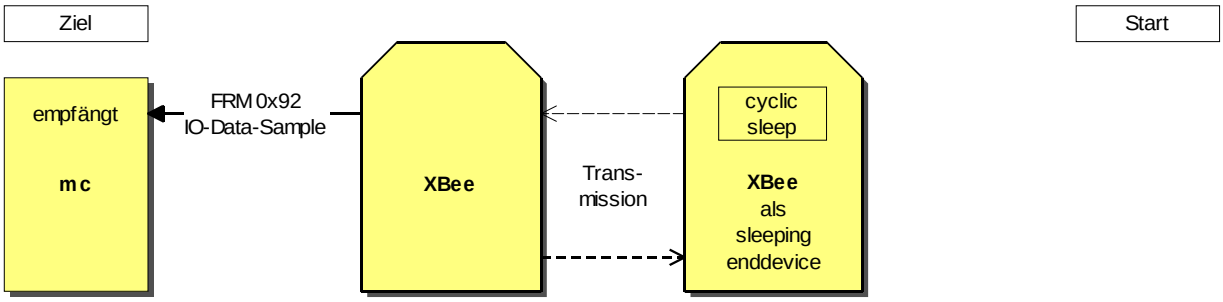


Ein FRM 0x8B wird nur zurückgegeben
wenn die Frame_ID <> 0 ist.

3.1) DATENAQUISE DURCH XBEE OHNE MC

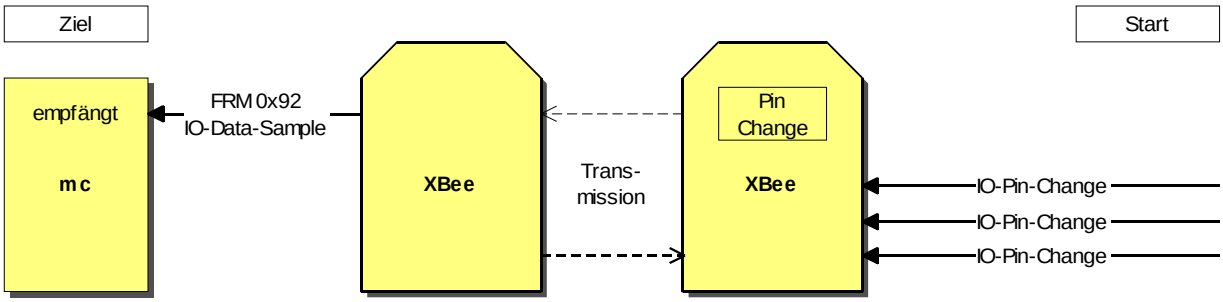
3.1.1

Ein Sleeping Enddevice ohne Microcontroller.
Ein cyclic sleep sendet in regelmäßigen Abständen IO-Samples an den Coordinator / Router.



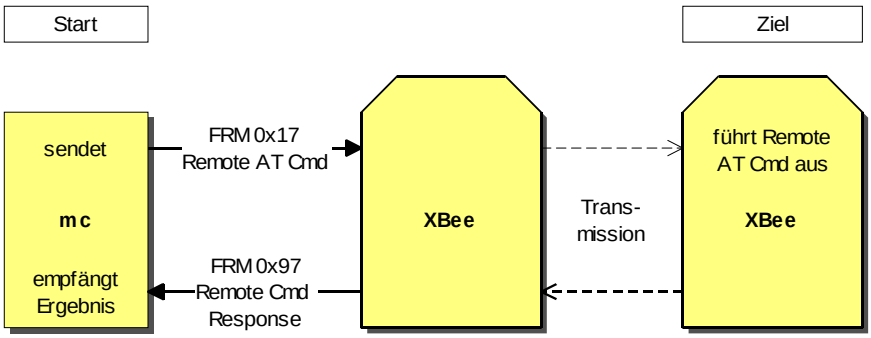
3.1.2

Ein Device ohne Microcontroller.
Ein Pin-Change sendet ein IO-Sample an den Coordinator / Router.



3.1.3

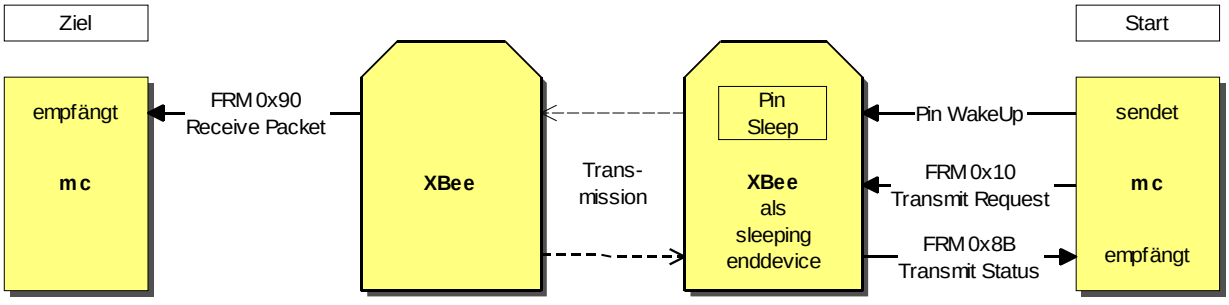
Ein Remote-AT-Command liest z.B. den Portstatus, ADC-Werte am entfernten Modem aus
und liefert das Ergebnis zurück (Frame_Id <> 0 !).



