Bernd vom Berg
Peter Groppe

Projekt Nr. 275:

„Am langen Draht …. der 1-Wire-Bus“

Grundlagen und Anwendungen des 1-Wire-Busses

(Stand: 17.03.2010, V1.1)

Worum geht's ?

In diesem Projekt werden die Grundlagen des 1-Wire-Busses von MAXIM/DALLAS vorgestellt und an Hand von verschiedenen Beispielen wird dieses Bussystem praktisch eingesetzt.

Nach den Erläuterungen der Grundlagen des 1-Wire-Busses wird zuerst eine kleine Sammlung von ´C´-Funktionen zum Betrieb dieses Kommunikationssystems erstellt (1-Wire-API).

Danach erfolgt der Anschluss eines einzelnen digitalen Temperatursensors, des DS18S20ers, an den Bus. Dieser Sensor wird in seiner Funktion erläutert und dann praktisch betrieben (Entwicklung der notwendigen Betriebsfunktionen in Form der DS1820er-API).

Darüber hinaus wird ein kleines 1-Wire-Experimental-Board vorgestellt, auf dem ein 4-fach A/D-Wandler, zwei 8-fach digital I/O-Porterweiterungen (mit Relais, LEDs und einem Summer) und mehrere digitale Temperatursensoren vorhanden sind, die alle über den 1-Wire-Bus angesteuert werden.
Ergänzt wird der Funktionsumfang dieses Moduls durch ein alphanumerisches LC-Display mit 4 Zeilen zu je 20 Zeichen, das ebenfalls über den 1-Wire-Bus betrieben wird.

Die im Projekt entwickelte ´C´-Betriebssoftware betreibt all diese 1-Wire-Stationen und zu jeder Komponente ist ein umfangreiches Demo-Programm vorhanden.

Die Software ist durchgängig in ´C´ geschrieben (IDE µC/51) und als Hardware-Plattform (1-Wire-Master-Station) wird ein 8051er-Mikrocontroller-System mit einem AT89C51CC03 der Firma Atmel verwendet.
Da alle Quellcodes offen gelegt sind, ist eine Anpassung an andere Mikrocontroller-Familien/Systeme leicht möglich.

I n h a l t

1. Einleitung

2. Der 1-Wire-Bus
2.1 Eigenschaften
2.2 Die Realisierung von 1-Wire-Master-Stationen
2.3 Die 1-Wire-Bus-Hardware
2.3.1 Die Busankopplung
2.3.2 Die Speisung von 1-Wire-Komponenten
2.3.3 Die Hardware-Basis des Projekts
2.4 Das 1-Wire-Datenübertragungsprotokoll
2.4.1 Die 1-Wire-Kommunikationspyramide
2.4.2 Grundlegendes
2.4.3 Die sechs 1-Wire-Basisfunktionen zum Datentransfer
2.4.4 Die Höheren Funktionen
2.4.5 Die Slave-spezifischen Funktionen
2.5 Weitere 1-Wire-Eigenschaften

3. Die 1-Wire-Funktionen in ´C´ für 8051er

4. Die verfügbaren 1-Wire-Komponenten – eine Übersicht

5. Das PT-1-Wire-Experimentalboard (PT-1-Wire-ExBo)

6. Der DS18S20 – Digitaler Temperatur-Sensor
6.1 Allgemeines
6.2 Die DS18S20er-API
6.3 Das DS18S20er-Demo-Programm

7. Der DS2450 – 4-fach A/D-Wandler
7.1 Allgemeines
7.2 Die DS2450er-API
7.3 Das DS2450er-Demo-Programm

8. Der DS2408 – 8-fach digitale I/O-Porterweiterung
8.1 Allgemeines
8.2 Die DS2408er-API
8.3 Das DS2408er-Demo-Programm 1: Relais, LEDs, Summer
8.4 Das DS2408er-Demo-Programm 2: Betrieb eines LC-Displays

9. Die große Gesamt-Demo

10. Literatur

11. Version History

1. Einleitung

Die Verwendung serieller Gerätebus- und serieller Feldbussysteme in den verschiedensten Geräten und Systemen der Automatisierungstechnik, der Unterhaltungselektronik oder auch in Geräten der Haushalts- und Spielzeugindustrie erfreut sich mittlerweile einer immer weiter wachsenden, größeren Beliebtheit.
Auch für die engagierten (Hobby)Praktiker werden solche Datenkommunikations-Konzepte immer interessanter.
Aufgrund der fortschreitenden Halbleitertechnik können sowohl die Hardware- als auch die Softwarekomponenten solcher Datenübertragungssysteme immer einfacher und kostengünstiger in spezielle Chips „gegossen“ oder als zusätzliche ON-Chip-Funktionsbaugruppen in normale Mikrocontroller integriert werden.
Die aktuell interessantesten Vertreter dieser seriellen Bussysteme auf der Geräte- bzw. Feldebene sind sicherlich:

- der SPI-Bus,
- der 1-Wire-Bus
- der I^2C-Bus und der
- CAN-Bus.

In diesem vorliegenden Projekt soll nun das schwerpunktmäßig Augenmerk auf den **1-Wire-Bus** gelegt werden.
Eine **bewertende** Einordnung des 1-Wire-Busses in die Familie der zuvor erwähnten Bussysteme ist sicherlich nicht ganz einfach und nicht eindeutig durchführbar. Vor allen Dingen ist die Frage, welches System nun das Beste sei, natürlich nie zu beantworten.
Es kommt eben immer auf den jeweiligen Anwendungsfall und auf die Vorlieben des Entwicklers bzw. der Entwicklerin an.
Nachfolgend soll zunächst einmal nur versucht werden, den 1-Wire-Bus gegenüber den anderen Bussystemen in Bezug auf **sechs wichtige Kriterien** abzugrenzen.
Wir lassen hier allerdings einmal außer Acht, dass es sich

- beim SPI- und I^2C-Bus um synchrone (der Takt wird mit übertragen)

  und

- beim 1-Wire- und CAN-Bus um asynchrone (also keine parallel Übertragung eines zusätzlichen Taktes)

Datenübertragungsverfahren handelt.

1. Die Bushardware

beim 1-Wire-Bus ist ähnlich einfach zu realisieren, wie beim SPI- oder beim I^2C-Bus. Während bei den letzt genannten Systemen allerdings immer mindestens zwei Leitungen (zwei digitale I/O-Ports) für die Datenübertragung notwendig sind, kommt der 1-Wire-Bus, wie der

Name schon sagt, mit nur einer Leitung für die Kommunikation aus: ein bidirektionaler digitaler I/O-Port, zusätzlich zur mitgeführten Masse-Leitung.
Beim SPI-Bus ist die Hardware-Bilanz allerdings noch etwas schlechter: denn wenn mehrere SPI-Slave-Bausteine angeschlossen werden sollen, kommt in allgemeinen zur Daten- und zur Taktleitung noch für jeden Chip eine eigene, einzelnen Chip-Select-Leitung (weiterer digitaler I/O-Port) hinzu (Hardware-Adressierung der Teilnehmer).
Da ist der I^2C-Bus schon besser aufgestellt, denn hier bleibt es immer nur bei den zwei digitalen Portleitungen, da die Bausteinadressierung per Software erfolgt.
Der CAN-Bus wird ebenfalls mit nur zwei Datentransferleitungen realisiert, wobei hier jedoch noch ein zusätzlicher externer Bustreiber-Baustein eingesetzt werden muss, um leistungsfähige CAN-Bussysteme aufzubauen.
Der Hardware-mäßige Aufwand für die Datenübertragung erreicht somit beim 1-Wire-Bus das absolute Minimum.

2. Die Kommunikationssoftware,

also das Datenübertragungs-Protokoll, ist, wenn man es selber schreiben muss, beim SPI- und beim 1-Wire-Bus bestechend einfach.
Für eine I^2C-Bus-Realisierung muss man schon „etwas“ mehr nachdenken, dieses ist aber auch von Nicht-Profis gut zu schaffen.
Ein komplettes CAN-Bus-Protokoll dagegen ist nur von den „richtigen“ Profis korrekt zu implementieren, und das auch mit einem Zeitaufwand von einigen Wochen bis hin zu einigen Monaten.
Soll die Software also selber entwickelt werden, so ist dieses beim 1-Wire-Bus sehr schnell erledigt.
Diese ganzen Betrachtungen sind natürlich überflüssig, wenn man entsprechende Chips einsetzt, die bereits das gesamte Kommunikationsprotokoll „in Silizium gegossen“ enthalten oder wenn der ausgesuchte Mikrocontroller bereits eine geeignete ON-Chip-Peripherie-Einheit zu diesem Bussystem besitzt. Solche Lösungen finden sich mittlerweile als Standard für den I^2C- und den CAN-Bus.

3. Die Busausdehnung

Bei den Betrachtungen zur Busausdehnung lassen sich ganz grob zwei Gruppen bilden: SPI- und I^2C-Bus sind vom Grund her so genannte „**Gerätebus-Systeme**“, d.h. sie sind primär dazu konzipiert worden, ICs mit geringen räumlichen Abständen mit einander zu verbinden, also z.B. Bausteine auf einer einzigen Platine oder in einem (kleinen) Geräteschrank.
Mit ausgefeilten „Trickschaltungen“ kann man heut zu Tage die Reichweite dieser beiden Systeme vom 10 cm-Bereich aber auch in den Meter oder 10 m-Bereich erhöhen.
Ganz anders sieht es dagegen beim CAN-Bus aus: er ist zur Überbrückung von Entfernungen im 100 m oder sogar im km-Bereich konzipiert worden (**Feldbussystem**).
Der 1-Wire-Bus nimmt hier eine Zwischenstellung ein: ursprünglich von seinen Vätern als Gerätebus entwickelt, hat man inzwischen erkannt, dass er auch sehr gut zur Überwindung von Entfernungen im Meter oder 100 m-Bereich verwendet werden kann.

Es gibt sogar entsprechende „offizielle“ Application-Notes von MAXIM/DALLAS zur Realisierung von “**Long Line 1-Wire Networks**“.

4. Die erreichbare Busgeschwindigkeit (Datenübertragungsrate)

ist unmittelbar und sehr eng mit der Busausdehnung verknüpft, denn es gilt der **allgemeine Grundsatz**: je schneller die Daten übertragen werden, desto geringer ist die dabei erzielbare Reichweite.
Da die Bitrate beim SPI-, I^2C-und CAN-Bus in weiten Bereichen einstellbar ist, kann hier eine optimale Anpassung an die jeweiligen Anforderungen erzielt werden.
Mit Bitraten bis zu 1 MBit/s besitzt der CAN-Bus hierbei einen recht guten Wert, wobei der I^2C-Bus darunter und der SPI-Bus durchaus (weit) darüber liegt (bis zu 10 MBit/s)
Der 1-Wire-Bus nimmt eine Sonderstellung ein, denn er bietet dem Anwender zunächst nur eine Datenübertragungsrate von 15,4 kBit/s, was aber wiederum für sehr viele (Sensor/Aktor)Anwendungen völlig ausreichend ist. Denn man muss immer bedenken, dass der 1-Wire-Bus primär für Sensor-/Aktor-Bussysteme konzipiert worden ist und nicht für extern schnelle Datentransfers ausgelegt wurde wie das z.B. bei verteilten Datenbanksystemen oder dem Internet der Fall ist.

5. Die Anzahl der anschließbaren Busteilnehmer

Legen wir hier einmal die in Sensor-/Aktor-Anwendungsbereichen sehr oft vorkommende Single-Master / Multi-Slave-Struktur zu Grunde, so stellt sich ganz einfach Frage: wie viele Slave-Stationen kann man im System, unter normalen Bedingungen (also ohne Repeatereinsatz u.ä), anschließen ?
Beim SPI-Bus ist diese Anzahl meistens auf 1 - 3 Slaves beschränkt, da jeder weitere Slave sehr oft einen zusätzlichen, eigenen Chip-Select-Anschluss (digitaler I/O-Port-Pin) benötigt.
Da beim I^2C-Bus die Adressierung der Slaves Software-mäßig, über zugehörige Bausteinadressen, erfolgt, können an die zwei Datenübertragungsleitungen mehrere Slave-Bausteine angeschlossen werden, z.B. 10 – 20 Stück. Die Maximalanzahl der Slaves ergibt sich hierbei durch die Summe aller kapazitiven Belastungen (Kabel, Stecker, Eingangsstufen) an den beiden Masteranschlüssen des Busses.
Beim CAN-Bus ist die Anzahl der physikalischen Stationen pro Segment auf 32 beschränkt, wobei allerdings durch CAN-spezifische Konzepte die Anzahl der austauschbaren Kommunikationsobjekte bis zu 2048 betragen kann.
Beim 1-Wire-Bus liegt die Zahl der handhabbaren Slave-Bausteine bei einigen 10 Stück.
In diesem Zusammenhang ergibt sich allerdings ein kleiner „Wermutstropfen“ für den 1-Wire-Bus: während es für den SPI-, den I^2C- und den CAN-Bus unzähligen Master- und Slave- Bausteine von allen großen Halbleiterherstellern der Welt gibt, ist man beim 1-Wire-Bus auf die Bausteine der „1-Wire-Erfinder“ MAXIM/DALLAS angewiesen, d.h. es gibt weltweit keine anderen Produzenten von 1-Wire-Chips.
Aber die Auswahl von Bausteinen bei MAXIM/DALLAS ist recht groß und man kann sich damit selber sehr einfach seine eigenen 1-Wire-Slaves zusammen bauen.

6. Die Sicherheit der Datenübertragung

In diesem Punkt ist der CAN-Bus der unangefochtene Spitzenreiter: mit einer automatisch realisierten Hamming-Distanz (HD) von 6 erfüllt er selbst die kritischen Anforderungen aus der Automobil-Industrie.
Beim SPI- und beim I^2C-Bus sieht es dagegen „recht düster“ aus: diese beiden Bussystemen bieten zunächst einmal keinen „eingebauten“ Schutz gegenüber auftretenden Störungen bei der Datenübertragung. Hier muss der Anwender selber zusätzliche geeignete Maßnahmen programmieren, wenn ein eine gewisse Sicherheit erreichen will.
Der 1-Wire-Bus nimmt auch hier wieder eine Zwischenstellung ein: er bietet dem Anwender, bei Bedarf eine Datensicherung durch CRC (Cyclic Redundancy Check) an, die ein gewisses Maß an Grundsicherung gegenüber Datenübertragungsfehlern bietet.

