Einleitung

Auch ich konnte mich der Versuchung nicht entziehen, mit einem Raspberry Pi zu experimentieren.

Dabei haben mir die Möglichkeiten gefallen, unmittelbar vom PC aus auf Peripherie wie I/O-Pins, Interrupt-Pins und insbesondere den TWI-Bus zugreifen zu können.
Also unter Umgehung eines immer wieder neu zu programmierenden Microcontrollers mit seinen eher bescheidenen Debugmöglichkeiten.

Da ich die Netzwerk-/Multimediafähigkeiten des Raspberry zur Zeit eher wenig nutze, ist der Raspberry als Hardware für meine Zwecke völlig unterfordert.

Und so kam der Gedanke auf, auf einem sparsamen (und preiswerten) ATMEGA88 die Funktionalität des Raspberry im Bereich I/O und TWI nachzubilden, also einen "Rasp"-Mega Pi IO zu bauen.

Der Raspberry wurde (so heißt es) entwickelt, um in der Ausbildung benutzt zu werden und Anfängern den Einstieg in das Programmieren zu ermöglichen.

Der "Rasp"-Mega schließt sich diesem Ziel an.
Für einen spielerischen Einstieg in die Programmierung, bei der LED's blinken, Temperaturen ausgelesen werden, sollte die Latte zunächst einmal auf niedriger Höhe aufgehängt werden, damit sich Erfolgserlebnisse einstellen können.

Die Auseinandersetzung mit einem Betriebssystem wie Linux kann zu einem späteren Zeitpunkt nachgeholt werden.

Der ATMega88 muss sich vor dem Raspberry nicht verstecken, er bietet bis zu:

- 16x IO-Pins als programmierbare Ein- oder Ausgangspins
- 16x IO-Pins für externe PinChange Interrupts programmierbar
- 2x IO-Pins für externe, einstellbare Interrupts programmierbar (INT0/INT1)
- 4x Eingänge als 10-Bit ADC
- 6x Eingänge zum Auslesen von DS18B20 Temperatursensoren
- 1x TWI-Bus, der als TWI-Master / TWI-Slave arbeitet
- Die Stromaufnahme des ATMega beträgt im Idle-Modus unter 1mA.

Vom PC aus kann mit beliebiger Software, die via Serieller Schnittstelle senden und empfangen kann (z.B. mit Python - oder im einfachsten Fall mit Hilfe eines Terminal-programms) mit dem "Rasp"-Mega kommuniziert werden.

Das verwendete Protokoll für einen binäre Datenaustausch ist in der Datei "Doku.pdf dokumentiert.

Der Sourcecode in C und ein Hexfile für den ATMega sind in der Datei "raspmega_c.zip" beigefügt.

Im Archiv "python.zip" sind eine Reihe von Programmbeispielen für Python3.2/Tkinter enthalten, mit denen ich getestet habe.

Programmbeispiele in Python

Zum Testen habe ich diverse C-Programme, die ursprünglich auf einem Mikrocontroller ihren Dienst verrichteten, für den "Rasp"-Mega in Python umgeschrieben.

Zunächst sind die grundlegenden Funktionen entstanden, die zum Setzen und Auslesen von Ports/Pins und zum Zugriff auf TWI-Bus und zur Temperaturmessung mit dem DS18B20 erforderlich sind (siehe pin_tools.py).

Und i2cdetect() scannt den TWI-Bus nach empfangsbereiten Devices.