3.2) DATENAQUISE DURCH SLEEPING XBEE MIT MC

3.2.1

Ein Sleeping Enddevice wird vom Microcontroller via Pin-Sleep geweckt.
Der Controller versendet Daten via FRM 0x10.



Wenn AO = 1, dann wird anstelle des FRM0x90 ein FRM0x91 gesendet

Ein FRM 0x8B wird nur zurückgegeben wenn die Frame_ID <> 0 ist.

2.3 Beschreibung des grundsätzlichen Programmablaufs

Die gesamte Koordination und Konfiguration des Netzwerkes machen die XBee-Module unter sich aus.

Ein- und Ausgaben zwischen Controller und XBee-Modul werden über die Serielle Schnittstelle abgewickelt.

Der Controller an einem XBee-Modul empfängt die seriellen Daten interruptgesteuert im Hintergrund.

Der Beginn einer Übertragung ist mit dem Start Delimiter 0x7E markiert. Es folgt die Länge der Nachricht.

Sobald das dritte Byte empfangen worden ist, kann die Gesamtlänge der Nachricht berechnet werden.

Sind alle erwarteten Bytes eingegangen, dann setzt die Interrupt-Routine ein Flag.

Aus main() heraus prüft die Funktion FRM_is_reiceved(), ob dieses Flag gesetzt ist.

Ist es gesetzt, dann wird ermittelt, welcher Frametyp empfangen wurde und anschließend wird in ein Unterprogramm verzweigt, das für die Behandlung des spezifischen Frametypes zuständig ist.

```
main()
{
while(1)
    {
    if FRM_is_received()
        {
        frame = FRM_get_frame_typ();
        if (frame == 0x90) FRM0x90();
        else if (frame == 0x92) FRM0x92();
        .......
        }
    tu_noch_dies();
    tu_noch_das();
    }
}
```

Da die Frames eine klar definierte Struktur habe, können relativ leicht Funktionen entwickelt werden, die gezielt die Nutzdaten der Frames auslesen und exportieren (siehe "xbee_api.c")

Soll ein Frame versendet werden, dann werden die zu übermittelnden Daten in einem Array aufbereitet und anschließend an eine der drei Sendefunktionen übergeben.

Die Adressierung einzelner XBee-Module erfolgt über ihre Position/Index innerhalb des Arrays xbee_target_adr[]:

Die Seriennummern aller Module des Netzwerkes sind in diesem Array aufgelistet.

Die beiden ersten Einträge der Liste haben eine Sonderfunktion:
Der erste Eintrag beschreibt den Coordinator.
Seine 64-Bit [00 00 00 00 - 00 00 00 00] und 16Bit Adresse [00 00] sind bekannt und unveränderlich.

Das gilt auch für den zweiten Eintrag, die Adresse für ein Broadcast [00 00 00 00 00 00 FF FF] [FF FE].

Ab der dritten Zeile folgen nun die Adressen der eingesetzten Module.
In der dritten Zeile trage ich die 64-Bit Seriennummer des Coordinators ein, er hat die konstante 16 Bit Adresse [00 00].

Zur Adressierung werden die Module über ihren Index im Array xbee_target_adr[] angesprochen:
Device #0 ist der Coordinator, Device #1 ein Broadcast, Device #2 ist noch einmal der Coordinator.
Ab Device #3 folgen dann die Router und Enddevices.

Die 16-Bit Netzwerkadressen der Module (default FF FE) werden im laufenden Betrieb vom Programm automatisch durch die vom Coordinator vergebenen 16-Bit Adressen ersetzt.

3. XBee-Netzwerk mit standalone Routern

3.1 Einschränkung zur Vereinfachung des Programms

Nicht durch die XBee-Module sondern zur Vereinfachung der Programmierung und zur Reduzierung der Datenmengen und der SRAM-Anforderung habe ich (fast willkürlich) folgende Einschränkungen definiert:

- Es können maximal 32 XBee-Module verwaltet werden
  (unter Berücksichtigung von Coordinator und Broadcast bleiben 30 physikalische Geräte),
- An jedem XBee-Modul können maximal 8 Digitale Eingänge (DIO0 bis DIO7) benutzt werden.
- Es können insgesamt maximal 256 binäre Eingänge definiert werden (an 32 Geräte je 8 = 256).
- Es können insgesamt maximal 256 Aktoren definiert werden.

Die maximale Anzahl der binären Ausgänge eines Moduls beträgt 12 (Ursache hier: die Hardware).

Diese Einschränkungen sollten für ein "einfaches XBee-Netzwerk" keine echte Beschränkung darstellen.

Um es noch einmal ausdücklich zu betonen:
die Einschränkungen sind nicht durch die XBee-Module bedingt sondern dienen dazu, den Programmcode und den Speicherbedarf zu reduzieren.

Die Zuordnung von auslösenden Sensoren zu schaltenden Aktoren könnte durch ellenlange und qualvolle if ... then ... else Abfragen realisiert werden.

Aber jede Änderung der Logik würde jeweils das Umprogrammieren des Controllers erfordern - und der Programmcode würde vermutlich lang und unübersichtlich werden.

Daher habe ich mich entschieden, die Zuordnung über zwei Tabellen innerhalb eines Serielle Eeprom zu realisieren.
Die Tabellen haben einen Umfang von 3KB und können über die TWI-Schnittstelle und sogar über Funk aktualisiert werden.