Fazit

Fassen wir daher als Fazit in Bezug auf den 1-Wire-Bus zusammen: der 1-Wire-Bus kann eingesetzt werden

- beim Aufbau von Bussystemen mit geringer bis mittlere Ausdehnung im cm-Bereich bis in den m-Bereich,
- wenn sowohl die Bushardware als auch die Datenkommunikationssoftware aufwandsarm selbst erstellt werden sollen,
- wenn die Datenübertragungsrate für einfache Sensor- und Aktor-Anwendungen ausreichend ist,
- wenn die Anzahl der anzuschließenden Bus-Slaves 20 bis 30 nicht übersteigt und
- wenn eine einfache Sicherheit bei der Datenübertragung ausreichend ist.

Und genau solche Anwendungsfälle wie

- die busgestützte digitale Temperaturmessung,
- die Ansteuerung von Relais, LEDs und Summern,
- die Erfassung von Messwerten durch A/D-Wandler,
- der Betrieb von alphanumerischen LC-Displays,
- etc.

sollen nachfolgend näher betrachtet werden.

Aber zuerst einmal schauen wir uns den 1-Wire-Bus selber etwas detaillierter an.

2. Der 1-Wire-Bus

2.1 Eigenschaften

Das gesamte 1-Wire-Konzept ist denkbar einfach aufgebaut und daher auch z.B. mit „kleinen“ 4-Bit-Mikrocontrollern realisierbar.
Man benötigt in erster Line eigentlich nur:

- **Hardware-mäßig**: einen digitalen, bidirektionalen I/O-Port-Pin mit Open-Drain-Eigenschaft und

- **Software-mäßig**: die Möglichkeit, verschieden große Zeitverzögerungen im µs-Bereich zu generieren.

Damit lässt sich bereits ein komplettes 1-Wire-Bussystem aufbauen.

Zusammen gefasst besitzt der 1-Wire-Bus nun folgende **Kerneigenschaften**:

- Aufbau gemäß einer **Single-Master -- Multi-Slave-Struktur** und das bedeutet: in solch einem Bussystem gibt es genau eine einzige Master-Station, die alle anderen Slave-Stationen abfragt bzw. mit geeigneten Informationen versorgt.

- **Serielles, asynchrones Datenübertragungsverfahren**, d.h. die Datenbits werden seriell (nacheinander) übertragen und es ist kein zusätzliches Taktsignal parallel zur Datenübertragung notwendig.

- Es gibt zwei unterschiedliche **Datenübertragungsraten**: einmal die normale Geschwindigkeit von ca. 15,4 kBit/s und den so genannten **„Overdrive“-Modus** mit ca. 125 kBit/s.

- Es können verschiedene **Busstrukturen** aufgebaut werden, [6]:

    - Lineare Struktur,
    - Lineare Struktur mit Stichleitungen (Stubs) von mehr als 3 m Länge,
    - geschaltete Linearstruktur mit Segmentausdehnungen von über 50 m Länge,
    - Sternstruktur.

- Jeder 1-Wire-Bus-Slave besitzt eine eindeutige, beim Herstellungsprozess fest einprogrammierte, unveränderbare **64 Bit große Identifikationskennnummer**.
  Das ist seine eigene **Slave-Adresse**, auch **ROM-Code** oder **ROM-Identifier** genannt.
  Da sich zwei Slaves der gleichen Art (z.B. zwei Temperatursensoren) in 56 dieser 64 Bits unterscheiden, können „im gesamten bekannten Universum“ insgesamt

$$\mathbf{2^{56} = 72.057.594.037.927.936}$$

dieser 1-Wire-Slave-Chips eingesetzt, d.h. von den Master-Stationen eindeutig unterschieden und betrieben werden.
Das ist sicherlich für eine lange Zeit mehr als ausreichend, so dass gar nicht die Notwendigkeit besteht, den Slave-Stationen eigene, änderbare Adressen zuzuordnen. Man arbeitet hier einfach mit den vom MAXIM/DALLAS fest vorgegeben Adressen.
Zum Vergleich einmal folgende Zahlen (Stand: 02:03.2010, 18:00 Uhr):

- Unterschiedliche Slave-Adressen: 72.057.594.037.927.936
- Weltbevölkerung (Menschen): 6.848.184.583
- Verschuldung von Deutschland (€): 1.678.396.403.115

- Eine Vielzahl von Slaves kann auf zwei verschiedene Arten mit **Betriebsspannung** versorgt werden:

  - einmal durch eine **lokale Speisung**, d.h. durch Anschluss einer externen Spannungsversorgung oder alternativ

  - durch eine Versorgung, die direkt über die eigentliche Datenleitung des 1-Wire-Busses erfolgt. D.h. in diesem Falle werden Datenübertragung und Spannungsversorgung über die gleiche Leitungsader realisiert.
    Man spricht hierbei von der „**parasitären Speisung**“ der 1-Wire-Slaves und der Anwender spart eine weitere Leitung bei der Realisierung des Busses ein.
    Allerdings muss man beachten, dass bei einigen Slaves, unter bestimmten Betriebsbedingungen, eine parasitäre Speisung nicht immer möglich ist und man in diesem Fällen eine externe (lokale) Versorgung der Bausteine sicher stellen muss.

- Bei den Slaves selber gibt es bereits ein Vielzahl von **verfügbaren Funktionsbausteinen**, **Abb.2.1.1**:

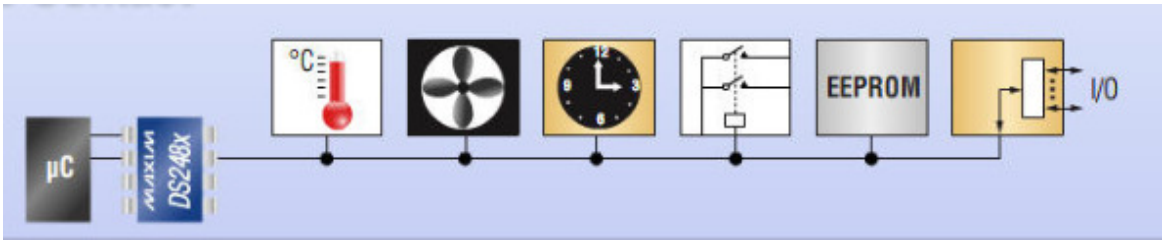


Abb. 2.1.1: Die Vielzahl der 1-Wire-Slave-Funktionen, [2]

Man findet:

- Temperatursensoren,
- A/D-Wandler,

- digitale I/O-Port-Erweiterungen,
- Real Time Clocks (RTC),
- EEPROM-Speicher,
- etc.

Ergänzt wird dieses Spektrum noch durch die so genannten **iButtons**, die dem Anwender z.B. ganze Messwerterfassungssysteme für Temperatur und Luftfeuchtigkeit zur Verfügung stellen (s. Kap.4).

Werfen wir nun als Erstes einen Blick auf den Aufbau von 1-Wire-Master-Stationen.

2.2 Die Realisierung von 1-Wire-Master-Stationen

Zum Aufbau von Masterstationen für einen 1-Wire-Bus (das sind i.a. Mikrocontroller) gibt es grundsätzlich **drei verschiedene Möglichkeiten**, [2]:

1) Der **Mikrocontroller selber** übernimmt direkt die Rolle des Masters. Es ist also auf der Mikrocontroller-Seite lediglich ein freier digitaler bidirektionaler I/O-Port-Pin mit Open-Drain-Eigenschaft notwendig, der dann extern über einen 4,7 kΩ-Pull-Up-Widerstand an die (+5V)Betriebsspannung gelegt wird.
   Das gesamte 1-Wire-Datenübertragungsprotokoll wird dann vom Anwender selbst auf dem Mikrocontroller programmiert: der Mikrocontroller wickelt also zusätzlich zu seiner eigentlichen Applikation noch den 1-Wire-Datentransfer ab.
   Diese ist die einfachste und preiswerteste Realisierung für einen 1-Wire-Master.

2) Es wird ein spezieller externer Chip als 1-Wire-Master eingesetzt (z.B. der DS2482: **integrated 1-Wire Driver**), der an den Mikrocontroller über eine I^2C-Bus-Verbindung angekoppelt ist.
   Dieser Baustein entlastet den Mikrocontroller von der kompletten Abwicklung des 1-Wire-Datentransfers und es werden nur die eigentlichen Nutzdaten zwischen den beiden Bausteinen ausgetauscht, die der 1-Wire Driver dann normgerecht aussendet bzw. bereits empfangen hat.

3) MAXIM/DALLAS stellt dem Anwender auch eine Funktionsbibliothek/Funktionsbeschreibung für einen kompletten 1-Wire-Master zur Verfügung, die dieser dann in einen selbst erstellen, applikationsspezifischen **ASIC-Baustein** einbinden und somit die 1-Wire-Funktionalität mit in diesen Chip integrieren kann.

Wir entscheiden uns in den weiteren Ausführungen für die erste Version, bei der unser 8051er mit einem geeigneten Port-Pin die zusätzlichen Aufgaben eines 1-Wire-Masters mit übernimmt.

2.3 Die 1-Wire-Bus-Hardware

2.3.1 Die Busankopplung

Der 1-Wire-Bus ist ein **serieller asynchroner Bus** und das bedeutet, dass zum Datentransfer kein besonderes separates Taktsignal (über eine eigene, getrennte Leitung) mit übertragen werden muß.
Eine **einzige Datenleitung DQ** ist ausreichend, sofern diese **bidirektional** ausgeführt ist, d.h. über diese eine Leitung werden dann **abwechselnd** Daten in beide Richtungen - vom Master zum Slave und vom Slave zum Master - gesendet (Halbduplex-Verfahren).
Daher stammt auch die Namensgebung „**1-Wire-Bus**“: nur ein Draht ist für die Kommunikation notwendig.
Aber genauer betrachtet das stimmt eigentlich nicht, denn zu allen Stationen muß immer noch die gemeinsame Masse mit gezogen werden, so dass man **mindestens zwei Leitungen** zwischen den Komponenten des Busses verlegen muß !
Wird der Slave dann auch noch lokal gespeist, so ist **noch ein dritte Leitung**, die Versorgungsleitung zum Slave hin, notwendig (s. Kap.2.3.2).
Wir bleiben aber trotzdem nachfolgend bei der Bezeichnung „1-Wire-Bus“.

Die nun benötigte Datenleitung DQ kann im einfachsten Fall aus einem bidirektionalen, digitalen Mikrocontroller-I/O-Port-Pin mit Open-Drain-Eigenschaft bestehen, der (Master-seitig) über einen **Pull-Up-Widerstand** der Größe 4,7 kΩ an die Betriebsspannung (+5 V oder +3 V) gelegt wird, **Abb.2.3.1.1**:

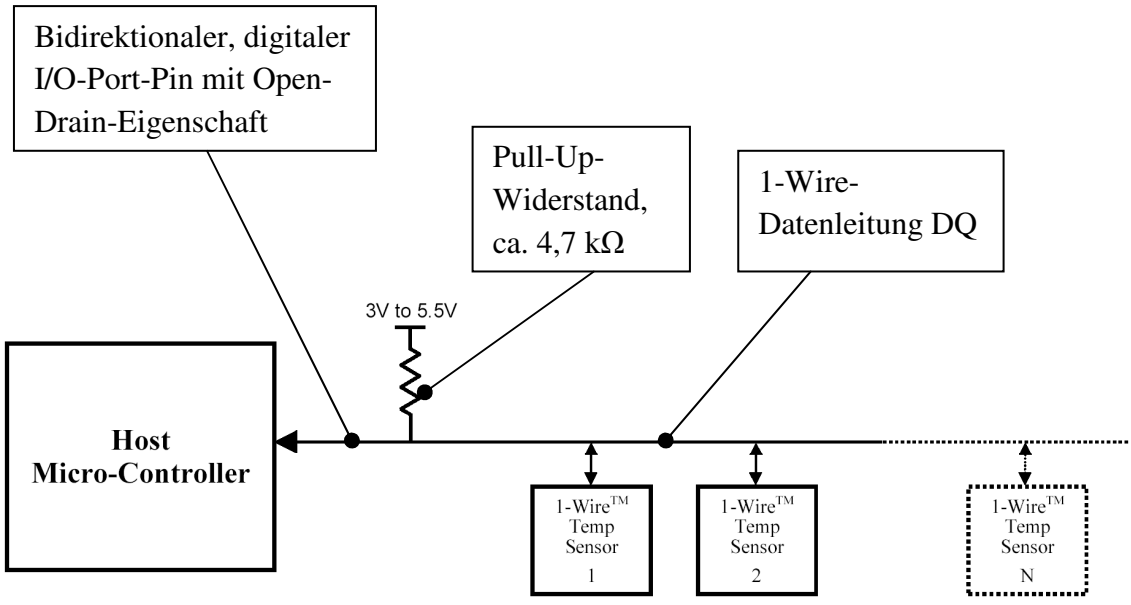


Abb.2.3.1.1: Die einfachste 1-Wire-Busankopplung, [1]

Alle anderen 1-Wire-Slave-Bausteine (hier z.B. digitale Temperatursensoren) werden nun an diese Leitung (und an die gemeinsam Masse, nicht mit eingezeichnet) angeschlossen.

Und fertig ist die einfachste 1-Wire-Busankopplung !

