KNX CALIBUR TOCHTER API

API Dokumentation

Florian Guggenberger
0226774 / 533

21. Dezember 2008

Inhaltsverzeichnis

1 Grundkonfiguration

Mit folgenden Pinbelegungen können alle Funktionen der API benutzt werden.

Pinbelegung		
Sensor/Device	McuPin	TochterPin
Leds	P70 : P77 (*	LED1 : LED8
Switches	P70 : P77 (*	SWT1 : SWT8
Buttons	P70 : P77 (*	BUT1 : BUT8
Buzzer	P25 GND	Buzzer(1) Buzzer(2)
Relais	P70 (*	Relais
TempDigital	P80 (*	TEMP_DIG
TempAnalog	P81	TEMP_AN
Helligkeit	P82	LDR
Druck	P83	PRESS
Feuchte	P27 P84 PB5	0V ADC AC
Zeit(DCF77)	PA0	DCF77
RS485	Pxx P87 Pxx BUSA BUSB	DI RE/DE RO BUSA BUSB

(* diese Pinbelegungen können vom Benutzer konfiguriert werden.

1.1 Konfigurationsmöglichkeiten

Die Pinbelegungen für LEDs, Switches, Buttons, Relais und Temp_Digital lassen sich benuzterspezifisch konfigurieren. Alle anderen Belegungen müssen auf Grund von Hardwarelimitationen nach oben genannten Belegungen beibehalten werden, und lassen sich nicht verändern.
Für die veränderbaren Pinbelegungen steht das File *tochter_config.h* zur Verfügung. Dabei muss lediglich die Nummer des Ports(in Hex-format) angegeben werden, an dem

das jeweilige Device angeschlossen ist. Werden beispielsweise alle LEDs an Port 7 (default) angeschlossen, wird im oben genannten Konfigurationsfile bei dem #define LEDS die Nummer 0x7 angegeben.
Das jeweilige Direction-Register wird von den Aufrufenden Funktionen automatisch gesetzt. Dabei ist es durchaus möglich ein Port sowohl als Aus- als auch als Eingang zu benutzen.

2 I/O Schnittstellen

Hier werden die wesentlichen Softwareschnittstellen für die Ansteuerung der I/O Komponenten beschrieben.

2.1 I/O Funktionen

Vorbereitung:
Konfigurieren Sie die entsprechenden Defines für LEDS, BUTTONs und SWICHes im File *tochter_ config.h*. Geben Sie dabei die jeweilige Portnummer in Hexformat an, an der die I/O Komponenten angeschlossen sind.

Funktionen:
Für die Ansteuerung der I/O Komponenten existieren vier wesentliche Funktionen:

```
int setPort(int port, int value);
```

Gibt an Portnummer port den Wert Value aus, und setzt automatisch das Port als Ausgang.

```
int getPort(int port);
```

Setzt das ganze Port als Eingang und retourniert die anliegenden Pinvalues.

```
int setPin(int port, int pin, int onoff);
```

Setzt den korrespondierenden Pin als Ausgang und setzt ihn entsprechend dem Parameter *onoff* auf logisch 1 (onoff $>$ 0) oder auf logisch 0 (onoff =0)

```
int getPin(int port, int pin);
```

Retourniert den Zustand des korrespondierenden Pins und setzt dazugehöriges Direction-Register auf Eingang.

Beim Umgang mit diesen Funktionen ist darauf zu achten, dass die Port-Methoden (setPort() und getPort()) nur richtige Werte liefern, falls das gesammte Port entweder als Ausgang oder als Eingang verwendet wird.
Wird ein Port sowohl als Ein- als auch als Ausgang verwendet, sollten die Pin-Methoden (setPin() und getPin()) zum Einsatz kommen um korrekte Ergebnisse zu erhalten.

2.2 RELAIS

Vorbereitung:
Konfiguration der Defines *#define RELAIS* und *#define RELAIS_ DIR* in *tochter_ config.h*

Funktionen:

```
void relais(int on_off);
```

Schaltet das Relais. Wird eine 1 übergeben wird das Relais angezogen, wird eine 0 als Parameter übergeben, wird das Relais gelöst

Bitte bedenken Sie, dass das Relais über 5V versorgt wird und vergessen Sie bitte nicht den 5V Schalter zu betätigen.
Alternativ kann das Relais natürlich auch über die SetPin()-Methode gesteuert werden. Der Unterschied dabei ist nur, dass bei der relais()-Methode die Defines wie in der Vorbereitung beschrieben gesetzt sein müssen, während bei der SetPin()-Methode der Port und der Pin explizit übergeben wird.

3 Sensoren

In diesem Kapitel werden die wichtigsten Funktionen zur Ansteuerung aller Sensoren vorgestellt, ausserdem wird auf, bei der Entwicklung aufgetretene Probleme, in Bezug auf einige Sensoren näher eingegangen.

3.1 Analoger Temperatursensor

Vorbereitung:
Herstellen einer elektrischen Verbindung zwischen Temp_AN und Pin P81 siehe Grundkonfiguration.
Funktionsweise:
Der Sensor verändert in Abhängigkeit der Umgebungstemperatur seine Ausgangsspannung.
Die Messung der Ausgangsspannung wird über Kanal 9 (Pin P81) des ADCs durchgeführt.

Funktionen:

```
void initTempAnalog(void);
```

Initialisiert ADC für den analogen Temperatursensor

```
float getTempAnalog();
```

Liefert den Temperaturwert des analogen Temperatrusensors als Gleitkommazahl in Grad Celsius

```
int sgetTempAnalog(char * szTempString);
```

Speichert den Temperaturwert als formatierte Zeichenkette in den übergebenen Parameter und retouniert die Anzahl der geschriebenen Zeichen.

Bemerkung:
Der analoge Temperatursensor benötigt ca. 5 min. um auf Betriebstemperatur zu gelangen. Diese Zeit sollte abgewartet werden, um einen gültigen Temperaturwert auszulesen. Auf Grund der relativen Ungenauigkeit des Temperatursensors (Fehler 2-6 Grad Celsius) wird der Temperaturwert über 50 Messungen gemittelt.

3.2 Digitaler Temperatursensor

Vorbereitung: Herstellen einer elektrischen Verbindung zwischen Temp_DIG und Pin P80 siehe Grundkonfiguration, oder Konfiguration der Defines *#define DQ* und *#define DQ_DIR* in *tochter_config.h*
Funktionsweise:
Der digitale Temperatursensor wird über one-wire mit dem Mikrokontroller verbunden (Pin P80). Die Kommunikation mit dem Sensor erfolgt durch einen seriellen Datenstrom über diese Verbindung. Dabei wird das Timing für die Kommunikation über eine delay-Loop realisiert.
Da der Sensor durch den Betrieb einer Eigenerwärmung von ca. 4 Grad unterworfen ist, wird dieser Offset vor Ausgabe des Messwertes vom selbigen abgezogen. Siehe TEMPD_COR_FAC.

Funktionen:

```
double fRead_Temperature (char * TempString);
```

Liefert den digitalen Temperaturwert als Gleitkommazahl.

```
int sRead_Temperature (char * TempString);
```

Liest den Temperaturwert des digitalen Temperatursensors und schreibt diesen als formatierte Zeichenkette in den übergebenen Parameter. Das übergebene Feld muss mindestens die Größe von BUFFER_SIZE (10) haben. Die Funktion retouniert entweder die Anzahl geschriebener Zeichen oder 127 falls kein Sensor gefunden wurde. Es kann auch vorkommen, dass die Erkennung eines nicht angeschlossenen Sensors fehlschlägt(floating pin). In diesem Fall wird die Messung 0 liefern.

```
signed char iRead_Temperature (void);
```

Liest den Temperaturwert des digitalen Temperatursensors und retouniert diesen als Ganzzahl. Retourniert 127 falls kein Sensor angeschlossen ist und sonst den Temperaturwert als Ganzzahl.

3.3 Drucksensor

Vorbereitung: Herstellen einer elektrischen Verbindung zwischen PRESS und Pin P83 siehe Grundkonfiguration.
Funktionsweise:
Der Drucksensor ändert in Abhängigkeit des Luftdrucks seine Ausgangsspannung. Die Messung dieser Ausgangsspannung wird über Kanal AN11 (Pin P83) des ADCs abgewickelt.

Die Messung verwendet keine Interrupts. Da jeder Sensor, bedingt durch die Herstellung, eine Kalibrierung benötigt, muss diese über den Korrekturfaktor PRESS_COR_FAC durchgeführt werden. Im Beispiel-Anhang unter Punkt 7.2.3 ist die genaue Vorgehensweise erläutert.