Zusätzlich werden noch einige Arrays mit 256 Bit im SRAM benötigt, in denen dynamische Inhalte abgelegt werden.

Die Beschreibung der internen Zusammenhänge zwischen den Tabellen wird im Kapitel 3.3 versucht.

3.2 Konfiguration der Module

Um die Erklärungen nicht unnötig unübersichtlich zu machen, wird als Beispiel von einem kleinen Netz aus 4 Modulen ausgegangen

Als Firmware wird Modem Typ XB24-ZB geladen, für den Coordinator Function Set ZIGBEE COORDINATOR API V21xx und für die Router Function Set ZIGBEE ROUTER API V23xx.

Alle Modulen erhalten eine gemeinsame Extended PAN ID zugewiesen.

Bei den Routern werden 3 Pins als Digitaler Eingang und 3 Pins als Ausgang-Low konfiguriert.
Es werden diejenigen Pins gewählt, die keine wichtige Erstbelegung überschreiben:

D1 = 3, D2 = 3, D3 = 3	(DIO1, DIO2 und DIO3 als Digitaler Eingang)
IC = (2 + 4 + 8) = 14	(DIO1, DIO2 und DIO3 für PinChange freigeben)
D4 = 4, P1 = 4, P2 = 4	(DIO4, DIO11 und DIO12 als Digitaler Ausgang, low-level)

In der Software, die auf dem Controller am Coordinator läuft, müssen die 64-Bit Adressen der beteiligten Module eingetragen werden:

Zeile 0 und Zeile 1 bleiben immer gleich und sind dem Coordinator bzw. dem Broadcast vorbehalten.

In Zeile 2 sollte die Adresse des Coordinators, in den folgenden Zeilen die Adressen der Router eingetragen werden. Die Module werden später jjeweils über den Index (Zeilennummer) innerhalb der Adresstabelle angesprochen. Der Index beginnt mit 0 = Coordinator, 1 = Broadcast etc.

Tabelle 1 32 * 16 Byte im Eeprom / 32 * 10 Byte im SRAM
Byte0 .. 7 = 64-Bit Netzwerkadresse - Byte 8/9 = 16 Bit Netzwerkadresse

Idx	Adr	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0	0x0000	00	00	00	00	00	00	00	00	00	00				Coordinator		
1	0x0010	00	00	00	00	00	00	FF	FF	FF	FE				Broadcast		
2	0x0020	00	13	A2	00	A1	12	0B	0C	00	00				Coordinator		
3	0x0030	00	13	A2	00	A1	13	1C	CC	FF	FE				Router_1		
4	0x0040	00	13	A2	00	99	A0	55	88	FF	FE				Router_2		
5	0x0050	00	13	A2	00	C4	06	3A	AA	FF	FE				Router_3		
6	.....														.....		

Tabelle 1 - Adressen der XBee's im SRAM / Eeprom des ATMega88

3.3 Zuordung von Sensoren und Aktoren über Tabellen

Führen wir uns vor Augen, wie die Kommunikation im Netzwerk abläuft.

Ein Schalter, ein Reed-Kontakt oder sonstiges wird betätigt und verändert an einem als Digitalen Eingang konfigurierten und für den PinChange freigegebenen Pin den Pin_Status.

Daraufhin sendet das XBee-Modul ohne externes Zutun seinen Port_Status an dasjenige Device, dessen Adresse in den Registern Destination_High und Destination_Low eingetragen ist.

Per default stehen hier 8 Nullen, das Zieldevice ist daher der Coordinator.

Der Coordinator empfängt die Nachricht als Frame 0x92.

Aus diesem Frame kann er folgende Informationen herausdestillieren:

- wer ist der Absender der Nachricht ?
- welchen Status haben die einzelnen Pins ?

Der Port_Status hat genaugenommen eine Länge von 12 Bit.
Bedingt durch meine Definition, dass nur die Pins DIO0 bis DIO7 als Eingang genutzt werden dürfen, ist es ausreichend, nur das Low-Byte mit den unteren 8 Bit zu prüfen !

Die Funktion FRM0x92_get_sample_lowbyte() liefert den 8-Bit Port_Status zurück.

Der Absender der Nachricht wird mit der Funktion FRM_get_sender_idx() ermittelt und liefert den Index des Modules in der Tabelle 1 (siehe oben).

Nehmen wir an, dass der Coordinator folgendes registriert:

```
der Absender ist der Router_1 (Index 3),        -> sender_idx = 3
der Portstatus 0x06 (Pin.1 und Pin.2 gesetzt)   -> port_state = 0x06
```

Für jedes der 32 zulässigen logischen XBee-Module ist in der Tabelle 2 im Seriellen Eeprom ein Datenbereich von 32 Byte (für 8 Datensätze mit je 4 Byte) reserviert.

Das Startbyte des Datenbereiches berechnet sich aus (Absender_Idx * 32), also (3 * 32) = 96 = 0x0060.

Das Programm liest ab dieser Position 32 Byte in das Array sensor_data[] ein.

Tabelle 2 32 * (8 * 4) Byte in Seriellem Eeprom - Start ab Adresse 0x0000

	Adr	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
	.....																
3	0x0060	01	03	00	00	02	04	00	00	04	05	00	00	FF	FF	FF	FF
	0x0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
4	0x0080	02	03	00	00	02	01	00	00	FF	FF	FF	FF	FF	FF	FF	FF
	0x0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
5	0x00A0	04	05	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
	0x00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
	.....																