Im Detail sieht das dann im Inneren der Chips wie folgt aus, **Abb.2.3.1.2**:

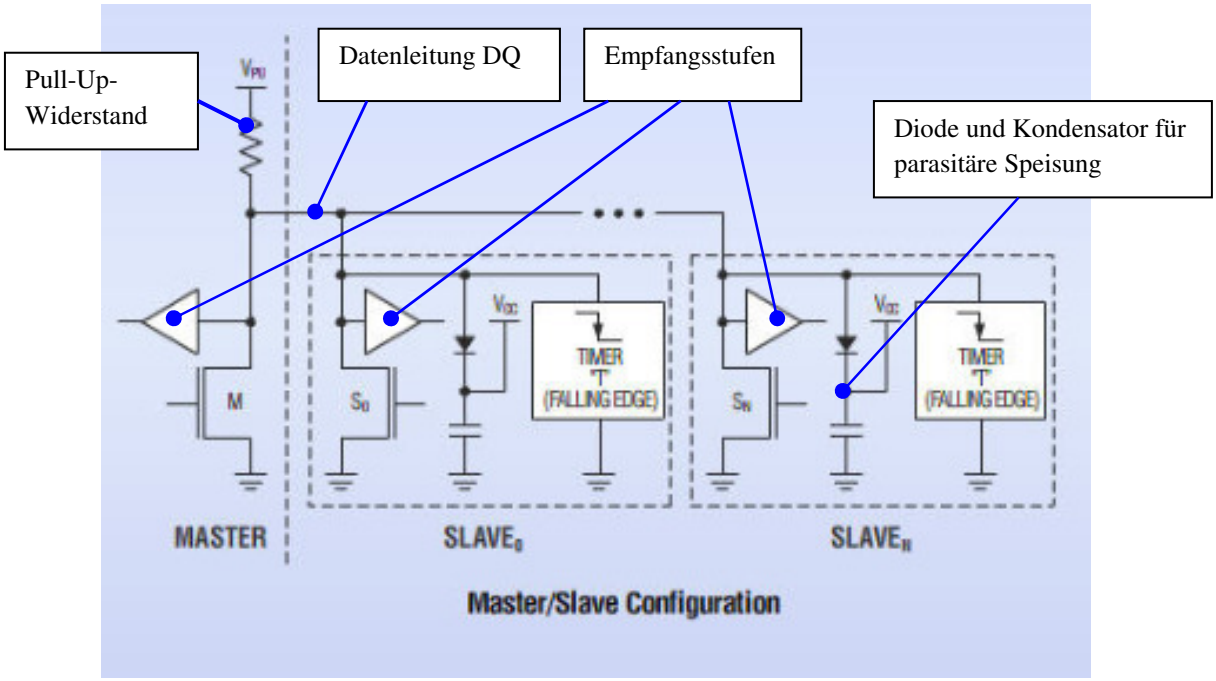


Abb.2.3.1.2: Interna, [2]

Im Ruhezustand liegt die Datenleitung DQ über den Pull-Up-Widerstand auf Betriebsspannungsniveau, also auf High-Pegel.
Wenn der Master nun eine log.´0´ senden will (Low-Pegel), so wird der **Transistor M** durchgesteuert und DQ wird nach Masse gezogen.
Das entsprechende gilt, wenn einer der Slave-Bausteine eine ´0´ ausgeben will: dann wird der entsprechende **Transistor S_0 … S_N** im Slave durchgeschaltet.
Soll nun eine log.´1´ gesendet werden (≡ High-Pegel auf DQ ≡ Ruhelage auf DQ), so bleibt der zugehörige Transistor einfach geschlossen und über den Pull-Up-Widerstand wird DQ wieder auf die Versorgungsspannung gezogen.
Beim Empfang von Bits wird der jeweilige Zustand auf DQ über die **Empfangsstufen** erfasst, aufbereitet und intern weiter geleitet.
Die **Dioden-/Kondensator-Kombination** in den Slave-Chips dient zur Realisierung der parasitären Speisung dieser Bausteine, s. Kap.2.3.2.

An dieser Stelle ist die Beantwortung zweier wichtiger Fragen unumgänglich:

1) **Wie viele Slave-Bausteine** sind eigentlich an einen 1-Wire-Bus anschließbar ?

2) Wie groß kann die **maximale Ausdehnung** eines 1-Wire-Busses sein ?

Die Antworten hierauf sind nicht ganz so einfach anzugeben, denn beiden Größen (Anzahl der Slaves und Busausdehnung) hängen von einigen Faktoren auf der Anwenderseite ab, die schwer in eine allgemeine Aussage bzw. in eine Berechnungsformel zu fassen sind:

- Wie viele Slaves werden parasitär betrieben und wie viele werden lokal gespeist ?
- Welches Buskabel wird verwendet (Leitungsparameter des Kabels ?)
- Welche Steckverbinder bzw. welche Anschlusstechnik wird für die Slaves eingesetzt ?
- Wird mit einem aktiven Pull-Up-Widerstand („Strong Pull-Up“) gearbeitet ?
- Welche Signal-Verzerrungen auf der 1-Wire-Datenleitung DQ sind noch zulässig ?

In der Praxis sieht das dann sehr oft so aus, dass man einige Kombinationen einfach einmal aufbauen und näher untersuchen muss, um so die Einsatzgrenzen heraus zu finden.
Alternativ gibt es zur Beantwortung dieser beiden Fragen auch einige Applikationshinweise von MAXIM/DALLAS, z.B. [6].

Für uns gilt daher bei den nachfolgenden Ausführungen:
Auf den Einsatz von „Spezial-Trickschaltungen“ zur Entwicklung von 1-Wire-Systemen mit größter Ausdehnung und mit einer Vielzahl von Slave-Stationen wird hier zunächst einmal verzichtet, wir betrachten als Erstes nur den einfachen „Standard-1-Wire-Bus“.

2.3.2 Die Speisung von 1-Wire-Komponenten

Zur Speisung von 1-Wire-Slave-Bausteinen gibt es nun zwei (drei) unterschiedliche Möglichkeiten:

- Die lokale, **Vor-Ort-Speisung**, d.h. die Versorgung des Slaves vor Ort durch eine geeignete Spannungsversorgung, z.B. durch ein Netzteil.

- Die **parasitäre Speisung** des Slave-Chips über die Datenübertragungsleitung DQ selber. In diesem Falle ist der Hardware-mäßige Aufwand für den Einsatz solcher Bausteine natürlich minimal und lässt daher kleine, kompakte und preiswerte Busankopplungen zu.
  Das ist somit die zweite Besonderheit beim 1-Wire-Bus und diese soll nachfolgend kurz einmal erklärt werden, **Abb.2.3.2.1**:

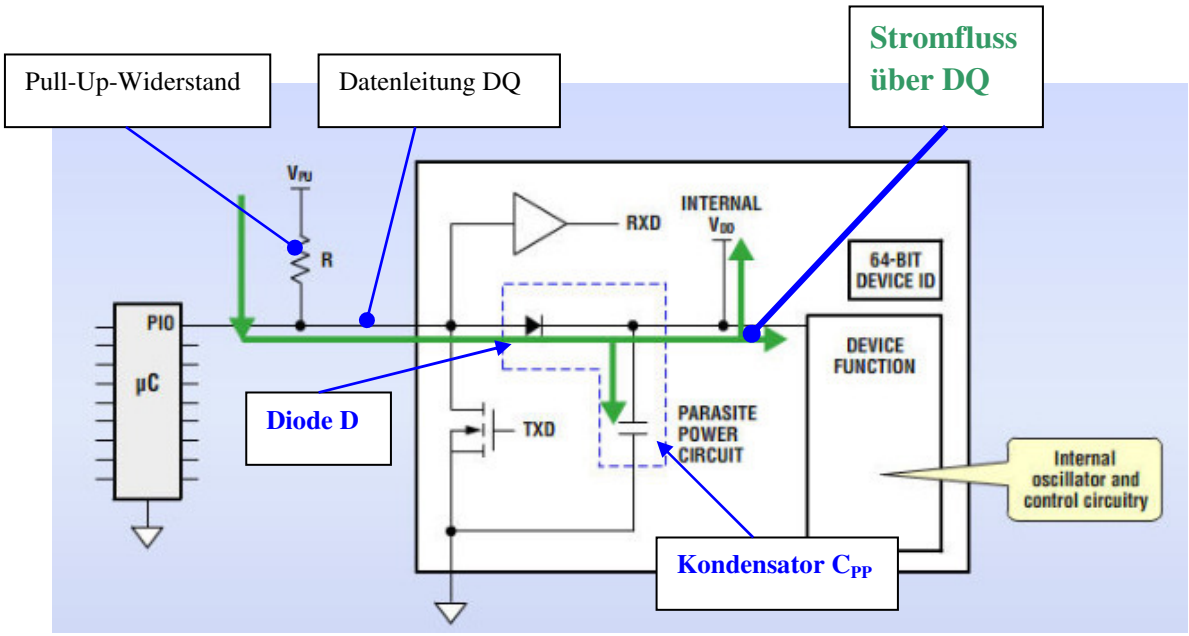


Abb.2.3.2.1: Die parasitäre Speisung von 1-Wire-Slaves über die Datenleitung DQ, [2]

Während des High-Pegels auf DQ (≡ Ruhelage von DQ bzw. Übertragung einer log.´1´) erfolgen über den Pull-Up-Widerstand und die interne Diode D **zwei wesentliche Aktionen** (**grüne Pfeile**):

- Zum einen wird der 1-Wire Slave aktuell mit Strom (Betriebsspannung) versorgt und kann somit seine Aufgaben erfüllen ≡ Verzweigung nach: „**INTERNAL V_{DD}**“

- Zum anderen wird durch den Strom über DQ der interne **Kondensator C_{PP}** aufgeladen (Verzweigung nach „**PARASITE POWER CICUIT**“).

Durch diese Ladung (Spannung) auf C_{PP} kann der 1-Wire-Slave weiter arbeiten, während auf DQ ein Low-Pegel anliegt.

Die Diode D verhindert jetzt (bei Low-Pegel auf DQ), dass sich C_{PP} über diesen Low-Pegel entlädt. Stattdessen gibt der Kondensator nun seine gesamte Ladung „nach rechts“ für den Betrieb des 1-Wire-Slaves ab.

Diese parasitäre Speisung wird von MAXIM/DALLAS zu Recht auch als „**Stolen Power**“-Betrieb bezeichnet und kann natürlich nur unter der Bedingung funktionieren, dass der Slave-Baustein im aktiven Betrieb wirklich „**nur sehr wenig Strom**“ verbraucht und dieser Strom auch „ungestört“ über den Pull-Up-Widerstand fließen kann, d.h. dieser Widerstand darf im praktischen Betrieb eine bestimmte Größe nicht überschreiten.
Und hier sind dann in der Praxis bei einigen 1-Wire-Slaves durchaus Grenzen gesetzt. Betrachten wir dazu einmal beispielhaft kurz den **DS18S20er**, den digitalen Temperatur-Sensor, den wir nachfolgend ja als ersten 1-Wire-Chip praktisch einsetzen wollen:

1) Im normalen, aktiven Betrieb, d.h. es wird nur Kommunikation abgewickelt und gerade keine Temperatur gemessen, verbraucht der Baustein so wenig Strom, dass die parasitäre Speisung über den normalen 4,7 kΩ Pull-Up-Widerstand völlig ausreichend ist.

2) Wird nun aber, auf Befehl des Masters hin, eine Temperaturmessung durchgeführt, so verbraucht der DS18S20er bis zu 1,5 mA Strom.
Dieser Strombedarf kann jedoch nicht mehr über den normalen Pull-Up-Widerstand gedeckt werden. In diesem Fall muss ein so genannter „**Strong Pullup**“ hinzugeschaltet werden, **Abb.2.3.2.2**:

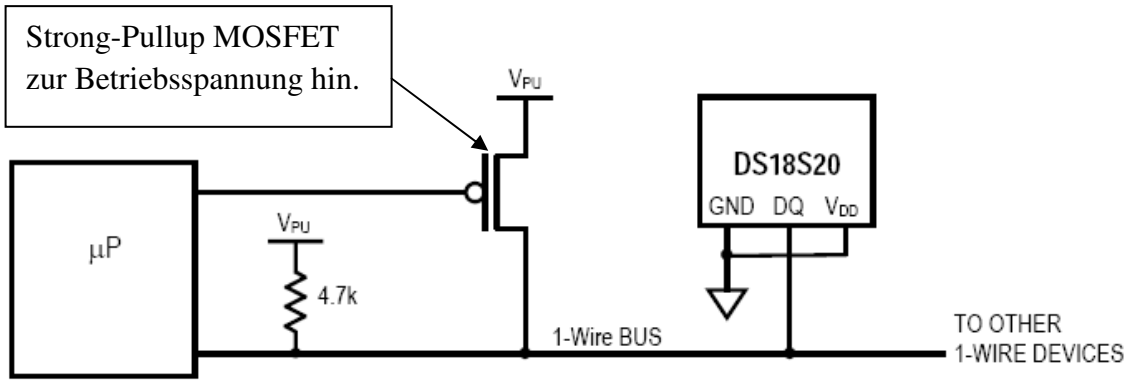


Abb.2.3.2.2: Der Strong-Pullup lässt richtig Strom fließen, [4]

Dieser Strong-Pullup ist ein MOSFET-Transistor, der parallel zum normalen Pull-Up-Widerstand direkt zur Betriebsspannung hin geschaltet ist und über einen weiteren

Port-Pin des Mikrocontrollers aktiviert, d.h. eingeschaltet wird.
Dadurch ist es möglich, dass über die Datenleitung DQ genau für den Fall, dass der DS18S20er eine Temperaturmessung durchführen soll, ein höherer Strom zum Chip fließen kann.
Weitere Informationen zu dieser Art der Stromerhöhung finden sich in den Datenblättern der jeweiligen 1-Wire-Chips oder in [8].

3) Beim DS18S20er gibt es aber noch eine kritische Betriebsbedingung, bei der dann selbst ein Strong-Pullup nicht mehr für den notwendigen Betriebsstrom sorgen kann: wenn der DS18S20er Temperaturen über 100°C messen soll (er kann Temperaturen bis zu +125°C messen), werden die internen Leckströme auf dem Halbleiter-Chip so groß, dass man in diesem Fall **immer eine lokale Speisung** des Baustein, durch eine externe Spannungsversorgung, vorsehen muss.
Für unsere nachfolgenden Anwendungen mit den DS18S20er ergibt sich daraus als Konsequenz, dass wir diesen Chip immer mit einer lokalen Speisung betreiben werden, da wir alle Möglichkeiten, d.h. den gesamten Messbereich des Bausteins, ausnutzen wollen.

Zusammen gefasst halten wir hier also fest:
Beim Betrieb eines 1-Wire-Slaves gibt es drei Arten der Betriebsspannungsversorgung:

- Versorgung über die Datenleitung DQ mit einem normalen Pull-Up-Widerstand.

- Versorgung über die Datenleitung DQ mit Hilfe eines zusätzlichen MOS-FETs ≡ Strong Pullup.

- Versorgung des Chips durch eine externe Spannungsversorgung ≡ lokale Speisung.

Und je nach Anwendungsfall muss man sich immer für eine dieser Möglichkeiten entscheiden.

2.3.3 Die Hardware-Basis des Projekts

Fassen wir nun einmal alle zuvor durchgeführten Hardware-Betrachtungen zusammen, so lässt sich hier festhalten, dass unsere nachfolgend **eingesetzte Hardware-Basis** wie folgt aussieht:

- **1-Wire-Master**: einfache Verwendung eines Port-Pins des 8051er-Mikrocontrollers (des AT89C51CC03ers der Firma Atmel), der bereits die geforderten Eigenschaften besitzt: digitaler, bidirektionaler I/O-Port-Pin mit Open-Drain-Eigenschaft.
  Die notwendige 1-Wire-Kommunikationsoftare für den Mikrocontroller wird selber geschrieben (einfaches Bit-Banging über diesen Port-Pin).

- Einsatz eines normalen **4,7 kΩ-Pull-Up-Widerstandes**, kein Strong-Pullup-MOSFET vorgesehen.

- **Lokale Speisung** aller 1-Wire-Slave-Bausteine, damit man auch all deren Möglichkeiten uneingeschränkt ausnutzen kann (keine parasitäre Speisung).

- Realisierung einer einfachen, nicht segmentierten **Linienstruktur**.

- **Keine** Verwendung von **„Trickschaltungen“** um die Ausdehnung des Busses zur erhöhen bzw. um eine große Anzahl von Slaves anzuschließen.