Klassen

- atmega_p.py Klasse für die pollenden Empfang von Daten
- atmega_i.py Klasse für den Empfang von Daten via Interrupt/Hintergrundthread

Module in Python3.2 mit einfacher graphischer Oberfläche (Tkinter):

```
pin_config.py - Die Pins des ATMega als IO konfigurieren
pin_set.py    - Den Pinstatus von Ausgangspins setzen
adc._guipy    - ADC-Eingänge auslesen und Messwerte darstellen
dsread.py     - DS18B20-Temperatursensoren auslesen, Messwerte anzeigen
```

Module in Python3.2 für einige gängige TWI-Slaves

```
hyt221.py  - TWI-Luftfeuchtigkeits-/Temperatursensor auslesen
ds1631.py  - TWI-Temperatursensor auslesen
bmp085.py  - TWI-Luftdruck-/Temperatursensor auslesen
srf02.py   - TWI-Ultraschall-Entfernungsmesser SRF02 auslesen
pcf8574.py - TWI-Portexpander PCF8574 Pins setzen
pcf8583.py - TWI-RTC Uhrzeit einstellen und auslesen
adxl345.py - TWI-Beschleunigungssensor ADXL345 auslesen
```

Beschreibung der Arbeitsweise des Programms auf dem ATMega

Der ATMega fällt bei Untätigkeit in den Sleep-Modus zurück.

Sobald via Serielle Schnittstelle ein Byte empfangen wird, wird ein RX-Interrupt ausgelöst, der Controller wacht auf und fügt das empfangene Byte in einen Empfangs-Ringpuffer ein.

Nun wird versucht, das STARTBYTE zu erkennen, das den Beginn einer Sendung kennzeichnet.

Das zweite empfangene Byte muss eine gültige Frame-Kennung tragen (der Frametyp zeigt auf, welchen Inhalt der Datensatz trägt).

Ist ein gültiger Frametyp erkannt, wird auf das dritte Byte gewartet, das angibt, wieviele Datenbytes noch nachfolgen werden.

Anschließend wird gewartet, bis die Anzahl der erwarteten Bytes eingegangen ist.

Nun wird der vollständige Datensatz vom zuständigen Unterprogramm (welches, darüber entscheidet der Frametyp) weiterbearbeitet.

In der Zwischenzeit können jederzeit weitere Bytes empfangen und in den Empfangs-Ringpuffer eingefügt werden.

Sind die empfangenen Daten verarbeitet (zum Beispiel über die TWI-Schnittstelle weitergereicht), dann kann der nächste Datensatz in Angriff genommen werden, der möglicherweise in der Zwischenzeit empfangen wurde.

Wenn der Empfangs-Ringpuffer leer ist - und alle Jobs abgearbeitet sind, verfällt der Controller wieder in den Sleep-Modus.

Versenden von Daten via TWI-Bus

Wenn ein Frame empfangen wurde, das eine TWI-Transaktion ausführen soll, dann übergibt das Programm zwei Pointer an das TWI-Modul, die auf den Anfang und das Ende desjenigen Datenbereiches im Eingangspuffer zeigen, der versendet werden soll.

Das TWI-Modul versendet nun den übergebenen Datenbereich Interrupt-gesteuert:
Sobald ein Byte versandt ist, veranlasst der TWI-Interrupt das Senden des nächsten Bytes - solange, bis alle Bytes abgeschickt sind.

Der TWI-Master holt bei einem TWI-Read vom angeschlossenen TWI-Slave Daten ab und legt sie in einem für den TWI-Empfang reservierten Ringpuffer ab.

Sobald der TWI-Master das Abholen der Daten erfolgreich beendet hat, werden Anfang und Ende des Datenbereichs via Pointer an das Sende-Modul der Seriellen Schnittstelle übergeben.

Und das Spiel setzt sich fort. Sobald die Serielle Schnittstelle ein Byte erfolgreich versandt hat, wird der TX-Interrupt ausgelöst, der das nächste Byte auf die Reise schickt.
Solange, bis alle Daten verschickt sind.

Timing

Die Kommunikation erfolgt grundsätzlich über die Serielle Schnittstelle, es gibt also genau zwei Gesprächspartner: den PC und µC.

Wenn vom PC aus eine Anfrage an den µC geschickt wird, dann kommt die Antwort in vorhersehbarem Zeitabstand (das können wenige Millisekunden sein, aber bei einigen Sensoren entstehen auch Wandlungszeiten von bis zu 0.8 Sekunden).