Tabelle 2 - Sensorbeschreibung

Jeweils 4 Byte beschreiben einen der 8 definierbaren Eingangspins eines XBee-Moduls:

Byte.0 = Pinmaske für den zugeordneten Pin, also 1 für DIO0, 2 für DIO1, 4 für DIO2, 8 für DIO3 etc.
Byte.1 = Referenz auf einen Eintrag in Tabelle 3 (Idx der Aktorbeschreibung)
Byte.2 = Spezialbyte für Sonderfälle (Invertieren des Pinstate etc.) (0 = inaktiv)
Byte.3 = Referenz auf eine Bedingung zur Ausführung (0 = inaktiv)

Der Bereich 0x0060 bis 0x07F in der Tabelle 2 sagen aus:

Für XBee #3 gibt es drei Sensordefinitionen, die übrigen 5 sind unbelegt:
1.) Pin_Mask 0x01 (Pin.0) ruft die Aktordefinition an Pos. 3
2.) Pin_Mask 0x02 (Pin.1) ruft die Aktordefinition an Pos. 4
3.) Pin_Mask 0x04 (Pin.2) ruft die Aktordefinition an Pos. 5

Der Bereich 0x0080 bis 0x009F in der Tabelle sagen aus:
Für XBee #4 gibt es zwei Sensordefinitionen, die übrigen 6 sind unbelegt:

1.) Pin_Mask 0x02 (Pin.1) ruft die Aktordefinition an Pos. 3
2.) Pin_Mask 0x02 (Pin.1) ruft die Aktordefinition an Pos. 1 (der Pin 1 führt damit 2 Aktoraktionen aus).

für XBee #5 gibt es nur 1 Aktordefinition, die übrigen 7 sind unbelegt:

1). Pinmask 0x04 (Pin.2) ruft die Aktordefiniton an Pos. 5

Der nächste Datensatz beginnt in Byte.0 mit 0xFF, es werden keine weiteren Definitionen mehr untersucht.

Die Arbeit beginnt mit der ersten Sensordefinition (0x01, 0x03, 0x00, 0x00):
Zuerst wird geprüft, ob das Byte.0 = 255 (das würde die Prüfung beenden).
Anschließend wird untersucht, ob tatsächlich nur genau ein einzelnes Bit in der Pin_Maske gesetzt ist.

Danach kann das Programm den Port_Status mit der Pin_Maske des 1. Datensatzes maskieren:

```
Port_Status   = 0x06
Pin_Mask      = 0x01        -> Pin_State     = FALSE
```

Das Problem ist nun - mit dieser Information alleine ist uns nicht geholfen.
Denn wir wissen nicht, ob der Pin_Status sich geändert hat - oder er vorher auch schon auf FALSE stand.

Da bis zu 8 Pins eines XBee-Moduls Auslöser für einen Frame0x92 gewesen sein können, kann am Pin_Status / Port_Status alleine nicht erkannt werden, welcher der Pins der Auslöser war.

Demzufolge muss ein Mechanismus her, der den letzten aktuellen Pin_Status liefert:
der jeweils letzte Pin_Status muss gespeichert werden.

Im SRAM wird ein Array aus 256 Bits (=32 Byte) namens "last_pin_state[]" eingeführt, das den letzten aktuellen Pin_Status widerspiegelt.

```
Tabelle 2a - Array im SRAM - last_pin_state[32] = 256 Bit
Notiert für jeden Sensorpin den letzten Status
```

Es muss last_pin_state verglichen werden mit dem aktuellen Pin_Status.
Nur wenn sich im Vergleich eine Veränderung ergeben hat, dann war dieser Pin der Auslöser des Frames.

Das Programm vergleicht den Pin_Status desjenigen Pins, der durch die Pinmaske definiert ist, mit dem Status im Array last_pin_state[].
Die Position des Bits errechnet sich aus ((XBee_Idx * 8) + Nr. der Sensordefinition) und steht in der Variablen last_pin_state_idx.

```
port_Status    = 0x06
pin_mask       = 0x01          -> pin_state     = FALSE
                               last_pin_state   = FALSE (Annahme)
                                                -> Pin.0 wurde nicht geändert
```

Sind last_pin_state und pin_state gleich, dann ist der Pin nicht geändert worden, es wird die nächste Definition untersucht (0x02, 0x04, 0x00, 0x00).

```
port_status    = 0x06
pin_Mask       = 0x02          -> pin_state     = TRUE
                               last_pin_state   = TRUE (Annahme)
                                                -> Pin.1 wurde nicht geändert
```

Wieder hat sich der Pin_Status nicht verändert, es geht weiter mit Definition 3 (0x04, 0x05, 0x00, 0x00):

```
port_status    = 0x06
pin_mask       = 0x04          -> pin_state     = TRUE
                               last_pin_state   = FALSE (Annahme)
                                                -> Pin.2 wurde geändert
```

In diesem Falle wird eine Veränderung erkannt.

Nun wird die zugehörige Aktordefinition eingelesen, auf die das Byte.1 verweist (-> 0x05):

Die Aktordefinitionen folgen hinter der Tabelle 2 im Seriellen Eeprom ab Position 1024.

Die referenzierte Definition steht an der Position 1024 + (Aktor_idx * 8) = 1024 + (5 * 8) und ist 8 Byte lang.