Hardware-mäßig können also die 1-Wire-Slave-Bausteine **direkt an den Port-Pin des Mikrocontroller angeschlossen werden** oder es kommt alternativ für die ersten Schritte unser speziell entwickeltes PT-1-Wire-Experimentalboard (**PT-1-Wire-ExBo**) zum Einsatz.
Auf dieser Platine (s. Kap.5) werden drei unterschiedliche Arten von 1-Wire-Chips praktisch betrieben:

- bis zu drei digitale Temperatursensoren **DS18S20**,

- ein Mal: vierfach A/D-Wandler **DS2450,**

- ein Mal: achtfach digital I/O-Port-Erweiterung **DS2408** zur Ansteuerung von Relais und LEDs und zur Abfrage von Tastern,

- ein Mal: achtfach digital I/O-Port-Erweiterung **DS2408** zum Betrieb eines alphanumerischen LC-Displays mit 4 Zeilen zu je 20 Spalten.

2.4 Das 1-Wire-Datenübertragungsprotokoll

Beschäftigen wir uns nun mit der Software-mäßigen Seite des 1-Wire-Busses, mit der **Datenübertragungssoftware**.
Sie werden erkennen, dass diese sehr einfach, aber auch sehr effektiv aufgebaut ist und im Prinzip nur aus dem Hin- und Herschalten eines Port-Pins (High-/Low-Pegel) und aus entsprechenden Wartezeiten dazwischen besteht.

2.4.1 Die 1-Wire-Kommunikationspyramide

In einer ersten allgemeinen Übersicht über die benötigten ´C´-Funktionen zum Aufbau einer 1-Wire-Kommunikation (auf der Master-Seite) stellen wir Ihnen als erstes die **1-Wire-Kommunikationspyramide** vor, an Hand derer man sehr gut erkennen kann, wie die gesamte 1-Wire-Software strukturiert ist, **Abb.2.4.1.1**:

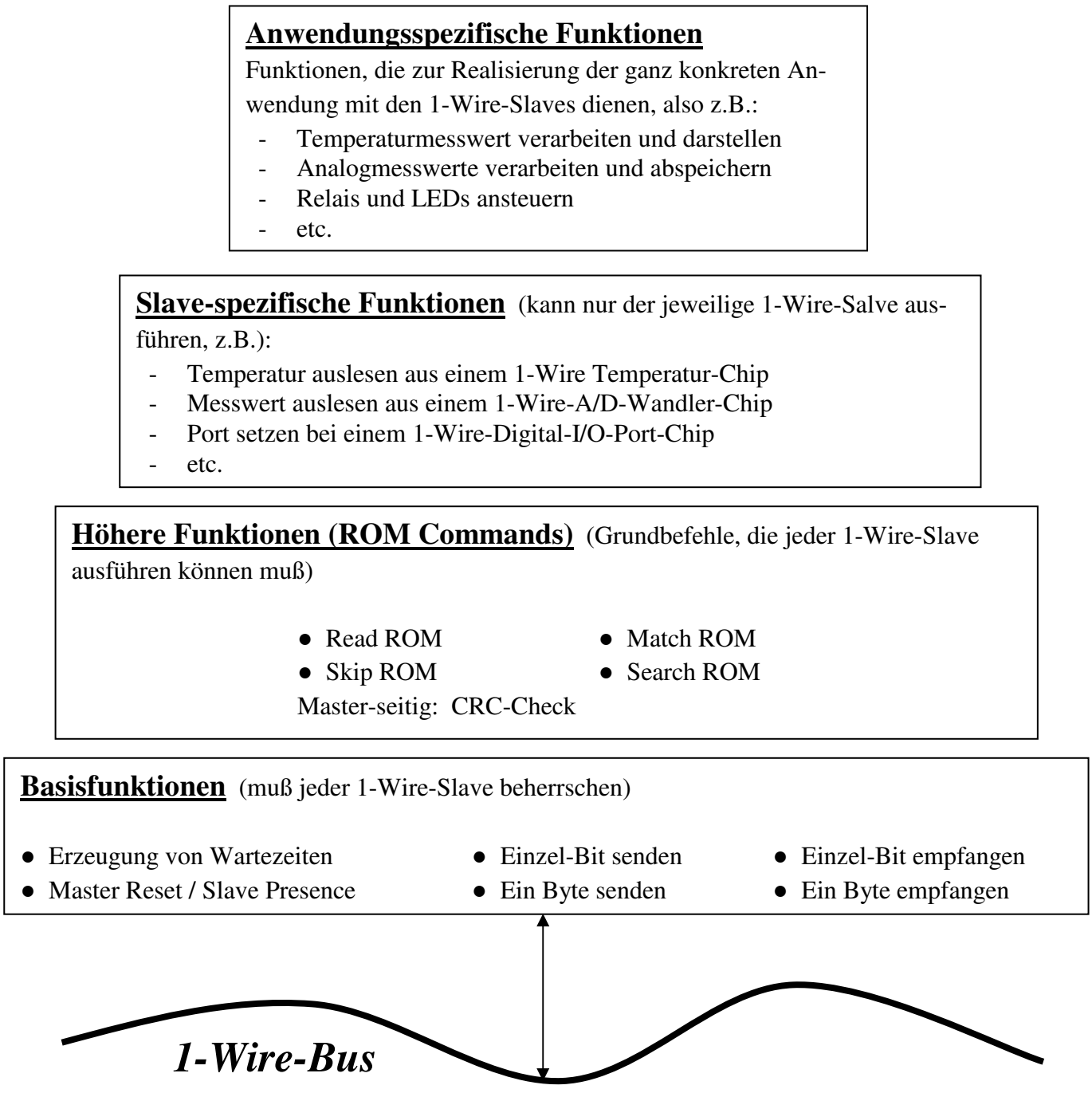


Abb.2.4.1.1: Die 1-Wire-Kommunikationspyramide (Master-seitig)

Eine 1-Wire-Bus-Master-Station muß also für die 1-Wire-Kommunikation mit den 1-Wire-Slaves den folgenden Funktionsumfang beherrschen:

Basisfunktionen

Mit Hilfe dieser Funktionen wird auf der „**ganz untersten Ebene, direkt am 1-Wire-Bus-Draht**“ der Transfer von Bits und Bytes über den 1-Wire-Bus abgewickelt.
Diese und alle nachfolgenden Funktionen können in ganz „normalem“ Standard-´C´ entwickelt werden und sind so recht einfach, mit nur minimalen Änderungen, auf eine Vielzahl anderer Mikrocontroller (Mikrocontroller-Systeme) portierbar (hier sieht man daher einen weiteren Vorteil der Programmiersprache ´C´ gegenüber z.B. Assembler oder Basic, bei denen solch eine Portierung (fast) unmöglich ist).
Die einzige Ausnahme bilden hier die Funktionen zur Erzeugung von Zeitverzögerungen im µs-Bereich: dabei muss man immer die Taktfrequenz des jeweiligen Mikrocontrollers, dessen Takt- und Maschinenzykluszeiten, den internen (Assembler-)Befehlsablauf und letztendlich die Besonderheiten des jeweils eingesetzten ´C´-Compilers genauestens berücksichtigen. Mit anderen Worten: man kommt an dieser Stelle bei jedem System um ein wenig Rechenarbeit bzw. besser: um ein wenig Messen mit dem Oszilloskop, nicht herum.
Diese Realisierung der Basisfunktionen werden wir uns in Kapitel 2.4.3 vornehmen.

Höhere Funktionen

Ist der Einzel-Transfer von ganzen Bytes gelöst worden, so lassen sich als nächstes recht einfach Anweisungen (Befehle) und Daten an die 1-Wire-Slaves senden und die Antworten (Daten) der Slaves im Master empfangen und weiter verarbeiten.
Diese vier „Höheren (Befehls)Funktionen“ bilden den allgemeinen Grundbefehlswortschatz den jeder 1-Wire-Slave verstehen muß: z.B. Abfrage der eindeutigen, individuellen internen 64-Bit-Slave-Adresse, Adressierung des Slaves über diese Adresse, etc.
Diese Befehle werden in den Datenblättern zu den 1-Wire-Komponenten auch als „**Device Selection“-Befehle** bezeichnet.
Die dazu gehörigen Funktionsabläufe in ´C´ erstellen wir in Kapitel 2.4.4.

Die Slave-spezifischen Funktionen

Darunter versteht man nun diejenigen Funktionen (Befehle) die für jeden Slave (individuell) typisch sind, z.B.:

- Bei einem **1-Wire-Temperatur-Chip**: Start der Temperatur-Messung, Auslesen des Temperatur-Messwertes, etc.

- Bei einem **1-Wire-A/D-Wandler**: Auswahl des Eingangskanals, Start der A/D-Wandlung, Auslesen der Wandlungsergebnisse, etc.

- Bei einem **1-Wire-Digital-I/O-Port-Baustein**: Setzen von Ausgangsports, Einlesen von Eingangsports, etc.

Diese Befehle werden in den Datenblättern zu den 1-Wire-Komponenten auch als „**Device Function“-Befehle** bezeichnet.

Die Erstellung der hierzu passenden ´C -Funktionen werden wir nachfolgend als Erstes am Beispiel des digitalen Temperatursensors DS18S20 zeigen.

Anwendungsspezifische Funktionen

Im eigentlichen Anwendungsprogramm verwendet der Master(Programmierer) nun all die zuvor erstellen Funktion, um die gewünschte Applikation mit den 1-Wire-Komponenten zur realisieren, z.B. Aufbau einer Vielstellen-Temperaturmesseinheit mit Anzeige der Temperaturen auf einen LC-Display und Ansteuerung von Relais und LEDs, wenn Temperatur-Grenzwerte über- oder unterschritten werden.

Solch eine „übergeordnete Anwendung“ werden wir zum Abschluss unserer Betrachtungen über 1-Wire-Bus näher vorstellen (s. Kapitel 9).

2.4.2 Grundlegendes

Das 1-Wire Datenübertragungsprotokoll ist denkbar einfach aufgebaut und daher auch z.B. mit „kleinen“ 4-Bit-Mikrocontrollern gut realisierbar.
Ein 1-Wire- Bus ist von der Grundstruktur her immer als **Single-Master -- Multi-Slave-System** aufgebaut und das bedeutet: in solch einem Bussystem gibt es genau eine einzige Master-Station, die alle anderen Slave-Stationen abfragt bzw. mit geeigneten Informationen versorgt.
Grundlage für den Datentransfer sind die so genannten „**Time Slots (Zeit-Schlitze, Zeit-Intervalle)**“, die eine bestimmte, genau festgelegte zeitliche Länge haben und in deren Verlauf dann „etwas passiert“.
Somit **beginnt jeder Datentransfer** auf der Datenleitung DQ mit einem Signal vom Master, hier ganz genau: mit einer fallenden Flanke, d.h. **der Master gibt über DQ einen Low-Pegel aus** (die Ruhelage auf DQ ist ja der High-Pegel, s. Kap.2.3.1).
Diese fallende Flanke ist das so genannte **Synchronisationssignal**, sowohl für den Master selber als auch für alle Slave-Stationen, die an DQ angeschlossen sind.
Mit anderen Worten: bei der fallenden Flanke auf DQ starten alle Stationen ihre internen Timer (Zeitmesser) und ab jetzt beginnt die Zeit eines Time Slots (eines Zeit-Intervalls) in allen Stationen zur laufen, alle Stationen messen ab jetzt die verstrichene Zeit und prüfen, was nun zu welchem Zeitpunkt auf DQ passiert.
Innerhalb eines Time Slots setzt der Master nun nach dem Ablauf bestimmter Zeiten die Leitung DQ wieder auf High oder er belässt sie auf Low. So werden dann ganz einfach die logischen Zustände ´0´ und ´1´ übertragen. Die **Abb.2.4.2.1** verdeutlicht diesen Ablauf:

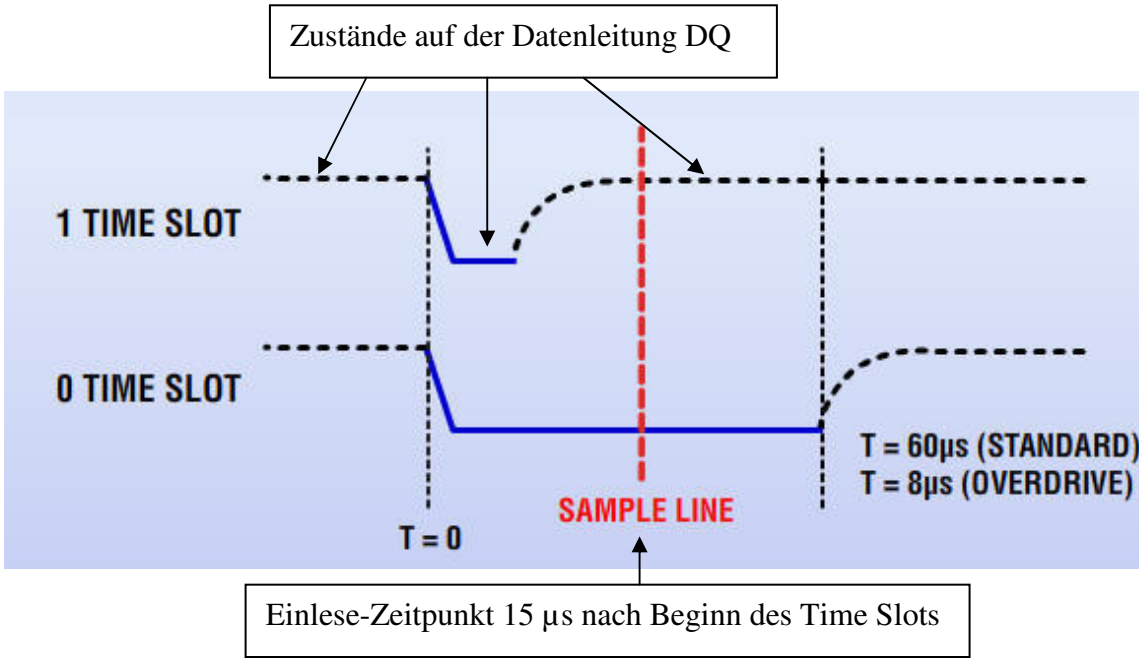


Abb.2.4.2.1: Der 1-Wire Time Slot und die Übertragung der Bits (Master sendet), [2]

Zum Zeitpunkt T = 0 setzt der Master DQ auf Low und der Time Slot in allen angeschlossenen Slave-Stationen beginnt (zu laufen).
Ein **Standard-Time Slot T** ist nun **60 µs** lang (den Overdrive-Zustand, also die Möglichkeit den 1-Wire-Bus mit der zweiten, höheren Datenrate zu betreiben, betrachten wir hier erst einmal nicht).
Nach Ablauf von **15 µs** tasten die Slaves nun die Datenleitung DQ ab: **Sample Line** oder **Sample Point** (diese Zeit von 15 µs ist Slave-intern vom Slave-Hersteller vorgeben und darum berauchen wir uns nicht weiter zu kümmern).
Hat der Master nun aber vor Ablauf der 15 µs DQ wieder auf High gesetzt, so lesen die Slaves eine log.´1´ ein (≡ **1 Time Slot**).
Hält der Master DQ aber bis zum Ende des Time Slots auf Low, so lesen die Slaves eine log.´0´ ein (≡ **0 Time Slot**).
Vereinfacht bedeutet das also:

- Transfer einer log.´1´ ≡ kurzer Low-Impuls auf DQ
- Transfer einer log.´0´ ≡ langer Low-Impuls auf DQ

Das war´s dann schon, so einfach ist der Bit-Transfer beim 1-Wire Bus: **ein Time Slot ist also die Zeit, um ein Bit zu übertragen !**
Und acht hintereinander übertragene Bits ergeben dann ein übertragenes Byte, denn auch bei diesem Bus läuft der Datentransfer immer **Byte-organisiert** ab.