Funktionen:

```
void initPressure(void);
```

Initialisiert AD-Wandler für die Verwendung des Drucksensors. Konfiguriert Kanal 11 des ADCs und verwendet Pin P83 als analogen Eingangspin.

```
float getPressure();
```

Liefert den absoluten Luftdruck in kPa als Gleitkommazahl.

```
int sgetPressure(char * szPressString);
```

Schreibt den absoluten Luftdruck des Drucksensors als formatierte Zeichenkette in den übergebenen Parameter und retouniert die Anzahl geschriebener Zeichen bzw. -1 im Fall, dass der Druckwert nicht im Plausibilitätsbereich von 70 - 110 kPa liegt. In diesem Fall sollte die elektrische Verbindung zum Sensor geprüft werden, oder sicher gestellt werden, ob die Versorgungsspannung auf 5V eingestellt ist. Das übergebene Feld muss mindestens die Größe von BUFFER_SIZE (10) haben.

Bemerkung:
Es muss darauf geachtet werden, dass der Drucksensor mit 5V versorgt sein muss. (5V Schalter betätigen!)

3.4 Helligkeitsensor

Vorbereitung: Herstellen einer elektrischen Verbindung zwischen LDR und Pin P82 siehe Grundkonfiguration.
Funktionsweise:
Der Helligkeitssensor (LDR) verändert seinen Widerstand in Abhängigkeit der einfallenden Lichtintensität. Die am LDR abfallende Spannung wird über Kanal AN10 des ADCs abgegriffen. Die Messung verwendet keine Interrupts.

Funktionen:

```
void initLDR(void);
```

Initialisiert ADC für die Verwendung des Helligkeitssensors

```
int getBrightness();
```

Liefert einen ganzzahligen Wert zwischen 0 und 255. Dabei entspricht der Wert 0 einem abgedeckten Sensor (dunkel) und der Wert 255 einem bestrahltem Sensor (hell).

3.5 Feuchtesensor

Vorbereitung: Herstellen der elektrischen Verbindung zwischen 0V und Pin P27, AC und PB5 sowie ADC und P84, siehe Grundkonfiguration.
Funktionsweise:
Dieser Sensor wird über eine Wechselspannung von 1kHz versorgt. Um diese Wechselspannung zu generieren, werden 2 Kanäle des PPG-Timers in Anspruch genommen (PPG23 und PPG45). Diese erzeugen auf den Output-Pins PB5 bzw. P27 ein alternierendes Rechtecksignal von 1kHz. Der Sensor verändert in Abhängigkeit der Temperatur und der vorherschenden Luftfeuchte seine Impedanz. Diese wird über den Spannungsabfall am Sensor bestimmt. Durch die hohe Nichtlinearität des Sensors wird seine Kennlinie im Messprogramm interpoliert. Zur Messung des Spannungsabfalls am Sensor wird Kanal AN12 des ADCs herangezogen. Die Messung verwendet den Overflow-Interrupt für Kanal PPG23 des PPG-Timers. Außerdem muss wegen der Temperaturabhängigkeit auch der digitale Temperatursensor initialisiert und angeschlossen werden.
Da sich der Sensor durch den Betrieb auch erwärmt, wird zur korrekten Berrechnung der Luftfeuchtigkeit ein Korrekturfaktor (COR_FAC in humidity.h) herangezogen.

Funktionen:

```
void initHumidity(void);
```

Initialisiert sowohl den ADC als auch das Versorgungssignal, für die Verwendung des Feuchtesensors

```
int igetHumidity();
```

Liefert einen ganzzahligen Wert der relativen Luftfeuchtigkeit in %.

Bemerkung:
Es muss darauf geachtet werden, dass sowohl der digitale Temperatursensor als auch der Feuchtesensor laut Grundkonfiguration elektrisch korrekt verbunden ist.

4 Akkustische Signale

4.1 Buzzer

Vorbereitung: Herstellen der elektrischen Verbindung zwischen BUZZER(1) und Pin P25 sowie BUZZER(2) und GND, siehe Grundkonfiguration.
Funktionsweise:
Der Buzzer wird mit einem Rechteckimpuls (3.6kHz default) über Pin P25 angesteuert. Dazu wird Kanal PPG01 des PPG-Timers herangezogen. Es werden keine Interrupts verwendet.

Funktionen:

`void initBuzzer();`

Initialisiert das Buzzer-Signal über PPG-Timer

`int buzzerOn();`

Lässt den Buzzer ertönen. Liefert als Rückgabewert 1

`int buzzerOff();`

Schaltet den Buzzer aus. Liefert als Rückgabewert 0

`void setBufferFrequency(int freq);`

Verändert die Frequenz mit der der Buzzer ertönt, schaltet Buzzer selbst aber weder ein noch aus. Übergeben Sie der Funktion die Frequenz in Hertz.

5 Zeitsignal

5.1 DCF77

Vorbereitung: Herstellen der elektrischen Verbindung zwischen DCF77 und Pin PA0. Verbinden der DCF77 Antenne mit der 3.5mm Klinkenbuchse am Tochterboard.
Funktionsweise:
Die Antenne FZD01020 empfängt das DCF77 Signal und liefert ein demoduliertes Ausgangssignal. Zur Decodierung des Signals wird der 16bit Free-Run-Timer mit Input Capture Modul verwendet. Wird eine steigende Flanke am Capture-Input-Pin (Pin PA0) erkannt, wird zunächst die Zeit seit dem letzten Puls berechnet, bevor der Timer resetet wird. Diese Zeit liegt typischerweise zwischen 780ms bis 820ms bei einer Pulsbreite von 200ms bzw. zwischen 880ms bis 920ms bei 100ms Pulsbreite. Erst beim Ende des Protokollframes (nach 58 sec.) ist diese Zeit $>$ 1.2 sec.
Dann wird auf die Erkennung der fallenden Flanke umgeschalten. Tritt nun die fallende Flanke auf (typ. nach 100ms bzw. 200ms), wird die Pulsbreite vermessen und die Decodierung entsprechend des Protokolls vorgenommen.
Tritt über 3 Sekunden keine Flankenänderung auf (d.h. es wird kein Signal empfangen), wird ein Flag gesetzt welches anzeigt, dass kein Signal empfangen wird. Dies wird durch den Rückgabewert der zur Verfügung stehenden Funktionen (siehe unten) angezeigt.
Bei der Bestimmung der aktuellen Uhrzeit und des aktuellen Datums wird der Overflow-Interrupt des Free-run-Timers sowie der Input Capture 0/1 interrupt verwendet. Dabei hat der Input-Capture-Interrupt die höchste Priorität(2) im System.

Funktionen:

`void initDCF77(void);`

Initialisiert Input-Capture- und Timerfunktionalität für die Verwendung des Zeitsignals

`int getTime(char * time);`

Speichert im übergebenen Feld (char time[3]) an Stelle 0 die aktuelle Stunde, an Stelle 1 die aktuelle Minute und die aktuelle Sekunde wird an Stelle 2 des Feldes geschrieben. Das übergebene Feld muss mindestens die Länge 3 haben. Die Funktion retouniert 0 falls kein Signal empfangen wird und 1 sonst.

```
int getDate(char * date);
```

Speichert im übergebenen Feld (char date[4]) an Stelle 0 den aktuellen Wochentag (beginnend mit Montag = 1, an Stelle 1 den Tag, an Stelle 2 das Monat und an Stelle 3 das Jahr des aktuellen Datums. Das übergebene Feld muss mindestens die Länge 4 haben. Die Funktion retouniert 0 falls kein Signal empfangen wird, 1 sonst.

6 Kommunikationsschnittstellen

Diese API bietet auf Grundlage eines hardwaremäßig implementierten RS485 Buses und softwaremäßig realisiertem MODBUS-Protokolls, eine einfach zu verwendende Single Master - Multi Slave Kommunikationsschnittstelle an.

6.1 Grundlagen

Zunächst werden hier einige Grundlagen bezüglich des verwendeten Buses und des eingesetzten Übertragungsprotokolls erläutert.

6.1.1 RS485 Bus

Bei RS485 handelt es sich um eine sogenannte differenzielle Schnittstelle. Das bedeutet, dass für jedes Signal zwei Leitungen benötigt werden. Über die eine Leitung wird das Signal unverändert übertragen, auf der anderen Leitung wird das Signal invertiert.
Der Empfänger vergleicht die beiden Signale miteinander und gibt das original Signal aus. Durch dieses Verfahren werden Störungen nahezu eliminiert und man erreicht große Leitungslängen oder hohe Datenraten[1].
Prinzipiell gibt es die Möglichkeit den Bus als Zwei- oder Vierleiter zu realisieren, wobei die Vierleitervariante einen Fullduplex-Betrieb ermöglicht. Beim Tochterboard ist die Verbindung zwischen zwei Teilnehmern als Zweidrahtbus realisiert und ermöglicht aus diesem Grund nur eine Halfduplex-Übertragung.