Tabelle 3 256 * 8 Byte in Seriellem Eeprom - Start ab Adresse 0x0400

Idx	Adr	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0	0x0400	03	FF	44	34	04	05	FF	00
1	0x0408	03	FF	50	31	04	05	FF	00
2	0x0410	03	FF	50	32	04	05	FF	00
3	0x0418	04	FF	44	34	04	05	FF	00
4	0x0420	04	FF	50	31	04	05	FF	00
5	0x0428	04	FF	50	32	05	04	FF	00
	.....								

Tabelle 3 - Beschreibung der Aktoren

Byte.0 - Device_Idx des XBee-Moduls, auf dem sich der Aktor befindet
Byte.1 - Aktortyp (1 = Taster, 2 = Absolutschalter, 4 = Parallelschalter, alles andere = Wechselschalter)
Byte.2 - erstes Zeichen des AT-Commandes, das gesendet werden soll
Byte.3 - zweites Zeichen des AT-Commandes, das gesendet werden soll
Byte.4 - Parameter des AT-Commandes zum AUSschalten
Byte.5 - Parameter des AT-Commandes zum EINschalten
Byte.6 - Referenz auf einen 8-Bit Zähler (nur für einen Parallelschalter)
Byte.7 - Referenz auf eine Bedingung zur Ausführung (0 = inaktiv)

Das Byte.1 beschreibt das Verhalten des Aktors, wenn der zugeordnete Sensorpin geändert wird:

- Wechselschalter — der Aktor wird grundsätzlich invertiert (default).
- Absolutschalter — der Aktor wird grundsätzlich auf den Level des Sensors (pin_state) geschaltet.
- Taster — der Aktor wird nur beim EINschalten UMgeschaltet, beim Ausschalten wird keine Aktion ausgeführt.
- Parallelschalter — es können mehrere Schalter parallel geschaltet werden. AUSgeschaltet wird dann, wenn alle Schalter ausgeschaltet sind EINgeschaltet wird dann, wenn mindestens 1 Schalter eingeschaltet ist.

Die Aktordefinitionen in der Tabelle 3 sind alle Schalter als Wechselschalter (Byte.1 = 0xFF) definiert, die Definition bedeuten im Einzelnen:

Aktor 0 - Schicke an XBee #3 das AT-CMD "D3" mit Parameter 4 für AUS bzw. Parameter 5 für EIN
Aktor 1 - Schicke an XBee #3 das AT-CMD "P1" mit Parameter 4 für AUS bzw. Parameter 5 für EIN
Aktor 2 - Schicke an XBee #3 das AT-CMD "P2" mit Parameter 4 für AUS bzw. Parameter 5 für EIN
Aktor 3 - Schicke an XBee #4 das AT-CMD "D4" mit Parameter 4 für AUS bzw. Parameter 5 für EIN
Aktor 4 - Schicke an XBee #4 das AT-CMD "P1" mit Parameter 4 für AUS bzw. Parameter 5 für EIN
Aktor 5 - Schicke an XBee #4 das AT-CMD "P2" mit Parameter 5 für AUS bzw. Parameter 4 für EIN

In unserem konkreten Beispiel wird die Aktordefinition 5 aufgerufen, der aktuelle Pin_State ist TRUE.

Mit Hilfe der Daten in der Tabelle 3 kann der erste Teil des AT-Command zusammengestellt werden, das gesendet werden soll: es besteht aus den Bytes 2 und 3:
Zusätzlich wird als Parameter Byte 4 oder Byte 5 angehängt.

Das Kommando lautet also: "P2"
Das Zieldevice in Byte.0 des Datensatzes eingetragen: es ist das Modul mit dem Index 4.

Aber das Problem ist - welcher der beiden Parameter (4 oder 5) soll gewählt werden:
Wenn der Pin_Status gewechselt werden soll, dann muss der letzte Zustand des Aktors invertiert werden.

Nur wie soll geschaltet werden, wenn der letzte Zustand nicht bekannt ist ?

Man könnte natürlich per Remote-AT-CMD über Funk den Pin_Status auslesen ....

Schneller geht es so:

Es muss der aktuellen Zustand eines jeden Aktors mitgeführt werden.
Dazu wird wieder ein Array im SRAM angelegt, in dem für jeden der bis zu 256 Aktoren ein Bit reserviert ist.

```
Tabelle 3a - Array im SRAM -> last_aktor_state[32] = 256 Bit
Notiert für jeden Aktor den letzten Status
```

Das Programm holt sich den letzten Aktor_Status aus diesem Array - und nun kann entschieden werden, ob der Aktor EIN, AUS oder womöglich garnicht geschaltet werden muss (letzteres, wenn er bereits den Status innehat, der vom Anwender per Schalter gefordert wird).

Jetzt sind die Daten für das Remote-AT-CMD vollständig und können an das Modul des Aktor gesendet werden.
Das übernimmt die Funktion FRM0x17_send_remote_at_cmd().

Den Rest macht das adressierte Zieldevice, es schaltet einen digitalen Ausgang ....

Und das wars dann auch.

3.4 Ergänzende Hinweise

Um die Beschreibung auf Wesentliches zu konzentrieren, sind einige Hinweise unterschlagen worden.
Das soll hier nachgeholt werden.

Wenn in der Tabelle 2 (Sensordefinitionen) die Pinmaske (Byte.0) = 255, dann wird die Prüfung weiterer Pins abgebrochen - alle nachfolgenden Sensordefinitionen werden ignoriert.