Nach dem Absenden einer Anfrage kann der PC pollend auf eine Antwort warten.

Er vertrödelt dabei zwar Rechenzeit, wenn dadurch den Programmablauf aber nicht nennnenswert gestört wird, kann man diesen einfachen Weg wählen.

Nun gibt es aber auch externe Ereignisse, deren Auftreten zeitlich nicht vorhersehbar ist: etwa der Alarm eines Bewegungsmelders, die Betätigung einer Taste.

Um externe Interrupts zeitnah verarbeiten zu können, müsste die Anwendung auf dem PC ständig und in hinreichend engen Zeitintervallen die Serielle Schnittstelle abfragen.

Als Alternative bietet sich die Möglichkeit, einen Hintergrundthread zu starten, der nur eine einzige Aufgabe erfüllt:
In kurzen Zeitabständen zu prüfen, ob Daten empfangen wurden - und sich im negativen Fall wieder für einige Millisekunden in den Sleep-Modus zurückzieht.

Nur wenn tatsächlich Daten eingegangen sind, dann werden diese analysiert und an das zuständige Programmmodul weitergeleitet.

Damit der Hintergrundthread die empfangenen Daten zuordnen kann, sind in jeder Antwort des µC zusätzliche Informationen wie Frametyp, abgefragter Port / Pin / TWI-Adresse enthalten.

Dem Hintergrundthread können - für jeden Frametyp und ggf. auch für jede TWI-Adresse unterschiedliche Callbacks zugeordnet werden, die anschließend für die Bearbeitung der eingegangenen Daten zuständig sind.

Als konkretes Beispiel sei das Auslesen der DS18B20 Temperatursensoren beschrieben.

Um die Sensoren auszulesen, wird zunächst ein Frame versendet, das die Messung auslöst.

Danach benötigen die Sensoren ca. 800ms um den Messwert bereitzustellen.
Die Messwerte werden nach Ablauf der Wandlungszeit automatisch vom ATMega an den PC ausgeliefert.

Das Programm auf dem PC kann zwischen zwei Methoden wählen:

Es kann 800ms in einer Warteschleife verharren bis die Messwerte eingetroffen sind.

Oder es geht - nachdem die Messung angestoßen ist - seiner eigentlichen Arbeit nach.

Die Antwort wird vom Hintergrundthread empfangen, anhand der Framekennung als Temperaturmessung identifiziert und an das Modul weitergereicht, das für die Anzeige oder allgemein für die Weiterverarbeitung der Daten zuständig ist.

Der zuletzt beschriebene Weg erscheint der elegantere zu sein. Aber ...

zu Risiken und Nebenwirkungen

Nachdem die Arbeitsweise des Programmes bekannt ist, wird aber auch deutlich, in welchen Situationen Ungemach drohen kann.

Der ATMega arbeitet mit 3 Empfangspuffern als Ringpuffern von 256 Byte Länge, im ersten werden die via Serieller Schnittstelle empfangenen Daten abgelegt, im zweiten die Daten, die der TWI-Master von seinen TWI-Slaves abholt.

Die Funktion des dritten Ringpuffers wird weiter unten beschrieben und ist für die Risiken und Nebenwirkungen ohne Relevanz.

Zuallererst ist die maximale Datensatzlänge zu beachten.

Intern wird die Anzahl der Datenbytes durch ein Byte (8 Bit unsigned integer) codiert - der maximaler Wert beträgt also 255.

Allerdings muss auch der Header des Frames (3 Byte) in den Ringpuffer passen.

Ein Datensatz kann daher nicht mehr als 256 - 3 Bytes transportieren.

Solange aber die Option besteht, im Hintergrund bereits die nächsten Daten zu empfangen, muss die Datensatzlänge noch weiter reduziert werden.
Das wird empfohlen, bei einer einzelnen Übertragung max. 128 Datenbyte zu schreiben/lesen.

Zu beachten ist dabei das Verhältnis zwischen den Übertragungsraten der Seriellen Schnittstelle und des TWI-Bus.
Die Serielle Schnittstelle ist auf 9600 Baud eingestellt, der TWI-Bus arbeitet mit einem

SCL-Takt von 100KHz.