6.1.2 Der Treiberbaustein

Als Bustreiberbaustein wird am Tochterboard der MAX3471 verwendet.

DE und RE/ sind verbunden und stellen die Enableleitungen dar. DI(DriverInput) ist der Dateneingang des Treibers, d.h. der Datenausgang des Kontrollers. RO(RecieverOutput) ist der Datenausgang des Treibers, also der Dateneingang am Kontroller.
Zum Senden muss DE auf logisch 1 gezogen werden und zum Empfangen muss DE logisch 0 sein. Die folgende Tabelle veranschaulicht das Verhalten des Treibers:

A bzw. B stellen die Busleitungen dar. Am Schematic sind A und B als X2-1 bzw. X2-2 ausgewiesen und entsprechen der zweipoligen Klemmleiste am Tochterboard.

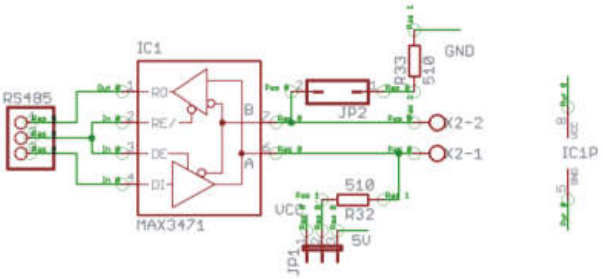

Abbildung 1: Schematic des Bustreibers

Table 1. Transmitting

INPUTS			OUTPUTS	
$\overline{RE}$	DE	DI	B	A
X	1	1	0	1
X	1	0	1	0
0	0	X	Z_D	Z_D
1	0	X	Z_D	Z_D

Z_D = Driver output disabled

Table 2. Receiving

INPUTS			OUTPUT
$\overline{RE}$	DE	A-B	RO
0	0	≥ -0.05V	1
0	0	≤ -0.45V	0
0	0	Open/Shorted	1
1	0	X	Z

X = Don't care
Z = Receiver output high impedance

Abbildung 2: Verhalten Bustreibers

6.1.3 Modbus Protokoll

Modbus ist ein auf einer Master-Slave Architektur basierendes Kommunikationsprotokoll. Mittels Modbus können ein Master und mehrere Slaves miteinander verbunden werden. Von Modbus existieren prinzipiell mehrere Varianten, die auch miteinander kombiniert werden können:

- Serielle Schnittstelle (RS232, RS485)
- Modbus on TCP
- Modbus Plus

Bei der seriellen Datenübertragung unterscheidet man noch einmal zwischen zwei Arten der Übertragung mittels Modbus:

- RTU (Remote Terminal Unit)
- ASCII Mode

Da es sich, um die in der API verwendete Übertragungsart, um eine serielle Kommunikation handelt, wird hier auf die Erklärung von Modbus on TCP und Modbus Plus bewußt verzichtet, und nur auf die tatsächlich verwendete Übertragungsart RTU näher eingegangen.

Modbus Frame

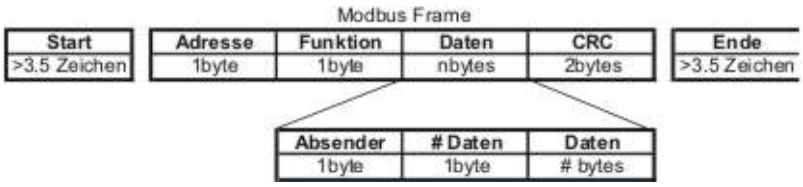

Abbildung 3: Modbus RTU Modus

Den Beginn sowie das Ende eines Modbus-Frames markiert eine Sendepause von mindestens 4 Zeichen. Das eigentliche Datenpaket ist dann wiefolgt aufgebaut:

Adresse: Jedem Teilnehmer am Modbus Netzwerk wird eine eindeutige, gültige Adresse zugeteilt. Über diese Adresse kann der Master einen Slave adressieren.

Funktion: Der Funktionscode (1byte) beinhaltet Information darüber, welche Operation durchgeführt werden soll.

Daten: Das Datenfeld wird seinerseits noch unterteilt in ein Absender-Byte, ein Datenanzahl-Byte und den eigentlichen Rohdaten. Im Absender-Byte ist die Adresse des Absenders gespeichert und im Datenanzahlbyte ist die Anzahl der nachfolgenden Datenbytes abgelegt. Die eigentlichen Roh-Daten können zusätzliche Information zur auszuführenden Operation beinhalten, oder beispielsweise das Ergebnis einer Operation.

CRC: CRC(Cyclical Redundancy Check) beruht auf Polydivision. Die Bitfolge der Daten wird duch ein vorher festzulegendes Generatorpolynom Modulo dividiert, wobei der Rest bleibt. Dieser Rest ist der CRC-Wert und wird an die Nachricht angehängt. Nach Abschluss der Transaktion wird der CRC-Wert vom Empfänger erneut berechnet. Anschließend werden beide Prüfwerte verglichen. Sind die Werte unterschiedlich ist ein Fehler aufgetreten.

6.2 Funktionscodes

Wie schon oben erwähnt beinhalten die Funktionscodes Information über die auszuführende Operation. Da der Funktionscode aus einem Byte besteht, stehen insgesamt 255 mögliche Funktionen zur Verfügung. In dieser API werden im Wesentlichen sechs verschiedene Funktionscode-Blöcke unterschieden.

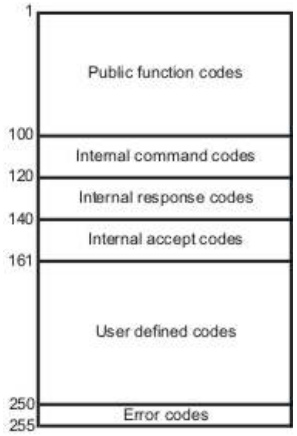

Abbildung 4: Funktionscodeblöcke

Public Function Codes Müssen eindeutig sein und werden von der MODBUS-IDA.org community validiert. Diese Codes sind dokumentiert.

Internal Command Codes: Internal Command Codes sind API interne Codes und bilden alle Funktionalitäten der API ab. Der Codebereich liegt zwischen 100 und 119.

Internal Funktioncodes	
100	LedOn(data)*
101	LedOff(data)*
102	switch_is_on(data)*
103	get_switches()*
104	btn_is_pressed(data)*
105	getButtons()*
106	relais(data)*
107	getTempAnalog()
108	getTempDigital
109	getPressure()
110	getBrightness()
111	getHumidity()
112	BuzzerOn()
113	BuzzerOff()
114	getTime()
115	getDate()
116	setPort(port, value)
117	getPort(port)
118	setPin(port, pin, onoff)
119	getPin(port, pin)

(*) Bei verwendung der gekennzeichneten Codes muss sichergestellt werden, dass die

Defines korrekt gesetzt sind. Alternativ können auch die Codes von 116-119 verwendet werden.

Internal Response Codes Internal Response Codes sind ebenfalls API interne Codes und bilden die Gruppe der Antwort-Codes. D.h.: Sendet der Master einen Modbusframe mit dem Funktionscode 109 an einen Slave, so antwortet dieser mit einem internen Funktionscode von 129. Der FC 129 ist also der Antwort-Code eines Knotens auf einen vorhergehenden Command-Code von 109.
Die Response-Codes sind also die Antworten eines Knotens auf einen vorangegangenen Command-Code, wobei der Response-Code immer um genau 20 größer ist als der vorherige Commando-Code. Daher liegt die Gruppe der Response-Codes auch genau zwischen 120 und 139.

Internal Accept Codes: Die Gruppe dieser Codes wird ebenfalls API intern abgearbeitet und deckt einen Bereich von 140 bis 160 ab. Dabei fallen den beiden internen Accept-Codes 140 und 160 gesonderte Bedeutung zu. 140 ist nämlich der Code, den der Master benutzt, um einen Slave zu fragen, ob dieser ihm etwas mitteilen will.
160 ist jener Code mit dem ein Slave antwortet, falls der Master vorher die Anfrage mit dem Code 140 gestellt hat, der Slave jedoch nichts mitzuteilen hat. Andernfalls könnte der Slave mit jedem beliebigen Code antworten.
Die restlichen Codes (141 - 159) sind die eigentlichen Accept-Codes. Sie informieren den Master über den Erhalt von internen Response-Codes des Slaves. Duch diese Codes weiß also der Master, dass eine Anfrage des Slaves zum Master positiv beantwortet wurde und der Slave die Daten korrekt erhalten hat.

User Defined Codes: Diese Codes können vom Benutzer der API mit eigenen Funktionalitäten belegt werden. Der Bereich der User Defined Codes liegt zwischen 161 und 250. Die Antwort eines Slaves auf den Empfang eines Frames mit User-Defined-Code, wird vom Slave mit demselben Code bestätigt.