Sofern in der Pinmaske (Byte.0) mehr als 1 Bit gesetzt ist, wird diese Sensordefinition überschlagen (sie ist nämlich unzulässig !).

In der Tabelle 2 (Sensordefinitonen) gibt es ein Spezialbyte (Byte.2) für Sonderfälle.

Wenn alle Bits gelöscht sind (0x00), dann wird das Byte ignoriert.

Wenn Bit.4 gesetzt ist, dann wird der erkannte Pin_Status invertiert.
Das ist nützlich, wenn ein Schalter/Taster nicht mit einem Arbeitskontakt, sondern mit einem Ruhekontakt arbeitet.

Wenn Bit.0/1 gesetzt ist, dann wird der Schaltertyp in der Tabelle 3 Aktordefinitionen überschrieben.

Wichtig im Zusammenhang mit der Pin_Maske ist:

Der Anwender muss dafür sorgen, dass die Pin_Maske immer genau auf EINEN Pin zeigt.
Dieser Pin muss als "Digitaler Eingang" konfiguriert sein und mit dem AT-CMD 'IC' für einen PinChange freigegeben sein.

Zeigt eine Pin_Maske versehentlich auf einen "Digitalen Ausgang" (=Aktor) - und wird dieser Ausgang verändert, dann wird die hinterlegte Schaltaktion ausgelöst.
Allerdings nicht unmittelbar - denn der digitale Ausgang selbst kann keinen Interrupt auslösen.
Erst wenn durch einen freigegebenen PinChange ein Port_Status an den Coordinator gesendet wird, dann wird die fehlerhalte Sensordefinition einen Aktor verändern können.

Zeigt eine Pin_Maske auf einen inaktiven Pin, dann wird diese Sensordefinition nie ausgeführt werden, denn dieser Pin liefert immer unverändert denselben Status.

3.5 Bedingungen

Die Ausführung von Aktionen kann davon abhängig gemacht werden, dass Bedingungen erfüllt sind.
Bedingungen können an zwei Stellen im Programmablauf eingesetzt werden:
in der Tabelle der Sensordefinitionen und in der Tabelle der Aktordefinitionen.

Dieses Feature ist nur testweise implementiert.
Die Arbeitsweise ist so gedacht:

Das Byte.3 bzw. Byte.7 verweisen auf ein Bit in einem Array "condition_state[32]" mit 256 Bit.

Ist das Bit gesetzt, dann gilt die Bedingung als erfüllt, ist es gelöscht, dann ist die Bedingung nicht erfüllt.

Das Setzen bzw. Löschen der Bits übernehmen Programmteile, die hier nicht beschrieben sind:
Erkennt etwa ein Helligkeitssensor, dass im Aussenbereich ein Grenzwert unterschritten ist, dann könnte er in der Tabelle der Bedingungen ein Bit setzen/löschen.
Die Aussenbeleuchtung wird durch einen Bewegungsmelder nur dann eingeschaltet, wenn das Bit für die Bedingung 'Dunkelheit == TRUE'.

Alternativ könnte das Byte auch auf ein Array von Funktionspointern zeigen.
Zur Laufzeit würde dann die hinterlegt Funktionen ausgeführt und liefern TRUE oder FALSE zurück.

Wenn als Bedingung 0x00 eingetragen ist, dann wird keine Prüfung durchgeführt.

3.6 Debugging

Da die Serielle Schnittstelle für die Kommunikation zwischen Controller und XBee benutzt wird, ist dieser Weg zum Debuggen verschlossen.
OK, man kann an TxD/RxD mitlauschen, aber da sieht man ja nur die Frames - die sind ohne Interpretationshilfe wenig aussagekräftig.

Daher werden Statusinformationen über den TWI-Bus ausgegeben.
Hier kann ein Datenlogger, ein LCD oder ein TWI-RS232-Umsetzer angeschlossen werden.

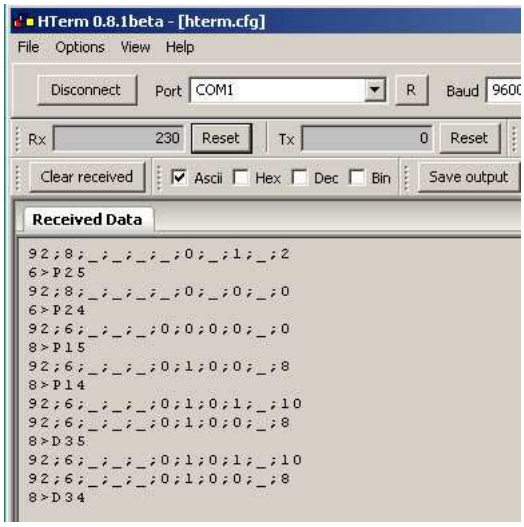
Letzteren habe ich benutzt, um die Daten auf dem Bildschirm eines PCs anzuzeigen
(siehe Codesammlung "Terminal/TWI-Adapter ...", damit lassen sich auch Daten ins TWI-Eeprom schreiben).

Die Protokollierung beginnt mit der Angabe des empfangenen Frametyps (hier '92'), durch Semikolon getrennt folgt der Absender des Frames ('8'), der Pinstatus in binärer Darstellung sowie als Dezimalwert.