Das bedeutet, dass über die Serielle Schnittstelle ca. 1.000 Byte/Sekunde versandt werden können, über den TWI-Bus dagegen 10.000 Byte/Sekunde.

Bei einem TWI-Write werden die Daten mit 9600 Baud an den ATMega geschickt - und der leitet sie mit fast der 10-fachen Geschwindigkeit an den TWI-Slave weiter.
Hier drohen keine Überraschungen, der Ringpuffer wird schneller "geleert" als er "gefüllt" werden kann.

Wenn der PC Aufträge versendet, IO-Pins zu setzen etc., dann werden in der Regel 4-6 Byte vom PC an den ATMega übertragen, die Abarbeitung der Jobs erfolgt noch schneller als das Senden über die TWI-Schnittstelle.
Auch diese Situation ist unproblematisch.

Anders sieht es aber aus, wenn neue Frames (auf Dauer) schneller gesendet werden, als sie bearbeitet werden können: dann wird früher oder später der Empfangspuffer von hinter her überschrieben.

Nehmen wir den Fall, dass das jemand auf den Gedanken kommt, den Status eines Ports zu pollen und dazu in einer Endlosschleife Read_Port() an den "Rasp"-Mega sendet.
Dabei werden jeweils 4 Byte an den Rasp-Mega gesendet, der liefert aber 7 Byte zurück.

Nach ca. 200 verschickten Reads hat wird der Ringpuffer überrollt. Es ereignet sich kein Programmabsturz, aber auf z.B. 512 Reads gehen nur 300 'Antworten' ein
(siehe portread-test.py).
Die verbleibenden Sendungen sind durch Überschreiben zerstört worden.

Ähnliches ereignet sich beim ADC-Read. Hier werden 4 Byte an den µC gesendet, aber 6 Byte werden zurückgeliefert (siehe adcread-test.py).

Eine vergleichbare Schwierigkeit tritt auf, wenn zu viele TWI-Reads mit größerem Payload ausgeführt werden:
Der PC sendet als TWI-Read ca. 5-7 Byte, aber der "Rasp"-Mega muss z.B. 128 Byte an Daten zurückliefern.

Wenn der PC 36 Frames (je 7 Byte) angefordert hat, dann ist der Ringpuffer fast voll.

Der "Rasp"-Mega konnte in der Zwischenzeit aber maximal 2 Pages (256 Byte) zurücksenden, damit sind erst 14 Byte aus seinem Empfangspuffer abgearbeitet und frei geworden.

Wenn der PC noch 2 oder 3 weitere Frames sendet, dann wird der Ringpuffer von hinten her überschrieben (siehe eeprom.py).

Nun, die beschriebenen Probleme sind nicht wegzudiskuttieren ... aber lösbar.

Um sie grundsätzlich auszuschließen, sollte kein weiterer Job versandt werden, wenn ein Read (gleichgültig ob TWI-, Port- oder ADC-Read) noch nicht abgeschlossen ist.

Folgender einfacher Mechanismus verschafft Abhilfe:

Bei jedem Aufruf eines Read wird ein Flag gesetzt, das erst dann wieder gelöscht wird, wenn die korrespondierende Antwort eingegangen ist.
Vor jeder Transaktion wird geprüft, ob dieses Flag gesetzt ist, und wenn ja, dann wird gewartet, bis es (beim Empfang der Antwort) gelöscht worden ist.

In der Klasse atmega_i.py sind die beschriebenen Vorkehrungen getroffen.

Die Klasse atmega_p.py kennt all die Probleme nicht, kann aber auch nicht auf externe Interrupts reagieren.

Interrupts

Die Pin-Interrupts des ATMega können dazu eingesetzt werden, um auf externe Ereignisse zu reagieren.