Error Codes: Error-Codes beinhalten Informationen über fehlgeschlagene Transaktionen. Retouniert der Slave beispielsweise einen Modbus-Frame mit einem Error-Code von 250, bedeutet dies, dass beim Slave der CRC-Check fehlgeschlagen hat. Wird ein Error-Code von 251 geliefert, ist der CRC-Check beim Master fehlgeschlagen. Die restlichen Error-Codes (252-255) sind reserviert.

6.3 API Kommunikationsframework zur Benutzung von RS485

Die API stellt ein leicht zu konfigurierendes Kommunikationsframework zur Verfügung.
Es beinhaltet bereits eine Master-Slave Architektur die ohne viel Aufwand an die Bedürfnisse des Benutzers angepaßt werden kann.
Die wichtigsten Paradigmen des Frameworks sind:

- Slave antwortet immer auf Befehl des Masters
- Interne Comando-Codes werden durch das Framework automatisch abgearbeitet
- Der Benutzer muss sich um kein Protokoll kümmern
  - automatische Adresserkennung
  - automatischer CRC-Check

6.3.1 Konfiguration des Frameworks

Vorbereitung: Herstellen der elektrischen Verbindung zwischen DI und Pin P94, RE/ DE und P87 sowie RO und P93.
Bitte setzen Sie an jedem Knoten den Jumper JP2.
Weiters werden die Leitungen A und B des RS485 Buses mit den Klemmen X2-1 bzw. X2-2 am Tochterboard verbunden.
Beachten Sie bitte auch, dass für jeden Knoten eine eindeutige Adresse festgelegt werden muss. Dies erfolgt im Konfigurationsfile *master_slave.h*, indem Sie das Define *#define MY_ADDRESS* entsprechend setzen.
Durch setzen eines Compiler-Defines, entweder *#define MASTER* oder *#define SLAVE*, bestimmen Sie, ob der Knoten als Master oder Slave agieren soll.
Um die Funktionen der API für die Master-Slave-Architektur verwenden zu können, müssen Sie in ihrem Quellcode den Header *master_slave.h* includieren.

6.3.2 Funktionen des Frameworks

Funktionen

```
void initMaster();
```

Diese Funktion initialisiert für den Master eine entsprechende UART über die die Buskommunikation stattfindet.Um die Funktion im Anwendungsfall korrekt kompilieren zu können, muss das Define *#define MASTER* in *master_slave.h* definiert sein.

```
void initSlave(rs485_rx_handler_t rhandler);
```

Diese Funktion initialisiert einen Knoten als Slave. Sie setzt die entsprechende UART für die Buskommunikation auf. Um die Funktion im Anwendungsfall korrekt kompilieren zu können, muss das Define *#define SLAVE* in *master_slave.h* gesetzt sein.

```
int Send2Slave(char slave, char code, char * msg, char * return_msg, int len);
```

Diese Funktion wird von einem Master verwendet, um einen Modbusframe an einen Slave zu senden. Der Parameter slave repräsentiert die Adresse des Slaves an den die Nachricht geschickt werden soll. Der Parameter code beinhaltet den anzuwendenden Funktionscode. Der Funktion werden durch den Parameter msg die eigentlichen Daten übergeben. In den Parameter return_msg befinden sich

nachdem die Funktion retouniert, die Empfangsdaten des Slaves. Im 5ten Parameter wird die Länge der zu übertragenen Daten, die im Parameter msg übergeben werden, angegeben. Die Funktion selbst retouniert die Anzahl der geschriebenen Zeichen in return_msg im Fall einer geglückten Kommunikation und 0 Falls der Slave nicht in der vorgegebenen Zeit antwortet, und dadurch in ein Timout läuft.

Im Gegensatz zum Master kann ein Slave von sich aus keine Kommunikation beginnen. Will nun der Slave dem Master aber doch etwas mitteilen, so bietet diese API eine Funktion an, mit Hilfe derer der Slave den Master bestimmte Informationen übegeben kann.
Voraussetzung dafür, dass die Daten letzendlich zum Master übertragen werden, ist, dass der Master selbst eine Anfrage an den Slave stellt. Diese Anfrage wird mit dem Functionscode 140 eingeleitet. Darauf hin sendet der Slave dem Master einen Modbusframe mit den Daten, die der Slave vorher(!) durch aufrufen der Funktion TellMaster(char code, char * data, int len) bereitgestellt hat.

```
int TellMaster(char code, char * data, int len);
```

Diese Funktion ermöglicht einem Slave, dem Master Daten zu übermitteln, wenn dieser eine Anfrage an den Slave stellt (Funktionscode=140).
Wird diese Funktion vom Slave nie aufgerufen, antwortet der Slave automatisch, durch die innere Codeverarbeitung, mit dem Funktionscode 160. Andernfalls werden dem Master exakt diese Daten, die dieser Funktion übergeben werden an den Master geschickt.
Der Parameter code beinhaltet den zu übermittelnden Funktionscode, data die Daten(max. 250 byte) und len die Anzahl der aus data zu übertragenen bytes. Die Funktion retouniert 1 falls die Daten dem Master auf Anfrage hin übergeben werden können. Existieren allerdings, vom Master noch nicht abgeholte Daten im Speicher des Slaves, retouniert die Funktion 0. Dadurch wird sichergestellt, das keine Daten die dem Master übergeben werden sollen verloren gehen.

6.4 Master

Wie schon erwähnt, wird eine Kommunikation am RS485-Bus immer duch den Master initiert. Ein Slave kann von sich aus nie eine Kommunikation beginnen.
Um dies zu gewährleisten, erwartet der Master in dieser API also auf jedes Komando das zum Slave geschickt wird, eine Antwort. Damit der Master eine Nachricht an einen Slave übermitteln kann, steht wie gesagt, die oben beschriebene Funktion Send2Slave() zur Verfügung. Ich möchte diese Funktion hier noch etwas erläutern.
Der Funktion wird neben der Adresse des Slaves, des auszuführenden Codes und der zu übermittelnden Daten auch ein Feld vom Typ char übergeben, welches die Antwortdaten des Slaves enthält, nachdem die Funktion retouniert. Die Antwortdaten haben folgenden Aufbau:
Im ersten Byte von rx_data, das im Übrigen mindesten 255 Bytes groß sein muss, steht

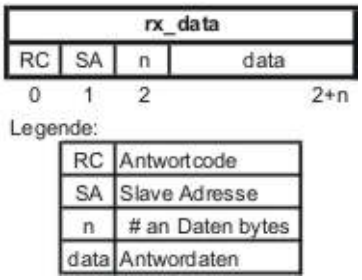

Abbildung 5: Returnmessage

der Return-Code des Slaves, im zweiten Byte befindet sich die Slaveadresse, von der empfangen wurde. Im dritten Byte wird die Anzahl der returnierten Daten gespeichert, und ab dem vierten Byte stehen optional die Daten die der Slave retunierte.

Die Funktion Send2Slave erwartet immer eine Antwort vom Slave und blockt solange bis der Slave geantwortet hat oder maximal bis zum festgelegtem Timout von 1,3 Sekunden.

Kommunikation aus Sicht des Masters:
Aus Sicht des Masters sind 3 verschiedene Kommunikationsszenarien denkbar:

1. Master sendet internen Kommando-Code
2. Master sendet User defined Code
3. Master sendet eine Anfrage (Code=140)

Senden eines internen Kommando-Codes Setzt der Master eine Nachricht mit internen Kommandocode an einen Slave ab, so antwortet das Framework des Slaves automatisch auf solch einen Befehl indem der gesendete Funktions-Code um 20 erhöht wird. D.h. sendet der Master den Funktionscode 109 (getPressure) and den Slave, so antwortet dieser mit 129 als Responsecode und den dazugehörigen Daten. Das Feld rx_data hätte in diesem Fall also folgenden Aufbau: Im ersten Byte befindet sich

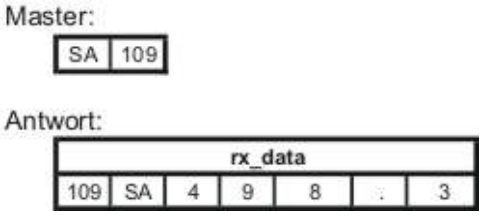

Abbildung 6: Returnmessage bei internen Kommandocodes

der Response Code auf den vorher versendeten Kommando-Code von 109, also 129. Im zweiten Byte steht die Absenderadresse, im dritten die Anzahl der Datenbytes

und ab dem vierten Byte die eigentlichen Daten.
Hier hätte also der Funktionscode 109 beim Slave einen Druck von 98.3 kPa ermittelt und 4 Zeichen in rx_data geschrieben.