Ganz ähnlich sieht nun Vorgang aus, wenn die Slave-Stationen Bits/Bytes senden und der Master diese empfängt (s. nachfolgendes Kapitel).

Das einzige Problem auf der Software-Seite ist zunächst eigentlich nur die möglichst exakte Erzeugung der Time Slot/Warte-Zeiten im µs-Bereich. Aber, wie Sie noch sehen werden, ist auch das kein wirkliches Problem.

An diese Stelle passt eine kleine Anmerkung zum **Overdrive-Modus**:
Die wesentlich höhere Datentransfergeschwindigkeit beim Overdrive-Modus (125 kBit/s anstatt 15,4 kBit/s im Standard Mode), kommt dadurch zustande, dass man alle (Ablauf)Zeiten erheblich verkürzt, z.B. Zeit des Time Slots von 60 µs auf 8 µs.
Das ist „auf dem Papier“ natürlich einfach machbar und funktioniert auch. Aber in der Praxis ergeben sich hierbei einige Probleme: während die notwendige Zeiten für den Standard-Mode mit langsam getakteten (kleinen) Mikrocontrollern und mit Hochsprachen(´C´)-Programmierung einfach realisierbar sind, können sich bei den (sehr) kleinen Zeiten für den Overdrive-Mode bereits Schwierigkeiten geben, diese zu erzeugen. Hier muß man dann u.U. auf höher getaktete Systeme und/oder auf die in diesem Fall (etwas) schneller ablaufende Assembler-Programmierung zurückgreifen.
Daher stützen wir uns bei den nachfolgenden Betrachtungen zunächst auf den 1-Wire-Standard-Mode ab, der eigentlich immer problemlos implementiert werden kann.

Und eines sollte man sowie so auch immer bedenken: bei der Erfassung von Temperaturen oder bei der Ansteuerung von Relais, Lüftern, etc. ist man sicherlich nicht auf die letzte Mikrosekunde angewiesen, so zeitkritisch ist das Alles bei weitem nicht.

Aber ein Punkt muß auf jeden Fall **IMMER beachtet** werden:
Die vorgegebenen Zeiten bei der Abwicklung eines 1-Wire-Kommunikationsblaufes müssen mehr oder weniger streng eingehalten werden, dürfen also nicht wesentlich verlängert werden und das bedeutet zwingend, da der Mikrocontroller ja der Zeit-Master im System ist:

Während der Kommunikation über den 1-Wire-Bus darf der Mikrocontroller durch KEINE INTERRUPTS gestört werden, alle Interrupts müssen also während der Kommunikation disabled sein !

Erst nach Beendigung der Kommunikation (eines kompletten Kommunikationszyklusses) sind Interrupts wieder zugelassen !

Nachdem wir nun geklärt haben, wie einzelne Bits bzw. ganze Bytes übertragen werden, müssen wir noch ein bisschen **Ablaufsteuerung** betreiben, damit die gesamte Kommunikation auf dem 1-Wire Bus auch sicher durchgeführt werden kann.
Die Firma MAXIM/DALLAS hat dazu ein **3-stufiges Transferschema** entwickelt, wie es eigentlich üblich ist für eine Master-Slave-Kommunikation:

1) Der Master sendet über DQ einen **Reset-Impuls**: alle angeschlossenen Slaves, die sich normaler Weise im Ruhezustand (Stand-By-Modus) befinden, wachen auf und melden sich mit einem so genannten **Presence-Impuls** über DQ zurück.

2) Der Master spricht den gewünschten Slave an (≡ **Device Selection-Befehl**) , denn jeder Slave besitzt intern eine eigene, individuelle, unverwechselbare Stationsnummer, die der Hersteller dem Slave fest und unveränderbar mitgegeben hat (≡ **ROM Code,** s. Kap.2.1).
   Nachdem der gewünschte Slave so selektiert worden ist, kann der Master den eigentlichen Befehl an den Slave übermitteln ≡ **Device Function-Befehl**.

3) Der angesprochene Slave reagiert, d.h. er führt die übermittelte Anweisung aus und antwortet gegebenenfalls mit den vom Master angeforderten Daten, wobei der Master auch in diesem Fall den Datentransfer (zeitlich) steuert.

Die Device Selection-Anweisungen sind Befehle, die alle 1-Wire-Slave verstehen (≡ Grundwortschatz der 1-Wire-Slaves), während die Device Function Befehle die individuellen Funktionen der Slave-Bausteine steuern.

Das ist zunächst einmal Alles, was man über die Kommunikation beim 1-Wire-Bus wissen muss und nachfolgend sehen wir uns ganz konkret an, wie die zugehörige ´C´-Software für unser System aussehen muss.

2.4.3 Die sechs 1-Wire-Basisfunktionen zum Datentransfer

Bevor wir uns nun die sechs Grundfunktionen zum 1-Wire-Datentransfer näher ansehen, müssen hier noch einige Randbedingungen festgelegt werden:

1) Wir betrachten zunächst nur den **Standard-Mode** des 1-Wire-Busses und dazu hatten wir in den vorhergehenden Kapiteln erwähnt, dass ein Time Slot 60 µs lang ist.
   So exakt genau schreibt die MAXIM/DALLAS-Norm diese Zeit aber nicht vor: es ist vielmehr ein recht großer Toleranzbereich von **60 µs … 120 µs** für die Dauer eines Time Slots vorgesehen bzw. zulässig, damit man diese Zeiten auch problemlos mit möglichst „allen Mikrocontrollern der Welt“ erzeugen kann.
   Lediglich die untere Grenze von 60 µs darf nicht unterschritten werden.
   Wir orientieren uns daher bei den nachfolgenden Betrachtungen am oberen Wert von ca. 120 µs und sind damit eigentlich immer auf der sicheren Seite.
   Auch bei anderen Zeiten des 1-Wire-Busses sind recht große Wertebereiche zulässig.

2) Zu beachten ist allerdings, dass der Slave-Sample-Point (beim Einlesen) innerhalb eines Time Slots immer bei ca. 15 µs nach Beginn des Time Slots liegt, s. Abb.2.4.2.1.
   Daher versuchen wir auch auf der von uns programmierten Master-Seite beim Einlesen von Bits diese Zeit einzuhalten

3) Werden Bits gesendet bzw. empfangen, so sollten die Time Slots immer komplett beendet werden, d.h. bei Bedarf werden zusätzlich Wartezeiten eingeführt, um die Rest-Zeit für einen Time Slot aufzufüllen.

4) Zwischen jeweils zwei aufeinander folgenden Time Slots muß eine Wartezeit von mindestens 1 µs eingehalten werden („**recovery time**“).
   Das ist hier bei der Programmierung in der Hochsprache ´C´ aber auch kein Problem und muß daher nicht weiter beachtet werden.

5) Wir entwickeln die 1-Wire-Software in ´C´ für ein **8051er-Zielsystem** (Mikrocontroller: AT89C51CC03 von Atmel im Standard(Takt)-Mode) mit Hilfe der **IDE µC/51** der Firma Wickenhäuser Elektrotechnik, [3].
   Somit müsste die Software **auch auf jedem anderen 8051er** unter diesen Taktbedingungen ablaufen und da die Quellcodes in ´C´ offen gelegt sind, ist eine Portierung auf andere Systeme leicht möglich.

Um nun Bits und Bytes über den 1-Wire-Bus austauschen zu können, muß man auf der Master-Seite **sechs Grundfunktionen** realisieren (egal in welcher Programmiersprache):

1) Funktion zur **Erzeugung von Zeitverzögerungen (Delays)**:
   Hier sind eigentlich zwei Funktionen bzw. zwei Programmlösungen notwendig, da man einerseits (sehr) kleine Zeiten im Bereich von 1 µs bis zu 10 µs und andererseits sehr große Zeiten im Bereich bis zu 700 µs erzeugen muß.

Meistens ist dieser Gesamtzeitbereich von ca. 1 µs bis ca. 700 µs (insbesondere bei der Verwendung von Hochsprachen) nicht mit einer einzigen Funktion abdeckbar, so dass wir hier zwei unterschiedliche Realisierungen vorsehen, die je nach Bedarf verwendet werden.

2) Funktion zur Erzeugung des Master-**Reset-Impulses** und zur Abfrage des **Slave-Presence-Impulses**.

3) Funktion zur **Aussendung eines Bits**, also Aussendung von log.´0´ oder log.´1´ über den 1-Wire-Bus.

4) Funktion zur **Aussendung eines Bytes**, also Aussendung von 8 Bits hintereinander.

5) Funktion zum **Einlesen eines Bits,** also Einlesen von log.´0´ und log.´1´ vom 1-Wire-Bus.

6) Funktion zum **Einlesen eines Bytes**, also Einlesen von 8 Bits hintereinander.

Diese Grundfunktionen haben gemeinsam, dass Sie für die **Kommunikation mit allen 1-Wire-Slave-Stationen** benötigt werden (s. Kap.2.4.1).
Darauf bauen dann später die individuellen, fortgeschrittenen Datentransfer-Funktionen auf, die für jeden 1-Wire-Slave typisch sind, z.B: Auslesen der Temperatur aus einem 1-Wire-Temperatur-Sensor oder Auslesen der 4 Wandlungswerte aus einem 4-kanaligen 1-Wire-A/D-Wandler oder Auslesen der Uhrzeit aus einer 1-Wire-RTC, etc.

1. Funktionen zur Erzeugung von Zeitverzögerungen (Delays)

Die einfachste und effektivste Möglichkeit unter ´C´ eine Zeitverzögerung zu erzeugen ist sicherlich die „Nichts-tuende“ for-Schleife:

```
for(i=0; i<wert; i++);
```

Je nach dem, welcher Wert für ´wert´ ganz konkret eingesetzt wird, wartet das Programm nun eine bestimmte Zeit.
Allerdings ergibt sich hier ein kleines Problem: die Länge der so erzeugten Wartezeit **ist immer systemabhängig**, d.h. abhängig

- vom verwendeten Mikroprozessor,
- von der verwendeten Taktfrequenz und
- vom verwendeten Hochsprachen-Compiler (wie dieser nämlich die for-Schleife in Assembler-Befehle umsetzt).

Mit andere Worten: an diesem Punkt lässt sich keine universelle, allgemeine ´C´-Angabe für die Größe von ´wert´ machen, aus der man dann einfach die Zeit bestimmten könnte.

Das ist also die einzige Stelle in der 1-Wire-Software, an der man selber „scharf“ nachdenken muss.
Bei allen anderen nachfolgend entwickelten Funktionen lässt sich dagegen immer eine allgemeine ´C´-Programmierung ausführen, die i.a. 1:1 auf beliebige andere Mikrocontroller-Systeme portierbar ist, d.h. ohne Änderungen übernommen werden kann.

Um die die reale Zeitverzögerung der gerade angeführten for-Schleife **für sein Zielsystem** zu ermitteln, gibt es zwei Möglichkeiten:

1) **Rechnerisch**: Man schaut sich nach der Compilierung auf der Assembler-Ebene an, wie der übersetzte Code für die for-Schleife aussieht, bestimmt die Maschinenzyklen der einzelnen Befehle und rechnet so die benötige bzw. verstrichene Zeit für diesen gesamten Konstrukt aus.
   Diese Methode ist jedoch recht aufwendig.

2) **Messtechnisch**: Direkt vor Beginn der for-Schleife setzt man im Programm z.B. die DQ-Port-Leitung auf Low-Pegel und direkt nach der for-Schleife wieder auf High-Pegel.
   Mit einem (Digital-Speicher)Oszilloskop, angeschlossen an DQ, kann man nur sehr einfach und sehr exakt die Bereite des Low-Impulses und damit die Länge der Zeitverzögerung in Abhängigkeit vom Wert von ´wert´ ausmessen.
   (Klemmen Sie dazu aber vorher einen vielleicht schon angeschlossenen 1-Wire-Slave von der Datenleitung DQ ab).

Wir haben uns für die einfachere zweite Möglichkeit entschieden und die zugehörige Programmsequenz sieht daher dem entsprechend wie folgt aus:

```
// Festl. des Port-Pins P3.4 des CC03ers als Datenleitung DQ
// Def. als globale Variable
unsigned char bit DQ  @  0xb4;
     .....
     .....
     .....
// „Irgendwo“ in main:

while(1)       // Endlosschleife
{
    DQ = 0;              // DQ auf Low
    for(i=0; i<4; i++); // for-Schleife zur Zeitverzögerung
    DQ = 1;              // DQ auf High
    _wait_ms(10);        // 10 ms warten
}
     .....
```

Als digitalen, bidirektionale Port mit Open-Drain-Eigenschaft zur Realisierung der 1-Wire-Datenleitung DQ haben wir den Port-Pin P3.4 mit der SFR-Bit-Adresse b4h gewählt und diesen ganz am Anfang des Programms als globale Variable definiert.
In main wird dann eine Endlosschleife programmiert, in der DQ abwechselnd auf Low und auf High gesetzt wird, mit der for-Schleife als Zeitverzögerung (hier wurde beispielsweise der Wert ´4´ für das Ende der for-Schleife gewählt, die for-Schleife wird also 4 Mal durchlaufen). Die letzte Anweisung in der Endlosschleife - _wait_ms(10) - ist µC/51 spezifisch und sorgt für eine sehr große Zeitverzögerung von 10 ms zwischen den einzelnen Low-Impulsen auf DQ, damit man solch einen Impuls auf dem Oszilloskop auch gut darstellen kann.
Wird nun das Programm mit unterschiedlichen Werten für das Ende der for-Schleife übersetzt und vom Mikrocontroller abgearbeitet, so erhält man für unser System die folgenden Zeitverzögerungswerte (der Zahlenwert für das Ende der for-Schleife wird immer direkt in die for-Schleife einsetzt und das Programm dann so compiliert !), **Tab.2.4.3.1**:

wert	**Zeitverzögerung in µs**
1	4,3
2	6,5
3	8,7
4	10,8
5	13,0
6	15,2
7	17,4
8	19,5

Tab.2.4.3.1: Die Zeitverzögerungen mit einer einfachen for-Schleife

Durch „scharfes Hinsehen“ lässt sich nun auch eine allgemeine Berechnungsgleichung für die Länge der Zeitverzögerung angeben, die allerdings immer **nur für diese** Systemkonfiguration gilt:

```
zeitverzögerung  =  (wert-1) * 2,2  +  4,2        in µs
```

Man erkennt also, dass man durch „direkten Einbau“ dieser for-Zeile in den Programm-Code sehr kleine Zeitverzögerungen erzeugen kann.
Das ist jetzt unsere erste Möglichkeit, für das 1-Wire-Protokoll, kleine Wartezeiten zu generieren.