Ausschließlich für die Pins PD.2 bzw. PD.3 können die Interrupts INT0 bzw. INT1 eingesetzt werden, die konfigurierbar sind auf 4 verschiedenen Level bzw. Flankenwechsel:

Pin auf Low, Pin von Low nach High, Pin von High nach Low oder beliebiger Flankenwechsel.

Der "Rasp"-Mega deaktiviert einen INT0/INT1 jewils automatisch, wenn die zugehörige ISR ausgerufen wird.
Das Programm muss den Interrupt jeweils - bei Bedarf - neu setzen.

Für alle Pins aller Ports können PinChange-Interrupts aktiviert werden.

Ein PinChange wird ausgelöst, wenn sich der Status eines (scharfgeschalteten) Pins verändert - und der Rasp-Mega löscht in diesem Falle den Interrupt nicht automatisch.

Er eignet sich daher gut für Bewegungsmelder etc., also Schalter, die über nicht prellenden Kontakte verfügen.

Die Interrupts sind in der gewählten Implementierung und bedingt durch die Performance der Seriellen Schnittstelle nicht für (Zeit-) Messungen geeignet.
Interupts werden auf dem "Rasp"-Mega erst dann behandelt, wenn laufende Übertragungen abgeschlossen sind.

Temperaturmessung mit DS18B20

Zur Temperaturmessung mit dem DS18B20 habe ich eine Methode gewählt, bei der die Kenntnis der Seriennummern der Sensoren nicht erforderlich ist.

Jeder Sensor ist an einem eigenen Pin angeschlossen, die DQ-Leitung ist jeweils mit 4.7k nach VCC abgeschlossen.
Um die Sensoren auszulesen, wird eine Pinmaske übergeben, die festlegt, an welchen Pins die Sensoren gesucht werden sollen.
Im Zweifelsfall wird 0xFF eingeben, dann wird alle Pins abgeklappert.

Wird an einem Pin keine Antwort gelesen (wegen fehlenden Pullups oder fehlendem Sensor), dann wird der Wert -999 zurückgeben.

Messwerte werden in aufsteigender Folge der (in der Pinmaske ausgewählten) Pins zurückgeliefert.

Der "Rasp"-Mega sendet ohne weitere Aufforderung nach ca. 800ms die Messwert an den PC zurück.

Der dritte Ringpuffer

Neben den beiden Ringpuffern, die die empfangenen Daten der Seriellen Schnittstelle und der TWI-Schnittstelle aufnehmen, wird ein dritter Ringpuffer benutzt.

Er erfüllte seine Aufgabe, wenn der Rasp-Mega als TWI-Slave in einem TWI-Bus agieren, und das macht er immer, solange er nicht selbst aktiver TWI-Master ist.

Wenn an die TWI-Adresse des "Rasp"-Mega Daten gesendet werden, dann werden diese im dritten Ringpuffer zwischengespeichert und nach Abschluss der Sendung über seine Serielle Schnittstelle an den PC weitergeleitet.

Diesen Modus nutze ich zum Debuggen oder Loggen in einer Umgebung von Microcontrollern, in der keine Serielle Schnittstelle zur Verfügung steht, dafür aber ein TWI-Bus mit einem TWI-Master.

Der TWI-Master kann hier Log-Daten etc. an die TWI-Adresse 0x04 (default) senden, ein Terminal-Programm auf dem PC zeigt die Daten anschließend an.

Da keine Multi-Master-Fähigkeiten vorhanden sind, darf der Rasp-Mega in dieser Umgebung (eigentlich) nicht als Master aktiv werden.
Sonst können (in seltenen Fällen) Kollisonen auf dem TWI-Bus auftreten.

Der TWI-Slave ist während der Übertragung der empfangenen Daten an den PC inaktiv, nimmt also in dieser Zeit keine neuen Daten an.