In der binären Darstellung bedeutet ein '_', dass dieser Pin deaktiviert ist, eine '0', dass der Pin auf Low und eine '1', dass der Pin auf High gezogen ist.

Aus dem nachfolgenden Listing kann man folgende Erkenntnisse ziehen:

An XBee #8 sind DIO1 und DIO3 als Digitaler Aus- oder Eingang geschaltet.
An XBee #6 sind DIO1, DIO2, DIO3 und DIO4 Digitaler als Aus- oder Eingang geschaltet.



Die Interpretation im Detail:
Frame92 wurde von #8 gesendet, der Portstatus ist 2
es wird an #6 das AT-CMD "P2" mit Parameter 5 gesendet.

Frame92 wurde von #8 gesendet, der Portstatus ist 0
es wird an #6 das AT-CMD "P2" mit Parameter 4 gesendet.

Frame92 wurde von #8 gesendet, der Portstatus ist 0
es wird an #6 das AT-CMD "P1" mit Parameter 5 gesendet.

Frame92 wurde von #8 gesendet, der Portstatus ist 8
es wird an #6 das AT-CMD "P1" mit Parameter 4 gesendet.

Frame92 wurde von #6 gesendet, der Portstatus ist 10
es wird keine Aktion ausgeführt.

Frame92 wurde von #6 gesendet, der Portstatus ist 8
es wird an #8 das AT-CMD "D3" mit Parameter 5 gesendet.

Frame92 wurde von #6 gesendet, der Portstatus ist 10
es wird keine Aktion ausgeführt.

Frame92 wurde von #6 gesendet, der Portstatus ist 8
es wird an #8 das AT-CMD "D4" mit Parameter 5 gesendet.

Das Verhalten des DIO1 an Device #6 (in den letzten 6 Zeilen) rührt daher, dass dieser Sensor als Taster definiert ist.
Er führt nur dann eine Aktion aus, wenn er betätigt wird (von High nach Low wechselt).
Beim Loslassen (Übergang von Low nach High = Portstatus wechselt von 8 auf 10) bleibt er untätig.

4. Schaltpläne

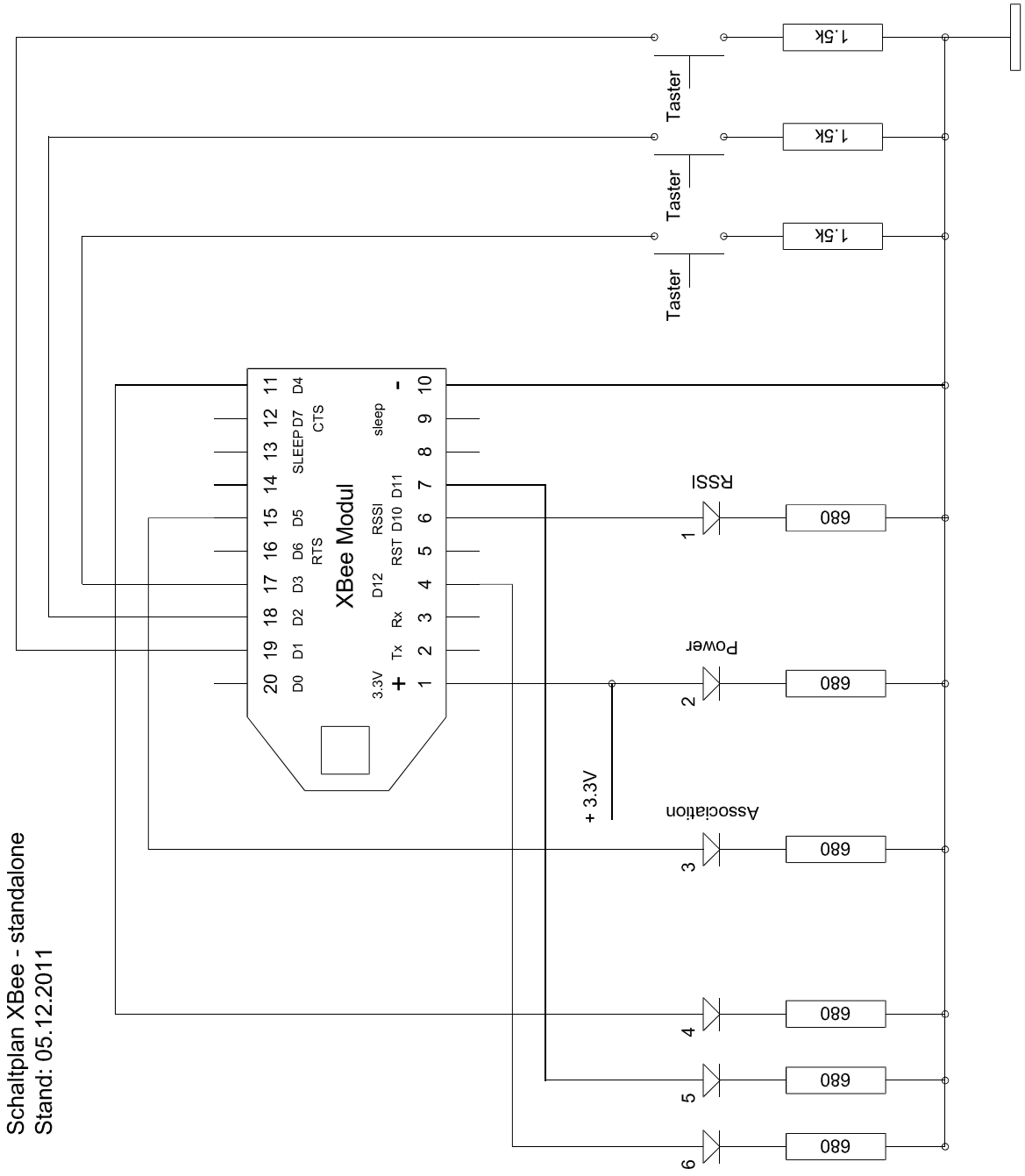
Zum Abschluss folgen die Schaltpläne für den Coordinator und die Router.