Etwas größere Zeitwerte erhält man nun, wenn man diese for-Schleife in eine eigene Funktion einbaut und den Endwert für die Schleife an die Funktion übergibt, also:

```
/*** Realisierung des 1-Wire-Delays: 2. Möglichkeit ***/

void ow_delay(unsigned char delay)
{
    unsigned char i;

    for(i=0; i<delay; i++);

}
```

Hier wird ein Wert zwischen 0 und 255 an die Funktion ´**ow_delay**´ übergeben und dieser Wert bestimmt die Gesamtlaufzeit der for-Schleife.
Ausgetestet werden kann diese Konstruktion wie folgt (hier lassen sich dann interaktiv, nach dem Programmstart, verschiedene Werte für ´delay´ eingeben und so die Zeitverzögerung einfach bestimmen):

```
// „Irgendwo“ in main
……
// Eingabe des delay-Wertes (µC/51 spezifisch)
inputse(s1,4);
delay = atoi(s1);

while(1)            // Endlosschleife
{
    DQ = 0;              // DQ auf Low
    ow_delay(delay);     // Aufruf d. Funktion mit Wert ´delay´
    DQ = 1;              // DQ auf High
    _wait_ms(10);        // 10 ms warten
}
```

Auch hier haben wir mit Hilfe einer kleinen Tabelle die Werte für die Zeitverzögerung in Abhängigkeit von ´delay´ ermittelt, **Tab.2.4.3.2**:

delay	**Zeitverzögerung in µs**
0	19,4
1	26
4	45
5	52
6	58
7	65
10	85
13	105

15	117
16	123
30	215
37	260
60	412
62	424
73	496
75	508

Tab.2.4.3.2: Die Realisierung größerer Zeitverzögerungen (einige Eckwerte)

Durch „scharfes Hinsehen“ lässt sich auch hier eine allgemeine Berechnungsgleichung für die Länge der Wartezeit angeben, die allerdings immer **nur für diese** Systemkonfiguration gilt:

```
zeitverzögerung  =  (delay * 6,55)  +  19,4      in µs
```

Durch Aufruf dieser Funktion lassen sich also im Programm größere Wartezeiten erzeugen. Das ist jetzt unsere zweite Möglichkeit zur Generierung von Zeitverzögerung für das 1-Wire Protokoll.

2. Funktionen zur Erzeugung des Master-Reset-Impulses und zur Abfrage des Slave Presence-Impulses

Bei der 1-Wire-Datenübertragung gilt ja:

Jeder Kommunikationszyklus, jede neue Datenübertragung, wird vom Master initiiert und beginnt IMMER mit einem Master-Reset- und einem Slave-Presence-Impuls !

Durch diesen Reset(Low)-Impuls auf der Leitung DQ werden alle daran angeschlossenen Slaves aufgeweckt, d.h. aus dem Stand-By-Modus in den aktiven Modus geschaltet.
Die so aktivierten Slaves antworten dann auf der DQ-Leitung mit einem Low-Signal und das bedeutet: nach dem Reset-Signal muß der Master DQ wieder auf High-Pegel setzen, damit die Slaves diese Leitung auf Low ziehen können.
Die **Abb.2.4.3.1** zeigt diesen Ablauf am Beispiel des digitalen Temperatur-Sensors DS1820 (dieser Ablauf ist aber identisch für alle anderen 1-Wire-Slaves):

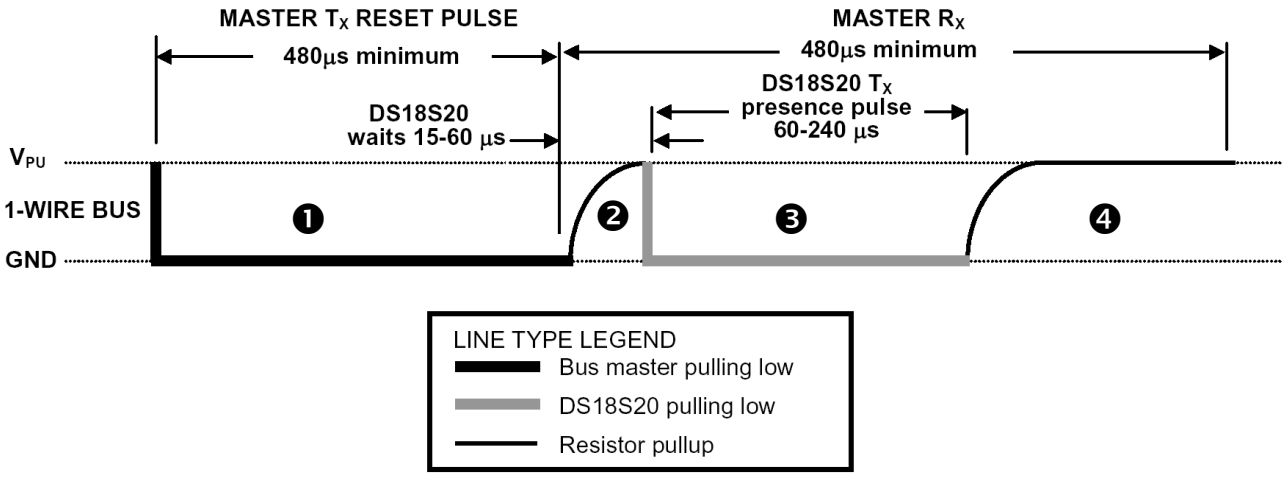


Abb.2.4.3.1: Der Master Reset- und der Slave Presence-Impuls, [4]

Vier Zustände auf DQ sind hier nun wesentlich und müssen per Software realisiert werden:

Zustand ❶:
Der Master erzeugt auf DQ einen Reset-Impuls, d.h. er zieht DQ für **mindestens 480 µs** auf Low-Pegel.
Durch diese fallende Flanke werden die Slaves aktiviert und die internen Zeitgeberstufen für das Zählen der Time Slots werden gestartet ≡ Startzeitpunkt für den Ablauf eines Kommunikationszyklusses (s. Abb.2.3.1.2, Baugruppe „**TIMER "T" (FALLING EDGE)**").

Zustand ❷:
Danach gibt der Master den Bus wieder frei, d.h. der Master gibt einen High-Pegel auf DQ aus (bzw. über den Pull-Up-Widerstand stellt sich ein High-Pegel auf DQ ein).
Dieser High-Pegel sollte zwischen 15 … 60 µs dauern: diese Zeit, beginnend nach der steigenden Flanke auf DQ (≡ Anfang von ❷), warten die Slaves, um ihre Anwesenheit kund zu tun, um danach also ihren Presence-Low-Impuls auf DQ auszugeben.

Zustand ❸:
Die Slaves ziehen nun, quasi als Lebenszeichen, die Leitung DQ auf Low. Das ist dann der (Slave)Presence-Impuls. Nach 60 … 240 µs nehmen die Slaves diesen Pegel wieder zurück, setzen DQ also wieder auf High.

Zustand ❹:
Nach Ablauf von insgesamt mindestens 480 µs (ab der steigenden Flanke auf DQ ≡ Anfang von Zustand ❷) ist dann der gesamte Reset/Presence-Vorgang abgeschlossen.

Für den Master bedeutet das nun:
Während des Abschnitts ❸ muß er DQ abfragen, um festzustellen ob überhaupt ein Slave am Bus angeschlossen ist, ob überhaupt ein Slave seinen Reset-Impuls erkannt hat:

- Liest der Master während des Anschnittes ❸ einen High-Pegel ein, so weis er, dass ihn kein einziger Slave gehört hat: es ist entweder kein einziger funktionierender Slave am Bus angeschlossen oder der Bus (bzw. die Busleitung DQ) ist „irgendwie kaputt“: unterbrochen oder gestört.
  In diesem Fall sollte die Kommunikation abgebrochen und eine kritische Fehlermeldung ausgegeben werden.

- Liest der Master während des Anschnittes ❸ aber dagegen einen Low-Pegel ein, so weis er, dass mindestens ein funktionierender Slave am Bus angeschlossen ist, der ihn gehört hat.
  Hier muß allerdings eines beachtet werden: da ja **jeder** funktionierende Slave einen Low-Presence-Impuls als Antwort auf dem Master-Reset-Impuls erzeugt und der Master immer nur einen (einzigen) Low-Pegel einlesen bzw. erkennen kann, kann der Master nie feststellen, wie viele Slaves eigentlich am Bus angeschlossen sind und sich so zurückmelden: es kann nur ein Salve sein oder es können auch 200 Slaves sein.
  Der Master kann also nur feststellen: mindestens Einer hat mich gehört !

Umgesetzt in ´C´ schreiben wir nun eine geeignete Funktion namens ´**ow_reset**´, die den Master-Reset-Impuls erzeugt und die zurückgibt, ob ein Presence-Impuls als Antwort (von irgendeinem Slave) empfangen wurde:

```
/*** 1-Wire-Reset ***/

unsigned char ow_reset(void)
{
    unsigned char zw;

    DQ = 0;                 // DQ auf Low
    ow_delay(73);           // 496 us warten
    DQ = 1;                 // DQ auf High
    ow_delay(7);            // 65 us warten
    zw = DQ;                // Zustand DQ einlesen
    ow_delay(62);           // 424 us warten = Ende des Reset-
                            // Vorgangs
    return(zw);             // Rueckgabe: 0 = Slave vorhanden,
                            // 1 = kein Slave vorhanden
}
```

Es handelt sich hierbei ganz einfach um die 1:1-Umsetzung des zuvor beschrieben Ablaufes !
Und wie bereits erwähnt: die Wartezeiten müssen nicht 100%ig eingehalten werden, das 1-Wire-Protokoll lässt großzügige Toleranzen zu.

Zu beachten ist, dass nach dem Einlesen des Presence-Zustandes noch ca. 424 µs gewartet werden, bis zum Ende des gesamten Reset/Presence-Vorganges (≡ Zustand ❹).
Nach Abarbeitung der Funktion erhält man somit einen Rückgabewert, der Auskunft darüber gibt, ob mindestens ein Slave am Bus angeschlossen worden ist oder nicht.
Im aufrufenden Programmteil muß diese Information dann entsprechend ausgewertet werden (s. nachfolgend).

Nach dem Ablauf dieser Reset/Presence-Funktion kann der Master nun Befehle an die Slaves senden.
Dazu muß aber erst einmal erklärt werden, wie eigentlich einzelne Bits/Bytes über den 1-Wire-Bus gesendet bzw. empfangen werden. Und damit beschäftigen wir uns jetzt:

3. Funktion zum Aussenden eines Bits (Master schreibt ein Bit auf den 1-Wire-Bus)

Die **Abb.2.4.3.2** zeigt das hierzu notwendige Bus-Timing:

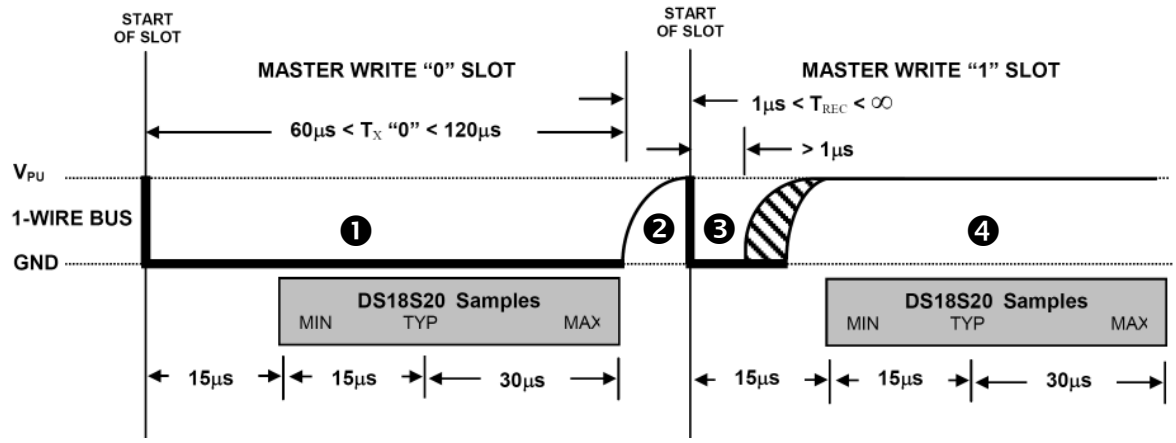


Abb.2.4.3.2: Das Schreiben von Bits, [4]

Hier kommt nun dem zuvor bereits erwähnten Time Slot eine große Bedeutung zu:

Master schreibt eine logische ´0´ (linke Bild-Hälfte von Abb.2.4.3.2)

Zustand ❶:
Der Master setzt DQ auf log.´0´ ≡ Start des Time Slots = Start der Zeitsynchronisation aller Zeitzähler in allen Slaves und im Master.
DQ wird nun bis zum Ende des Time Slots auf log´0´ gehalten, also 60 … 120 µs lang.
(In der Abb.2.4.3.2 ist der Time Slot 60 µs lang, aber die 1-Wire-Norm lässt ja im ungünstigsten Fall eine Länge von bis zu 120 µs für einen Time Slot zu).

Zustand ❷:
Danach setzt der Master DQ wieder auf log.´1´ und wartet mindestens 1 µs bevor das nächste Bit geschrieben wird bzw. bevor der nächste Time Slot angefangen wird.

Die Slaves tasten nun (frühestens) 15 µs nach Beginn des Time Slots (also nach Beginn der fallenden Flanke von Zustand ❶) die Leitung DQ ab (≡ **DS18S20 Samples - MIN**) und finden dort eine log.´0´, die sie dann einlesen.
Der maximale (ungünstigste) Sample-Zeitpunkt bei den Slave liegt bei 60 µs nach Beginn des Time Slots (≡ **DS18S20 Samples - MAX**), bis dahin sollte sich also der Pegel auf DQ nicht verändern.