Experimentierboard

Die minimale Hardware für einen "Rasp"-Mega kann erfreulich bescheiden ausfallen:

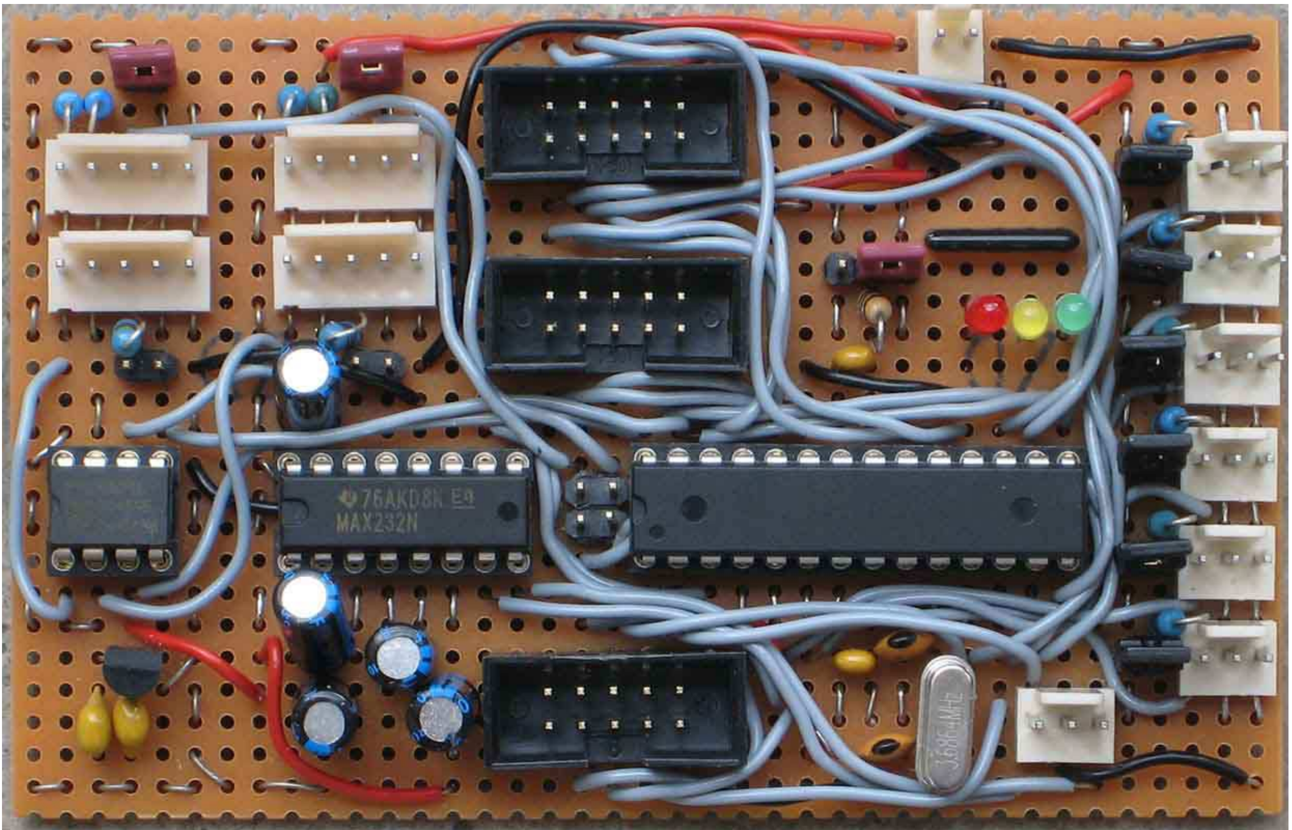
1 Breadboard, 1 ATMega88, 1 Quarz 3.686.400 MHz, 2x Kerko 20pf, 2x 4.7K

Diese Teile sollten in jeder Bastelkiste auf ihren Einsatz warten.

Es darf natürlich auch komfortabler werden.
Als Anhang füge ich eine schematische Darstellung meines Experimentierboard bei - und ein Foto der fliegend verdrahteten Schaltung auf 3er Lochrasterplatine.