Senden von User Defined Code Wird vom Master eine Nachricht abgesetzt, in der der Funktionscode der Gruppe der User Defined Codes angehört, antwortet das Framework des Slaves ebenfalls automatisch auf den Empfang einer solchen Nachricht. Allerdings nicht mit Daten, sondern lediglich mit einer Empfangsbestätigung, die besagt, dass der Slave die Daten zur Weiterverarbeitung erhalten hat.

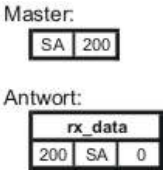

Abbildung 7: Returnmessage bei User Defined Codes

Bei User Defined Codes wird der selbe Code als Bestätigung zurückgesendet. Das Framework beim Slave leitet die Nachricht des Masters indes weiter, damit sie bearbeitet werden kann.

Senden einer Anfrage Sendet der Master eine Anfrage an den Slave (Functioncode=140) so kann die Antwort des Slaves wiederum 3 mögliche Ausprägungen haben.

1. Slave hat nichts mittzuteilen
2. Slave antwortet mit User Defined Code
3. Slave antwortet mit internem Kommando-Code

Slave hat nichts mittzuteilen In diesem Fall hat der Slave seine Funktion TellMaster() noch nicht aufgerufen und das Framework des Slaves antwortet auf die Anfrage des Masters automatisch mit dem Funktionscode 160.

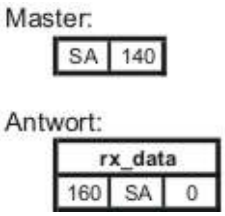

Abbildung 8: Returnmessage wenn Slave nichts mitteilen will

Slave antwortet mit User Defined Code Antwortet der Slave auf die Anfrage des Masters mit einem User Defined Code, so returniert die Funktion Send2Slave() bei fehlerfreier Übertragung wie gewohnt mit 1 und im Feld rx_data steht im ersten Byte der Übermittelte Code, im zweiten Byte wieder die Adresse des

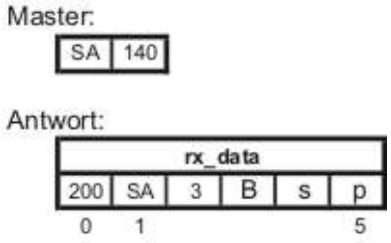

Abbildung 9: Returnmessage, Slave antwortet mit UserDef-Code

Absenders und danach folgen optional die Daten.
Wie vielleicht auffällt, kann ein User Defined Code auf unterschiedliche Weise beim Master interpretiert werden. Einmal als Bestätigung auf einen gesendeten User Defined Code und einmal als Antwort auf eine Anfrage beim Slave.
Das Kommunikationsframework dieser API bietet dem Benutzer die Freiheit selbst zu entscheiden, wie die Weiterverarbeitung auf Seiten des Masters aussieht.

Slave antwortet mit Kommandocode Returniert ein Slave auf die Anfrage des Masters einen internen Kommandocode, wird dieser vom Framework des Masters automatisch beantwortet.

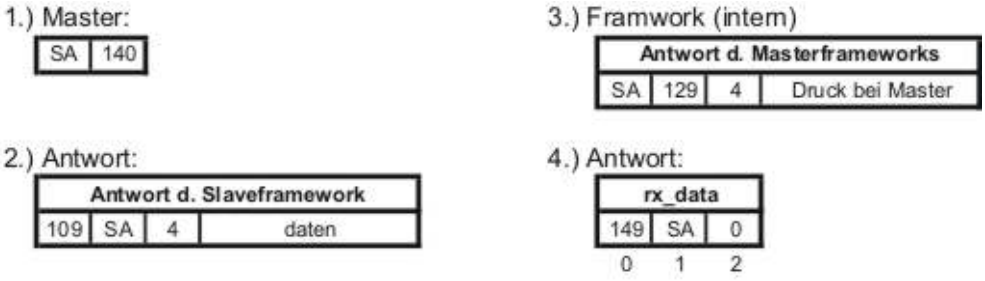

Abbildung 10: Returnmessage, Slave antwortet mit Kommando-Code

1.) Master stellt Anfrage an Slave, ob dieser ihm etwas mitteilen möchte. Code =140
2.) Framework des Slaves antwortet mit internem Kommando-Code. Code=109
3.) Masterframework stellt fest, dass Slave am Druck interessiert ist, ermittelt den aktuellen Druck und retouniert automatisch an den Slave. Code=129
4.) Slaveframework sendet Empfangsbestätigung über Erhalt der Daten. Code=149

6.5 Slave

Ein Slave wird hardwaremäßig genauso konfiguriert wie ein Master(siehe Konfiguration). Beim Slave muss lediglich darauf geachtet werden, dass das Define *#define ADDRESSE xxx* im Header *master_slave.h* im Netzwerk eindeutig ist.
Nach der Initialisierung durch die Funktion initSlave(rs485_rx_handler_t rhandler),

steht das Slaveframework zur vollen Verfügung. Der Funktionspointer der dieser Funktion übergeben wird, wird vom Framework dann aufgerufen, wenn es sich bei empfangenen Nachrichten vom Master um User-Defined-Codes handelt, oder der Slave vorher einen internen Kommandocode abgesetzt hat.
Alle Nachrichten vom Master, die als Funktionscode einen internen Kommandocode besitzten, werden durch das Slaveframework automatisch abgearbeitet.

Beispiel1: (Slave empfängt internen Kommandocode.) In diesem Fall erfolgt die Abarbeitung und Rückmeldung an den Master vollautomatisch durch das Framework.

Beispiel2: (Slave empfängt User Defined Code) Unter diesen Umständen wird vom Slaveframework, dem Master zunächst eine Empfangsbestätigung retuniert, und dann am Slave der Funktionspointer rhandler aufgerufen.
In dieser Funktion hat nun der Benutzer die Möglichkeit auf empfangene Codes zu reagieren und diese weiterzuverarbeiten.

Beispiel3: (Slave empfängt Anfrage) Empfängt der Slave eine Nachricht vom Master mit Funktionscode 140, wird vom Framework des Slaves dem Master entweder der Funktionscode 160 zurückgeliefert, in diesem Falle will der Slave nichts mitteilen, oder das Framework erkennt, dass der Slave über die Funktion TellMaster(); Daten zur Übermittelung an den Master zur Verfügung gestellt hat. In diesem zweiten Fall kommt es auf den Funktionscode an, der vom Slave der Funktion TellMaster() übergeben worden ist.
War dies ein interner Kommandocode, wird das Framework des Masters dies erkennen und automatisch die entsprechenden Daten an den Slave zurückliefern. Das Framework des Slaves wiederum, wird nun die Daten durch Aufruf des Funktionspointers rhandler weiterreichen. Der Benutzer hat also die Möglichkeit auf die Daten zu reagieren.
War der übergebene Funktionscode hingegen ein User Defined Code, wird vom Slaveframework keine Antwort erwartet.

Die Funktion TellMaster(char code, char *data):
Wie schon öfter erwähnt, bietet diese Funktion einem Slave die Möglichkeit Daten an den Master zu übermitteln. Denken Sie bitte daran, das die Funktion 0 zurückliefert, falls der Master ältere Daten noch nicht abgeholt hat. Besteht hingegen die Möglichkeit dem Master Daten mitzuteilen, liefert die Funktion 1 zurück.

7 Programmbeispiele

Hier werden einige Programmierbeispiele für die Verwendung der API vorgestellt.