Ganz ähnlich funktioniert nun:

Master schreibt eine logische ´1´ (rechte Bild-Hälfte von Abb.2.4.3.2)

Zustand ❸:
Der Master setzt DQ auf log.´0´ ≡ Start des Time Slots.
Nun wird DQ aber **sofort** (frühestens nach Ablauf von 1 µs, spätestens aber nach 15 µs) wieder auf log.´1´ gesetzt.

Zustand ❹:
Dieser Pegel bleibt bis zum Ende des Time Slots, also für max. 120 µs, auf DQ erhalten (die abschließende 1 µs Mindest-Recovery Time zwischen zwei aufeinander folgenden Time Slots ist in dieser Zeit dann schon enthalten).

Die Slaves tasten nun wiederum (frühestens) 15 µs nach Beginn des Time Slots (also nach Beginn der fallenden Flanke von Zustand ❸) die Leitung DQ ab (≡ **DS18S20 Samples - MIN**) und finden dort eine log.´1´, die sie dann einlesen.

Zusammen gefasst lässt sich also sagen:

- Master überträgt eine log.´0´ ≡ langer Low-Impuls im Time Slot auf DQ
- Master überträgt eine log.´1´ ≡ (sehr) kurzer Low-Impuls im Time Slot auf DQ

Die somit recht einfache Umsetzung nach ´C´ sieht daher wie folgt aus:

```
/*** Ausgabe eines Bits über den 1-Wire-Bus ***/

void ow_wr_bit(unsigned char bitwert)
{
  DQ = 0;                        // Start Time Slot: DQ auf Low
  if(bitwert) DQ = 1;            // Bei log.´1´: sofort wieder
```

```
                                  // auf High = nur kurzer
                                  // Low-Impuls
  ow_delay(13);                   // ca. 105 us warten bis
                                  // Ende des Time Slots
  DQ = 1;                         // DQ wieder auf High
}
```

Wir schreiben also eine eigene Funktion namens ´**ow_wr_bit**´, an die der Bitzustand (0 oder 1) übergeben wird und die diesen dann aussendet.

4. Funktion zum Aussenden eines Bytes

Ein Byte wird nun ganz einfach dadurch verschickt, dass acht mal hinter einander ein Bit gesendet wird, wobei gemäß den 1-Wire-Festlegungen das niederwertigste Bit des Bytes **(das LSB) zuerst gesendet wird**.

Die passende Funktion namens ´**ow_wr_byte**´ sieht dann so aus:

```
/*** Ausgabe eines Bytes über den 1-Wire-Bus ***/

void ow_wr_byte(unsigned char dat)
{
  unsigned char i;
  unsigned char maske = 1;

  // 8 Bits nacheinander raus schieben, LSB zuerst
  for (i=0; i<8; i++)
  {
    if (dat & maske) ow_wr_bit(1);     // log.´1´ senden
    else ow_wr_bit(0);                 // log.´0´ senden
    maske = maske * 2;                 // nächstes Bit
                                       // selektieren
  }
}
```

Der **wesentliche Trick** in dieser Funktion, an die das zu sendende Byte übergeben wird, besteht darin, die Bits des Bytes einzeln zu selektieren und dann auszusenden: das machen wir durch geschickte Verknüpfung mit einer Maske, bei der eine ´1´ durchgeschoben und so jedes Bit einzeln ausgewählt wird.

5. Funktion zum Einlesen eines Bits (Master liest ein Bit vom 1-Wire-Bus ein)

Die **Abb.2.4.3.3** zeigt das hierzu notwendige Bus-Timing:

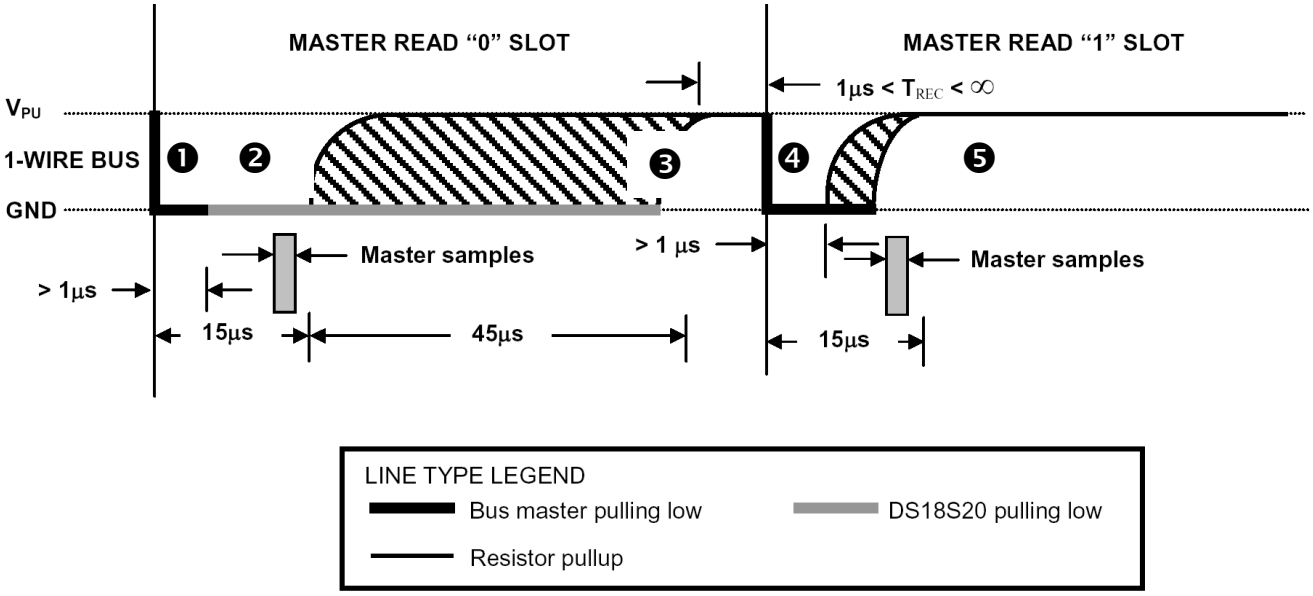


Abb.2.4.3.3: Das Einlesen von Bits, [4]

Hier kommt nun ebenfalls den Time Slots eine große Bedeutung zu:

Master liest eine logische ´0´ ein (linke Bild-Hälfte von Abb.2.4.3.3)

Zustand ❶:
Der Master setzt dazu DQ auf log.´0´ ≡ Start des Time Slots = Start der Zeitsynchronisation aller Zeitzähler in allen Slaves und im Master.
Möglichst schnell, nach 1 µs bis maximal 15 µs, setzt der Master DQ wieder auf log.´1´ bzw. der Master gibt den Bus frei und über den Pull-Up-Widerstand stellt sich auf DQ der High-Pegel ein (in Abb.2.4.3.3 nicht direkt eingezeichnet, da ja sofort der Slave reagiert).
Nach dieser steigenden Flanke auf DQ reagiert der Slave:

Zustand ❷:
Der Slave zieht nun seinerseits DQ auf Low und hält diesen Pegel auf DQ bis zum Ende des Time Slots, also 60 .. 120 µs lang (in der Abb.2.4.3.3 ist der Time Slot 60 µs lang, aber die 1-Wire-Norm lässt ja im ungünstigsten Fall eine Länge von bis zu 120 µs für einen Time Slot zu).
Spätestens 15 µs nach Beginn des Time Slots (nach der fallenden Flanke am Anfang von Zustand ❶) liest nun der Master den Zustand von DQ ein (≡ **Master samples**) und erkennt auf DQ einen Low-Pegel, also eine log.´0´.

Zustand ❸:
Nachdem der Master nun den Zustand auf DQ eingelesen hat, muß er noch bis zu Ende des Time Slots warten, bevor er etwas Neues initiiert: also insgesamt 45 … 105 µs warten, je nach angenommener Länge des Time Slots.
Die Bus-Recovery Time von 1 µs zwischen zwei aufeinander folgenden Time Slots lässt sich hier noch leicht „mit integrieren".

Ganz ähnlich sieht es nun aus bei:

Master liest eine logische ´1´ ein (rechte Bild-Hälfte von Abb.2.4.3.3)

Zustand ❹:
Der Master setzt dazu DQ auf log.´0´ ≡ Start des Time Slots = Start der Zeitsynchronisation aller Zeitzähler in allen Slaves und im Master.
Möglichst schnell, nach 1 µs bis maximal 15 µs, setzt der Master DQ wieder auf log.´1.
Nach dieser steigenden Flanke auf DQ reagiert der Slave:

Zustand ❺:
Der Slave gibt nun seinerseits auf DQ einen High-Pegel aus und hält diesen Pegel auf DQ bis zum Ende des Time Slots, also 60 .. 120 µs lang (ganz genau gesagt: der Slave gibt DQ frei und über den Pull-Up-Widerstand stellt sich auf DQ der High-Pegel ein).
Spätestens 15 µs nach Beginn des Time Slots (nach der fallenden Flanke am Anfang von Zustand ❹) liest nun der Master den Zustand von DQ ein (≡ **Master samples**) und erkennt auf DQ einen High-Pegel, also eine log.´1´.

Nachdem der Master nun den Zustand auf DQ eingelesen hat, muß er noch bis zu Ende des Time Slots warten, also insgesamt 45 … 105 µs, je nach angenommener Länge des Time Slots.
Die Bus-Recovery Time von 1 µs zwischen zwei aufeinander folgenden Time Slots ist darin schon enthalten.

Auch hierbei ist die ´C´-Implementierung recht einfach durchzuführen:

```
/*** Einlesen eines Bits über den 1-Wire-Bus ***/

unsigned char ow_rd_bit(void)
{
  unsigned char zw;
  unsigned char i;

  DQ = 0;                       // DQ auf Low
  DQ = 1;                       // DQ sofort wieder High
```

```
  for(i=0; i<6; i++);          // 15 us warten
  zw = DQ;                     // DQ einlesen u. speichern
  ow_delay(13);                // noch 105 us warten
                               // bis Ende Time Slot
  return(zw);                  // Rückgabe von DQ
}
```

Die Funktion ´**ow_rd_bit**´ ist eine Funktion mit Rückgabe-Wert, die entweder 0 oder 1 zurück gibt, je nach eingelesenem Zustand auf DQ.
Um die (sehr) kleine Zeitverzögerung von 15 µs zu erzeugen kommt hier die erste Möglichkeit zur Wartezeitgenerierung zum Einsatz, s. Tab.2.4.3.1.

6. Funktion zum Einlesen eines Bytes

Ein Byte wird nun ganz einfach dadurch eingelesen, dass acht mal hinter einander ein Bit eingelesen wird, wobei gemäß den 1-Wire-Festlegungen das niederwertigste Bit des Bytes **(das LSB) zuerst gelesen wird**.

Die passende Funktion namens ´**ow_rd_byte**´ sieht dann so aus:

```
/*** Einlesen eines Bytes über den 1-Wire-Bus ***/

unsigned char ow_rd_byte(void)
{
  unsigned char i;
  unsigned char wert = 0;

  // 8 Bits hintereinander einlesen, LSB zuerst
  for(i=0; i<8; i++)
  {
    if (ow_rd_bit()) wert |=0x01 << i;
  }

  return(wert);        // Komplettes Byte zurückgeben
}
```

In der Funktion ´**ow_rd_byte**´, mit Rückgabe-Wert, werden die einzelnen Bits von der Datenleitung DQ eingelesen und mit Hilfe einer „geschickten“ Schiebe- und Verknüpfungstechnik zu einem Byte zusammengebaut.

Damit haben wir nun die sechs Grundfunktionen zur Realisierung von 1-Wire Kommunikationszyklen erstellt und können uns nachfolgend ansehen, welche **fortgeschrittenen Funktionen** es für den 1-Wire-Bus gibt bzw. wie nun bestimmte Datenübertragungsabläufe abgewickelt werden.

2.4.4 Die Höheren Funktionen

Wie in Abb.2.4.1.1 dargestellt, gibt es vier höhere Funktionen, die so genannten **ROM Commands,** die jeder 1-Wire-Slave verstehen muss.
Nachfolgend werden wir diese etwas näher erläutern und ihren Gebrauch in den entsprechenden ´C´-Funktionen darstellen.

Zunächst aber einige **grundsätzliche Betrachtungen** zum Ablauf der Kommunikation über den 1-Wire-Bus:

1) Nach den Einschalten der Betriebsspannung sind alle Slaves im Stand-By(Schlaf)-Modus und warten darauf, dass sie vom Master angesprochen werden, d.h. die Kommunikation auf dem Bus beginnt immer mit einer Master-Aktion.

2) Durch den vom Master gesendeten Reset-Impuls wachen alle Slaves (Aktiv-Zustand) auf und melden sich mit ihrem Presence-Impuls zurück.
   Ab jetzt sind die Slaves permanent im Empfangs-Modus und sie wissen, dass die weitere Kommunikation immer gleich abläuft: als nächstes sendet der Master etwas und darauf hin antworten bzw. reagieren die Slaves.

3) Nun kann der Master also gezielt eine erste Anweisung an einen Slave senden (ein ROM-Command oder eine Slave-spezifische Anweisung): er adressiert den Slave mit seiner individuellen Adresse und übermittelt einen Befehle mit eventuell folgenden Daten.
   Die nicht adressierten Slaves gehen wieder in den Stand-By-Modus.

4) Der Slave empfängt seine Adresse und den Befehl, ev. mit nachfolgenden Daten.
   Der Slave führt die Anweisung aus, verarbeitet die gesendeten Daten bzw. kann seinerseits nun Daten zurück senden, wenn der Master welche angefordert hat.

5) Danach ist der Kommunikationszyklus beendet, der aktivierte Slave geht ebenfalls wieder in den Stand-By-Modus und ein neuer Kommunikationsablauf beginnt bei Punkt 2.