Vorhanden sind dort:

- PortB, PortC, PortD auf 2x5pol. Wannenstecker herausgeführt
- PortB, alle Pins auf 3pol. Stiftleisten mit zusteckbaren Pullup 4.7k herausgeführt (zum Anschluss von DS18b20, Bewegungsmelder etc.)
- TWI-Bus 5.0V mit option. 5.0V-Schiene sowie Interrupt-Leitung - auf PD.3 zusteckbar
- TWI-Bus 3.3V mit option. 3.3V-Schiene sowie Interrupt-Leitung - auf PD.2 zusteckbar
- LDO für 3.3V-Schiene
- Referenzspannung für ADC auf VCC steckbar - alternativ externe Ref.spannung steckbar



Viel Spaß beim Experimentieren,

Michael S.

MCP 1703

+3.3 V
+5.0 V
1uF
1uF

TWI-Bus 3.3 Volt
auf 5pol. Stiftleisten

Pullup für INT per
Jumper zuschakltbar

3.3V per Jumper
zuschakltbar

TWI-Bus 5 Volt
auf 5pol. Stiftleisten

Pullup für INT per
Jumper zuschakltbar

5.0V per Jumper
zuschakltbar

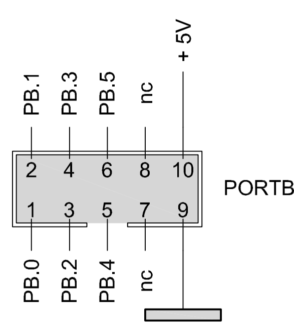
Referenzspannung
wählbar

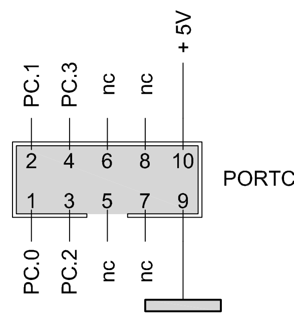
MAX232
P82B96
ATMega88
RS232
3.6864MHz
100uH
10nF
22pF

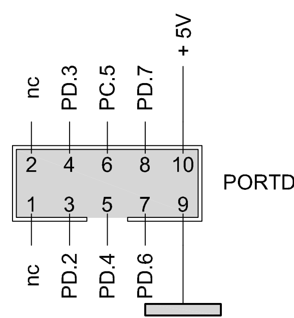
4 = PD.2
5 = PD.3
6 = PD.4
11 = PD.5
12 = PD.6
13 = PD.7

14 = PB.0
15 = PB.1
16 = PB.2
17 = PB.3
18 = PB.4
19 = PB.5

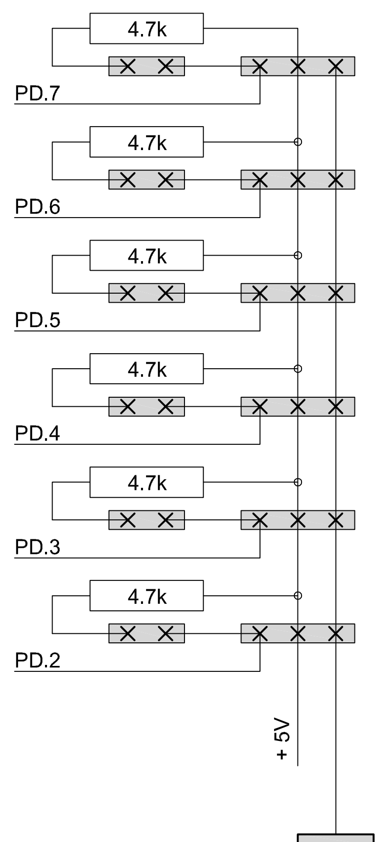
23 = PC.0
24 = PC.1
25 = PC.2
26 = PC.3







3 Ports jeweils
auf 2x5pol. Pfosten-
steckern herausgeführt



PD.2 - PD.7 mit steckbarem
Pullup 4.7k
auf 3pol. Stiftleisten herausgeführt

"Rasp"-Mega
Schema Experimentierboard
Stand: 12.08.2013