7.1 Simple I/O

7.1.1 Lauflicht

```
/* Examplecode for using KNX_Calibur_Tochter API LEDs *
*  Implements a simple running light using Portmapping *
*/

/* tochter_config.h */
//use Portmapping
#define LEDS   PDR7 //set LEDs to Port 7
#define LEDS_DIR DDR7 //set correspondign direction port

/*main.c*/
#include "tochter_config.h"
#include "io_operations.h"

void main(void)
{
    int a,b,i=0;
    while(1)
    {
   for(i=0; i<8; i++)
   {
      for (a=0;a<512;a++) {  //delay
         for (b=0;b<512;b++) {
         }
      }

        if(i == 0){
         setPin(LEDS, i, 1);
         setPin(LEDS, 7, 0);
      }else if (i > 0 && i < 7){
         setPin(LEDS, i, 1);
         setPin(LEDS, i-1, 0);
      }else if (i == 7){
         setPin(LEDS,i, 1);
         setPin(LEDS,i-1, 0);
         i=0;
      }
   }
    }
}
```

7.1.2 Buttons und Leds

Dieses Beispiel sollte das Pinmapping erklären. Dabei wird das Port 8 des Mikrokontollers sowohl für Leds als auch Buttons benutzt. Das Programm läßt Led n leuchten falls Button n gedrückt wird.

```
/**
 *  Example shows how to work with Pinmapping
 *  4 Buttons and 4 Leds are connected to
 *  Port7 of the microcontroller by this way:
 *  Led1:Led4 to PDR8_0:PDR8_3 and
 *  Btn1:Btn4 to PDR8_4:PDR8_7
 **/

/* tochter_config.h */
#define LEDS       0x8
#define LED1       0x0
#define LED2       0x1
#define LED3       0x2
#define LED4       0x3

#define BUTTONS      0x8
#define BTN1       0x4
#define BTN2       0x5
#define BTN3       0x6
#define BTN4       0x7
```

```
/* main.c */
#include "tochter_config.h"
#include "io_operations.h"

void main(void)
{
  initTochter();
  while (1){
    if(getPin(BUTTONS, BTN1)) //is BTN1 pressed?
      setPin(LEDS, LED1, 1);  //led1 on
    else
      setPin(LEDS, LED1, 0);  //led1 off
    if(getPin(BUTTONS, BTN2))
      setPin(LEDS, LED2, 1);
    else
      setPin(LEDS, LED2, 0);
    if(getPin(BUTTONS, BTN3))
      setPin(LEDS, LED3, 1);
    else
      setPin(LEDS, LED3, 0);
    if(getPin(BUTTONS, BTN4))
      setPin(LEDS, LED4, 1);
    else
      setPin(LEDS, LED4, 0);
  }
}
```

7.2 Sensoren

7.2.1 Analoger Temperatursensor

Dieses Beispiel zeigt den Umgang mit der API um den Temperaturwet des analogen Temperatursensors auszulesen. Immer wenn Button 1 gedrückt wird, soll der Temperaturwert auf die serielle Schnittstelle ausgegeben werden.

```
/**
 *  Example gets temperature from analog sensor
 *  Each time Button 1 is pressed, the value of the sensor
 *  is written to UART0
 *  Temp_AN to PDR8_1  and
 *  BTN1 to PDR8_80
 **/

/* tochter_config.h */
#define BUTTONS     8
#define BTN1        0


/* main.c */
/** includes for UART **/
#include "uartapi.h"
#include "error.h"
#include "ret.h"

#include "tochter_config.h"
#include "io_operations.h"
#include "sensors.h"

/* Prototypes */
void dataReady(byte port, byte data);
void dataSend(byte port);
void errorHandler(byte port, byte code);

void main(void)
{
  int pressedMore = 0;
  /* initialisations */
  initTochter();
  initTempAnalog();
  initUART0();

  // init UARTS 0
  if (uart_init(FALSE, UART_0,UART_DATA_8,UART_STOP_1,UART_BPS_19200,UART_PARITY_NONE,dataReady,dataSend,errorHandler)!=
    bailOut();
  }

  while (1){
```

```
    char temp_value[BUFFER_SIZE];
    if(getPin(BUTTONS, BTN1)){
      if(!pressedMore){
        sgetTempAnalog(temp_value);
        printfUART(UART_0, "Temperatur=%s\r\n", temp_value);
        pressedMore = 1;
      }
    }else
      pressedMore = 0;
  }
}

void dataReady(byte port, byte data){}

void errorHandler(byte port, byte code)
{
  printfUART(UART_DEBUG,"Error on port %x code %x\r\n",port, code);
}

void dataSend(byte port) {}
```

7.2.2 Digitaler Temperatursensor

In diesem Beispiel werden die Werte des analogen und digitalen Temperatursensors verglichen. Ist der Wert des digitalen Sensors höher, wird Led1 eingeschaltet. Ist der Wert des analogen Sensors höher, wird Led1 wieder ausgeschalten.

```
/**
 *  Example compares temperature values from digital and
 *  analog sensor to each other. If value of digital sensor
 *  is higher, Led1 is switched on. Else Led1 is switched off.
 *  Temp_AN to PDR8_1,  Temp_DIG to PDR8_0 and
 *  LED1 to PDR7_0
 **/

/* tochter_config.h */
#define LED1      70


/* main.c */
#include "tochter_config.h"
#include "io_operations.h"
#include "sensors.h"

void main(void)
{
  int temp_dig;
  float temp_an;

  /* initialisations */
  initTochter();

  while (1){
    temp_an = getTempAnalog();
    temp_dig= iRead_Temperature();

    if(temp_dig > temp_an){
      setPin(LEDS, LED1, 1);    //Led1 on
    }else
      setPin(LEDS, LED1, 0);    //Led1 off
  }
}
```

7.2.3 Drucksensor (Kalibration)

Der Drucksensor muss zunächst für jedes Board kalibriert werden. Hier wird ein einfaches Codebeispiel zur Verfügung gestellt, mit Hilfe dessen die Kalibrierung kein Problem darstellen sollte. Dieses Programm berechnet den aktuellen Luftdruck über 1 Minute und ermittelt den Mittelwert des gemessenen Luftdrucks und gibt diesen an UART1 aus. Dieser sollte dann mit einem geeichten Druckmesser verglichen werden und der Differenzwert wird im Define #define PRESS_COR_FAC x.xx im File sensors.h eingetragen.

```
/**
 *   Calculates mean value of pressure and writes it
 *   to UART1.
 *   Get the value of the define PRESS_COR_FAC  by use the equation:
 *   PRESS_COR_FAC = mean value - real pressure value
 *   PRESS to PDR8_3
 **/

/* sensors.h */
#define PRESS_COR_FAC  0.0

/* main.c */
/** includes for UART **/
#include "uartapi.h"
#include "error.h"
#include "ret.h"

#include "tochter_config.h"
#include "sensors.h"

/* Prototypes */
void dataReady(byte port, byte data);
void dataSend(byte port);
void errorHandler(byte port, byte code);
float CalculateMeanPressure();
int float2String(float nr, char * str);

void main(void)
{
  char str[BUFFER_SIZE];
  /* initialisations */
  initTochter();
  initPressure();
  initUART0();

  // init UARTS 0
  if (uart_init(FALSE, UART_0,UART_DATA_8,UART_STOP_1,UART_BPS_19200,UART_PARITY_NONE,dataReady,dataSend,errorHandler)!=
    bailOut();
  }

  printfUART(UART_0, "Start meassurements... Please wait!!..\r\n");
  float2String(CalculateMeanPressure(), str);
  printfUART(UART_0, "\r\nPressure(mean)=%s kPa\r\n", str);

  while(1)
  {
  }
}

void dataReady(byte port, byte data)
{
}

void errorHandler(byte port, byte code)
{
  printfUART(UART_DEBUG,"Error on port %x code %x\r\n",port, code);
}

void dataSend(byte port) {
  printfUART(port,"DataSend\r\n");
}

float CalculateMeanPressure()
{
  float total =0;
  float mean;
  int run, a, b;

  for (run=0; run <= 10; run++)
  {
    printfUART(UART_0, "*");
    for (a=0;a<1512;a++) {  //delay
        for (b=0;b<1512;b++) {
        }
    }
    total += getPressure();
  }

  mean = total / run;

  return mean;
}

int float2String(float nr, char * str)
```

```
{

  return sprintf(str, "%3.4f", nr);
}
```

7.2.4 Feuchtesensor

Dieses Beispiel zeigt wie der Feuchtesensor initialisiert und ausgelesen wird. Der Wert der relativen Luftfeuchtigkeit wird auf UART1 ausgegeben. Beachten Sie bitte, dass der Wert der Feuchtigkeit auf Grund der Spezifikation des Sensors +/- 7% abweichen kann.

```
/**
 *  Shows how to initialize and read out humidity sensor
 *  0V to P27, ADC to GND and AC to PB5 and
 *  Temp_DIG to P80
 *  Please do not forget to connect digital Temp-Sensor
 *  and enable interrupts in your programm!
 **/

/* main.c */
/** includes for UART **/
#include "uartapi.h"
#include "error.h"
#include "ret.h"

#include "tochter_config.h"
#include "sensors.h"

/* Prototypes */
void dataReady(byte port, byte data);
void dataSend(byte port);
void errorHandler(byte port, byte code);

void main(void)
{
  /* initialisations */
  initTochter();
  initHumidity();
  initUART0();

  // enable interrupts
  InitIrqLevels();
  __set_il(7);                  /* allow all levels          */
  __EI();                       /* globaly enable interrupts */

  // init UARTS 0
  if (uart_init(FALSE, UART_0,UART_DATA_8,UART_STOP_1,UART_BPS_19200,UART_PARITY_NONE,dataReady,dataSend,errorHandler)!=
    bailOut();
  }
  printfUART(UART_0, "Humidity= %i percent\r\n", igetHumidity());
  while(1)
  {
  }
}

void dataReady(byte port, byte data)
{
}

void errorHandler(byte port, byte code)
{
  printfUART(UART_DEBUG,"Error on port %x code %x\r\n",port, code);
}

void dataSend(byte port) {
  printfUART(port,"DataSend\r\n");
}
```