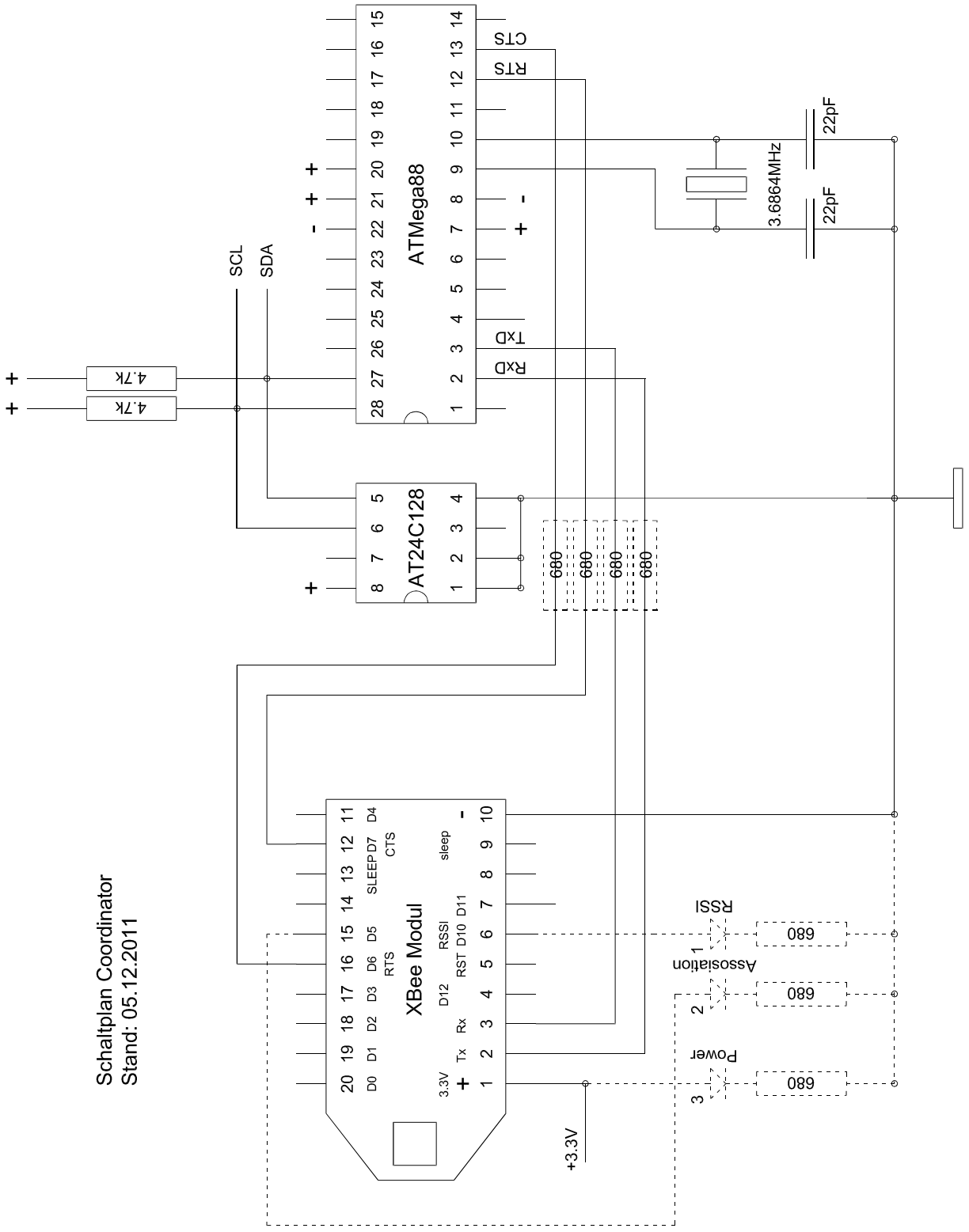
Die Widerstände in den Leitungen RxD, TxD, CTS, RTS sowie zwischen Tastern und gnd dienen dazu, bei versehentlichem Umkonfigurieren der Eingänge/Ausgänge portkillende Kurzschlüsse zu verhindern.

Um größere Lasten (als LowCurrent LED's) zu treiben, kann man FET's wie BTS 141 oder IRLZ 34 N verwenden, die kommen mit dem niedrigen 3.3 V Eingangslevel zurecht.

10.12.2011

Michael S.


Schaltplan XBee - standalone
Stand: 05.12.2011
XBee Modul
1.5k
Taster
680
RSSI
Power
Association
+ 3.3V


Schaltplan Coordinator
Stand: 05.12.2011
ATMega88
AT24C128
XBee Modul
SCL
SDA
RxD
TxD
CTS
RTS
3.6864MHz
22pF
4.7k
680
RSSI
Association
Power
+3.3V