7.3 Akkustisches Signal(Buzzer)

7.3.1 Buzzer und LDR

Hier wird sowohl die Initialisierung von Buzzer und LDR, als auch deren Verwendung durch die API gezeigt. Wird es dunkel sollte der Buzzer ertönen, wird es wieder heller so soll er wieder ausgehen.

```
/**
 *  Shows how to initialize the buzzer and LDR
 *  Buzzer rings with different frequencies dependig on the brightness measured by the LDR
 *  Buzzer(1):Buzzer(2) to P25:GND and
 *  LDR to P82
 **/

/* main.c */
#include "tochter_config.h"
#include "sensors.h"    //used for LDR
#include "io_operations.h"  //used for Buzzer

void main(void)
{
  /* initialisations */
  initTochter();
  initLDR();
  initBuzzer();

  buzzerOn();

  while(1)
  {
    int brightness;
    brightness = getBrightness();
    setBuzzerFrequency(brightness);
  }
}
```

7.4 Zeitsignal

7.4.1 Ausgabe der Uhrzeit:

Hier wird die Initialisierung des Zeitsignals DCF77 veranschaulicht. Bei Verwendung des Zeitsignals ist zu berücksichtigen, dass das Signal einige Zeit (ca. 1 min.) benötigt, um die korrekte Zeit anzuzeigen.

```
/**
 *  Initializes the timesignal and shows how
 *  to read out time.
 *  Time is written out to UART1 each time Button1 is pressed
 *  DCF77 to PA0 and Button1 to P70
 *  Enable interrupts in your programm!
 **/

/* tochter_config.h */
#define BUTTONS     7
#define BTN1        0

/* main.c */
/** includes for UART **/
#include "uartapi.h"
#include "error.h"
#include "ret.h"

#include "tochter_config.h"
#include "io_operations.h"
#include "dcf77.h"

/* Prototypes */
void dataReady(byte port, byte data);
void dataSend(byte port);
void errorHandler(byte port, byte code);

void main(void)
{
```

```
  int time[3];
  int hour, min, sec;
  int pressedMore = 0;
  /* initialisations */
  initTochter();
  initDCF77();
  initUART0();

  // enable interrupts
  InitIrqLevels();
  __set_il(7);                  /* allow all levels          */
  __EI();                       /* globaly enable interrupts */

  // init UARTS 0
  if (uart_init(FALSE, UART_0,UART_DATA_8,UART_STOP_1,UART_BPS_19200,UART_PARITY_NONE,dataReady,dataSend,errorHandler)!=
    bailOut();
  }

  while (1){
    if(getPin(BUTTONS, BTN1)){
      if(!pressedMore){
        if(getTime(time)){
          hour = time[0];
          min  = time[1];
          sec  = time[2];
          printfUART(UART_0, "Time= %i:%i:%i\r\n", hour, min, sec);
        }else{
          printfUART(UART_0, "Attention! Time may be incorrect! Only generated by timer\r\n");
          printfUART(UART_0, "Time= %i:%i:%i\r\n", hour, min, sec);
        }
        pressedMore=1;
      }
    }else
      pressedMore = 0;
  }
}

void dataReady(byte port, byte data)
{
}

void errorHandler(byte port, byte code)
{
  printfUART(UART_DEBUG,"Error on port %x code %x\r\n",port, code);
}

void dataSend(byte port) {
  printfUART(port,"DataSend\r\n");
}
```

7.5 Kommunikation über Modbus und RS485

7.5.1 Initialisierung des Masters

Hier wird gezeigt, wie ein Knoten als Master konfiguriert wird.

```
/**
 *  Initializes a knote as Master
 *  Sets up communication interface and timers needed.
 *  comunicates permanently with slave (adr 101) and prints out
 *  recieved message on UART0.
 **/

/* master_slave.h */
#define MASTER

/* main.c */
/** includes for UART **/
#include "uartapi.h"
#include "error.h"
#include "ret.h"

#include "tochter_config.h"
#include "master_slave.h"

/* Prototypes */
void dataReady(byte port, byte data);
void dataSend(byte port);
void errorHandler(byte port, byte code);
```

```
void Comunicate(char code, char * data, int len);
void PrintoutRX_MSG(volatile char * rx_msg);

void main(void)
{
  /* initialisations */
  initTochter();
  initMaster(); //initializes com-interface
  initUART0();

  // init UARTS 0
  if (uart_init(FALSE, UART_0,UART_DATA_8,UART_STOP_1,UART_BPS_19200,UART_PARITY_NONE,dataReady,dataSend,errorHandler)!=
    bailOut();
  }

  // enable interrupts
  InitIrqLevels();
  __set_il(7);                  /* allow all levels          */
  __EI();                       /* globaly enable interrupts */

  while (1)
  {
    Comunicate(BZ_ON,   "",0);   //switch Buzzer on slave on
    Comunicate(GET_PRESS,"",0);   //get pressure value from slave
    Comunicate(GET_BTN,   "",0);   //get value of BTN-register
    Comunicate(GET_TEMPA,"",0);   //get value of analog temp-sensor
    Comunicate(BZ_OFF,    "",0);   //swich Buzzer on slave off
    Comunicate(GET_TEMPD,"",0);   //get digital temp value from slave
    Comunicate(GET_HUM,   "",0);   //get rel. humidity from slave
    Comunicate(140,     "",0);   //ask slave if it wanna tell something
    Comunicate(177,     "",0);   //send user defined code(177) to slave
    Comunicate(25,      "",0);   //send public function code (25) to slave
  }

}

void PrintoutRX_MSG(volatile char * rx_msg)
{
  printfUART(UART_0,"CODE: %i\r\n",   rx_msg[0]);
  printfUART(UART_0,"FROM: %i\r\n",   rx_msg[1]);
  printfUART(UART_0,"#DAT: %i\r\n",   rx_msg[2]);
  if (rx_msg[2] == 1){
    if(rx_msg[0] == 127 || rx_msg[0]==128){     //temperature can be negative
      printfUART(UART_0,"DATA: %i\r\n", (signed char)  rx_msg[3]);
    }else
      printfUART(UART_0,"DATA: %i\r\n",   rx_msg[3]);
  }
  else
    printfUART(UART_0,"DATA: %s\r\n", & rx_msg[3]);
  printfUART(UART_0,"**********************\r\n");
}

void Comunicate(char code, char * data, int len)
{
  volatile char rx_data[MAX_RX_DATA];

  if( Send2Slave(SA1, code, data, rx_data, len) )
    {
      if(rx_data[2] == 0){
        printfUART(UART_0,"Slave %i returns no data\r\n", SA1);
        PrintoutRX_MSG(rx_data);
      }
      else
        PrintoutRX_MSG(rx_data);
    }
    else
      printfUART(UART_0,"Can't reach Slave %i \r\n", SA1);

}


void dataReady(byte port, byte data){}

void errorHandler(byte port, byte code)
{
  printfUART(UART_0,"Error on port %x code %x\r\n",port, code);
}

void dataSend(byte port) {}
```

7.5.2 Initialisierung des Slaves

Das Beispiel zeigt die Initialisierung eines Knoten als Slave. Ein so initialisierter Knoten kann schon mit einem Master kommunizieren. Benutzerdefinierte Kommandos werden durch den Handler, der der Funktion initSlave übergeben wird, abgearbeitet.

```
/**
 *  Initializes a knote as Slave
 *  Sets up communication interface and timers needed
 *  Knote is ready to comunicate with a master, it
 *  awnsers to internal command codes automaticaly over framwork
 *  user definded codes from master would call data485 function.
 **/

/* master_slave.h */
#define SLAVE
#define ADRESSE 101

/* main.c */
#include "tochter_config.h"
#include "io_operations.h"
#include "sensors.h"
#include "master_slave.h"

/* Prototypes for recieving from rs485 */
void data485(byte code, char *data);
void erro485(byte code);

void main(void)
{
  /* initialisations */
  initSlave(data485, erro485, erro485); //initializes com-interface
  initTochter();

  initTempAnalog();
  initLDR();
  initBuzzer();
  initHumidity();

  // enable interrupts
  InitIrqLevels();
  __set_il(7);                  /* allow all levels          */
  __EI();                       /* globaly enable interrupts */

  while (1)
  {
  }
}

/* recieves user defined codes from master*/
void data485(byte code, char *data)
{

}
/* recieves error messages in case of failure in comunication */
void erro485(byte code)
{

}
```

8 Belegte Resourcen

In folgender Tabelle werden die von den Komponenten verwendeten Resourcen des Mikrokontrollers gelistet.

Komponente	belegte Resourcen
Switches	frei wählbar (std. PDR7)
Buttons	frei wählbar (std. PDR7)
LEDs	frei wählbar (std. PDR7)
LDR	ADC Ch. AN10 = P82
Drucksensor	ADC Ch. AN11 = P83
analoger Temperatursensor	ADC Ch. AN09 = P81
digitaler Temperatursensor	Pin P80
Buzzer	16bit PPG Timer Channel (PPG01) PPG output pin PPG1/P25
Feuchtesensor	16bit PPG Timer Channel (PPG23) 16bit PPG Timer Channel (PPG45) PPG output pin PPG3/P27 PPG output pin PPG4/PB5
DCF77 Zeitsignal	16bit I/O-Timer (Input-Capture) Capture Input IN0 / PA0
RS485	UART3 (SIN3, SOT3) P87

9 Schaltpläne:

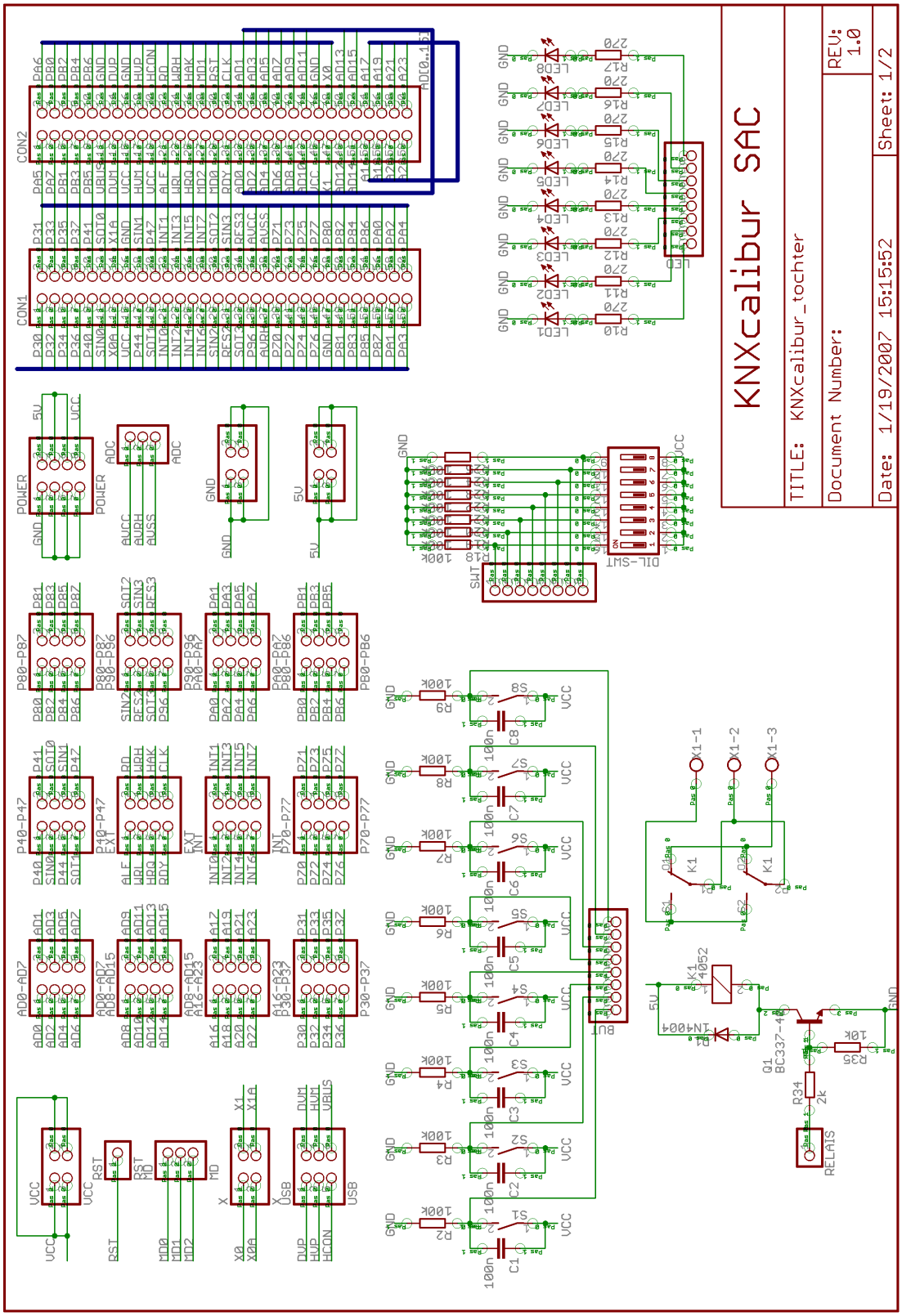

Abbildung 11: Schaltplan a

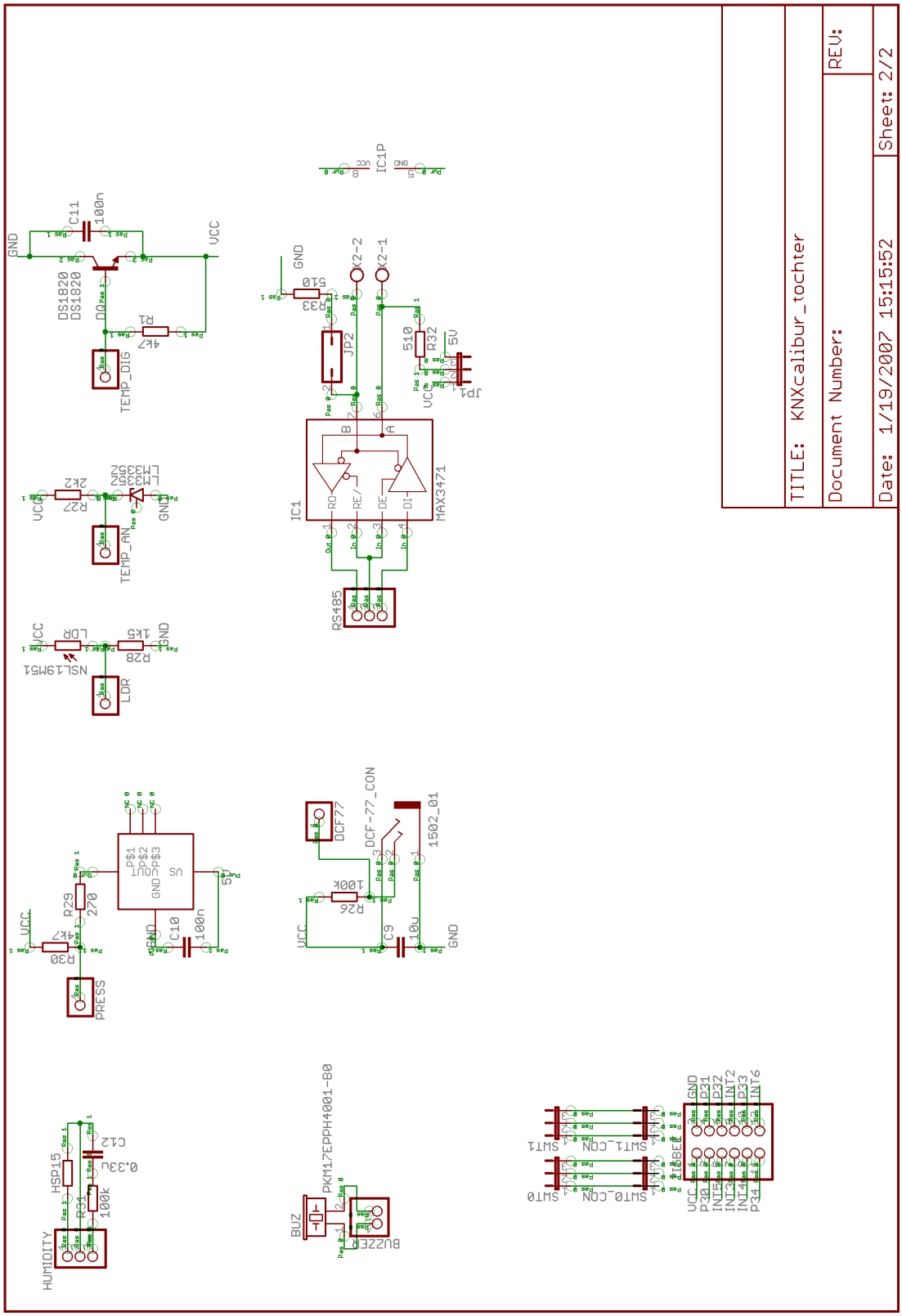

Abbildung 12: Schaltplan b