Und nun die Details:

Als Erstes definieren wir im ´C´-Programm ein globales, **8 Byte großes Array** namens ´**rom_c**´ aus unsigned char-Daten, da beim nachfolgend realisierten **allgemeinen Betrieb** mit 1-Wire-Slave-Bausteien ein genau 8 Byte großes Feld zur Aufnahme des 64 Bit großen Slave-individuellen **ROM-Codes** (**ROM-Identifiers**) benötigt wird.

```
unsigned char rom_c[8];
```

Die vier **Höheren Funktionen**, bzw. die **Grundbefehle**, bzw. die so genannten **ROM Commands**, die jeder 1-Wire-Slave verstehen muß, sind, [4]:

- Read ROM
- Match ROM
- Skip ROM
- Search ROM

Read ROM (Befehls-Code: 33h)

Wie zuvor bereits erwähnt, besitzt jeder 1-Wire-Bus-Slave-Chip eine eigene individuelle, 64 Bit lange Adresse (**ROM Identifier oder ROM Code**), die vom Hersteller beim Herstellungsprozess fest und unveränderbar in den Chip „eingebrannt" wird. Über diese Adresse ist jeder 1-Wire-Slave „auf der Welt" zweifelsfrei adressierbar (s. Kap. 2.1).
Aber: der Master im System muß diese Adresse ja erst einmal kennen, also zuerst einmal aus dem Chip selber auslesen. (denn die ROM-Codes sind LEIDER NICHT außen auf den Chips aufgedruckt).
Dazu dient der Befehl ´**Read ROM**´: man schließt EINEN EINZIGEN Slave an den Bus an, sendet diese Anweisung an den Slave und dieser antwortet dann mit seiner internen Adresse, die der Master dann abspeichert.
(Selbstverständlich funktioniert dann nur dann, wenn wirklich auch nur ein Slave am Bus angeschlossen ist, denn sonst würden sich durch diesen Befehl alle Slaves angesprochen fühlen und mit ihrer Adresse antworten. Der Bus würde also zusammenbrechen).
Damit sieht dieser Kommunikationszyklus in Form der Funktion ´**ow_rd_rom**´ wie folgt aus:

```
/*** Lesen des 64-Bit-ROM-Identifiers ***/

void ow_rd_rom(void)
{
  unsigned char i;

  // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
  if(ow_reset())
  {
    printf("\n\n\n\n  Fehler beim Lesen des ROM-Identifiers:
    KEIN Presence vom Slave !");
    printf("\n\n  Reset druecken ... ");
    while(1);
  }

  // Abfrage-Befehl senden: "READ ROM" = 0x33
  ow_wr_byte(0x33);

  // Antwort einlesen: 8 Byte = 64 Bit Identifier ins
  // globale Array rom_c[.]
  for (i=0; i<8; i++)
```

```
  {
    rom_c[i] = ow_rd_byte();
  }
}
```

Der Master beginnt den Datenaustausch mit dem Aussenden eines Reset-Impuls (≡ Aufruf der Funktion ´**ow_reset**´).
Erhält er darauf hin keinen Presence-Impuls, so erkennt er, dass gar kein Slave angeschlossen ist: es wird eine Fehlermeldung ausgegeben und das Programm läuft in einer Endlos(warte)schleife.
Erst durch Druck auf den Reset-Knopf arbeitet der Mikrocontroller erneut von vorne.
Erhält der Master dagegen einen Presence-Impuls, so geht er davon aus, dass ein Slave angeschlossen ist und sendet diesem nun den Befehls-Code 33h als Aufforderung an den Slave, ihm nachfolgend seinen ROM-Code zurückzusenden (Transfer von 33h mittels der Funktion ´**ow_wr_byte**´).
Der Master liest also nach Aussendung des Befehls acht Bytes mit Hilfe der Funktion ´**ow_rd_byte**´ aus dem Slave ein (8 * 8 = 64 Bit Identifier).
Diese Werte werden im globalen Array ´rom_c´ abspeichert und stehen damit im Hauptprogramm zur weiteren Auswertung zur Verfügung.
Danach ist die Funktion ´ow_rd_rom´ beendet.

Match ROM (Befehls-Code: 55h)

Mit dieser Anweisung kann der Master einen einzelnen Slave ganz gezielt über dessen Adresse (≡ 64 Bit ROM Code) ansprechen, d.h.: der Master sendet zuerst das Match ROM-Befehlsbyte (55h) und danach sendet er die entsprechende 64 Bit-Adresse des Slaves, also 8 weitere Bytes.
Damit ist (einzig und allein) der gewünschte Slave adressiert und aktiviert, mit anderen Worten: auf den nächsten Befehl, den der Master sendet, reagiert nur dieser zuvor adressierte Slave, die anderen Slaves bleiben inaktiv.
Hat der Slave den Befehl vom Master dann ausgeführt, ist dieser Kommunikationszyklus abgeschlossen und der Master kann dann z.B. einen anderen Slave mit Match ROM adressieren und mit diesem arbeiten.
In ´C´ würde das so aussehen:

```
// Match ROM-Befehl aussenden
ow_wr_byte(0x55);

// Sofort danach den ROM Ident. des Slave senden (8 Bytes)
for (i=0; i<8; i++) ow_wr_byte( …… );   // ausfüllen

// Nun Anweisung für den adressierten Slave senden
ow_wr_byte( …… );          // ausfüllen

// Slave reagiert entsprechend, z.B. sendet Daten zurück
```

```
// Master empfängt diese Daten
……

// Kommunikationszyklus beendet, Master kann
// nächsten Slave adressieren
```

Diese Match ROM-Anweisung und die nachfolgende Übertragung des Slave ROM-Identifiers wird also in die jeweilige Slave-spezifische bzw. Anwendungs-spezifische Funktion mit eingebaut (s. Abb.2.4.1.1).

Skip ROM (Befehls-Code: cch)

Mit dieser Anweisung kann der Master, im Gegensatz zu Match ROM, alle Slaves gleichzeitig ansprechen, d.h.: ohne jeden Slave einzeln über seinen 64-Bit ROM Code zu adressieren, reagieren alle Slaves simultan auf den Befehl der nach Skip ROM vom Master gesendet wird (Skip ROM ≡ Überspringe die ROM-Adresse).
Skip ROM hat also die Funktion einer „**Rundspruch-Anweisung**“ für alle angeschlossenen Slaves.

Beispiel:
Temperatur-Messung mit dem DS18S20 (s. auch Kap. 6): Sie haben in Ihrem System 10 solcher Temperatur-Sensoren eingebaut. Für den DS18S20 gibt es nun u.a. zwei Anweisungen zur Temperatur-Messung:

- Befehl zum Start der Temperaturmessung (Befehls-Code: 44h)
- Befehl zum Auslesen der gemessenen Temperatur (Befehls-Code: beh)

Um nun bei allen Sensoren **gleichzeitig (!)** die Temperatur zu messen, kann man an alle Slaves mit Skip ROM simultan den Start-Befehl zur Temperaturmessung übermitteln und danach von jedem Slave einzeln, nacheinander, mit Match ROM, die erfassten Temperaturwerte auslesen:
In ´C´ sähe das dann so aus:

```
// Skip ROM-Befehl aussenden: alle Slaves reagieren dann
// auf den danach folgenden Befehl:
ow_wr_byte(0xcc);

// Auf den nun gesendeten Befehl reagieren alle Slaves
// d.h. alle Slaves beginnen jetzt gleichzeitig mit der
// Temperaturmessung:
ow_wr_byte(0x44);

// Nun muß der Master bei den Slaves einzeln (!) die
// erfassten Temperaturmesswerte auslesen.
// Der Master arbeitet jetzt also mit Match ROM, wie
// zuvor beschrieben
```

.....

Spezialfall: Nur ein einziger Slave am Bus

Ist nur ein (!) einziger Slave am Bus angeschlossen, z.B nur ein einziger Temperatursensor und sonst keine andere Station, so kann man die Skip ROM-Anweisung für eine „**trickreiche“, zeitsparende Programmierung** verwenden:
Da nur ein Slave vorhanden ist, braucht man diesen ja gar nicht über seinen individuellen ROM-Code zu adressieren, es ja nur diese Station da.
Also arbeitet man hier nur mit dem Skip ROM-Befehl: der Slave reagiert somit immer auf die Anweisungen vom Master.
Man erspart sich dadurch die regelmäßige Übertragung der 64 Bit ROM-Adresse für den Slave und das ist dann schon eine erheblich Zeitersparnis.
Aber das geht eben nur, wenn nur ein einziger Slave am Bus angeschlossen ist. Bei mehreren Slave-Stationen muss man diese immer gezielt ansprechen, also mit Match ROM und ROM-Identifiern arbeiten. Es sei denn, die Kommandos für die Slave lassen „die Gleichzeitigkeit der Befehlsausführung“ zu, s. vorheriges Beispiel mit dem Start der Temperaturmessung.

Search ROM (Befehls-Code: f0h)

Dieser Befehl dient dazu, dass der Master, nach den Einschalten, erst einmal seinen Bus „untersuchen“ kann, um festzustellen, welche Slaves bzw. welche Slave-Adressen (ROM-Identifier) überhaupt angeschlossen sind.
Mit Hilfe eines bestimmten Algorithmusses fragt er die Slaves ab und erhält als Ergebnis eine Liste mit den zugehörigen ROM-Codes.
Da hört sich zunächst interessant und hilfreich an, ist es aber in der Praxis nicht immer:

Beispiel:
Betrachten wir wieder Ihre Anlage mit den 10 Temperatur-Sensoren DS18S20.
Nach dem Einschalten sucht sich der Master die 10 ROM-Codes zusammen und speichert diese intern ab.
Aber was dann ? Er hat zwar 10 Codes ermittelt, weiß aber absolut nicht, welcher Sensor an welcher Stelle in Ihrem Aufbau welche Adresse hat. Mit anderen Worten: die Zuordnung - ROM-Code zum jeweiligen Sensor - funktioniert überhaupt nicht und damit ist der gesamte Suchvorgang eigentlich sinnlos.
Sie müssen hier, wie wir das später auch machen werden, zuerst von jeden Sensor, **vor dem Einbau**, den ROM-Code auslesen, diesem selber im Mikrocontroller-Programm in einer Liste hinterlegen und dann können Sie im weiteren Programm eindeutig festlegen bzw. programmieren: „Sensor mit dem ROM-Code ´xyz´ misst die Wassereinlasstemperatur, Sensor mit dem ROM-Code ´abc´ misst die Wasserauslasstemperatur, usw.“

Daher gehen wir hier zunächst nicht näher auf den Befehl Search ROM und den zugehörigen Such-Algorithmus ein. Nähere Informationen dazu finden sich aber z.B. in der MAXIM/DALLAS Application Note 187, [5].

2.4.5 Die Slave-spezifischen Funktionen

Setzen Sie nun einen bestimmten 1-Wire-Slave ein, so besitzt dieser naturgemäß seine eigenen besonderen Funktionen, die aber auf den bereits kennen gelernten 1-Wire-Basis- und Höheren Funktionen aufsetzen.
Daher besprechen wir diese besonderen Eigenschaften erst dann, wenn wir uns mit den entsprechenden Chips näher beschäftigen.

2.5 Weitere 1-Wire-Eigenschaften

Der 1-Wire-Bus besitzt noch weiter gehende Eigenschaften, auf die wir hier aber in diesen ersten Einführungsbetrachtungen zur Zeit nicht näher eingehen:

- Überprüfung des CRC-Wertes beim Empfang von Daten von einem 1-Wire-Slave, um Datenübertragungsfehler festzustellen.

- Verwendung des schnelleren Overdrive-Modus.

- Verwendung der parasitären Speisung für 1-Wire-Slaves.

- Realisierung von 1-Wire-Bussystemen mit großer Ausdehnung (10, 20, 30, … Slave-Stationen an Bussen mit bis zu 100 m Ausdehnung, [6]).

In den MAXIM/DALLAS-Applications Notes zum 1-Wire-Bus und im Internet finden Sie aber nähere Informationen zu diesen Punkten.

3. Die 1-Wire-Funktionen in ´C´ für 8051er

Die zuvor beschrieben Funktionen zum Betrieb des 1-Wire-Busses

- **ow_delay(…)**
- **ow_reset()**
- **ow_wr_bit(…)**
- **ow_wr_byte(…)**
- **ow_rd_bit()**
- **ow_rd_byte()**
- **ow_rd_rom()**
- **das globale Array ´rom_c[8]´,**
- **und die Port-Pin-Definition für die 1-Wire-Datenleitung DQ**

sind zunächst in ´C´ für 8051er-Mikrocontroller erstellt worden (unter µC/51) und werden in einer entsprechenden API-Bibliothek, bestehend aus den beiden Dateien ´**ow.c**´ und ´**ow.h**´, zusammengefasst.
Dadurch ist es dem Anwender möglich, diese Funktionssammlung einfach in sein eigenes Projekt mit einzubinden und die darin enthaltenen Funktionen leicht und sofort in seinem Programm zu verwenden.
Weiterhin ist so auch eine Änderung bzw. Anpassung der Funktionen möglich, um diese auf anderen Mikrocontroller-Plattformen verwenden zu können.

In unseren nachfolgenden 1-Wire-Projekten werden wir auf diese 1-Wire-API zurückgreifen und zu jedem 1-Wire-Baustein eine eigene, weitere API erstellen, die speziell dessen Eigenschaften unterstützt.

4. Die verfügbaren 1-Wire-Komponenten – eine Übersicht

Der kleine Nachteil beim 1-Wire-Bus ist ja, dass nur MAXIM/DALLAS die Lizenzen für dieses Bussystem besitzt, diese nicht für andere Produzenten freigegeben hat und somit weltweit der einzige Hersteller von 1-Wire-Komponenten ist.
Anderseits macht dieses die Sache für den Anwender recht überschaubar und mit den mittlerweile verfügbaren Chips lassen sich sehr leistungsfähige Sensor-/Aktor-Feldbussysteme aufbauen.
Unter www.maxim-ic.com/products/1-wire/ findet man ein sehr gutes 1-Wire-Einführungs-Tutorial (in englischer Sprache) und eine Übersicht über die aktuell verfügbaren 1-Wire-Produkte, **Abb.4.1**:

Abb.4.1: Die Welt der 1-Wire-Produkte

Die Welt der 1-Wire-Slaves teilt sich somit in **drei große Gruppen** auf:

1. Die „echten“ Chips

Also die 1-Wire-Slaves in einfacher Chip-Form, wie z.B. den DS18S20, **Abb.4.2**:

Abb.4.2: Der DS18S20 in „reinster“ Chip-Form

Wie man auf der 1-Wire-Web-Seite erkennen kann, **Abb.4.3**, sind zur Zeit 44 einzelne Chips aus verschiedenen Bereichen und ein Entwicklungskit verfügbar

Abb.4.3: Die Vielfalt der 1-Wire-Chips

2. Die iButtons

Die iButtons sind eine interessante, aber leider nicht ganz preiswerte MAXIM/DALLAS-Anwendung von 1-Wire-Komponneten, die sich schön verpackt, mit ihrer „Speise-Batterie“, in Edelstahl-Gehäusen in Knopf-Form befinden, **Abb.4.4**:

Abb.4.4: Der 1-Wire-Slave, hübsch und äußerst robust verpackt (hier:DS1923-F5)

Solche Bausteine trotzen widrigen Umwelteinflüssen und verrichten unter „schwierigen“ Arbeitsbedingungen ihren Dienst.
Die Abb.4.4 zeigt den Hydrochron-iButton DS1923-F5, der ein komplettes Meßsystem für die Umgebungstemperatur und die Umgebungs-Luftfeuchtigkeit enthält. Er kann, einmal gestartet, selbständig bis zu 8192 Messdatensätze aufnehmen, intern speichern und diese später dann via 1-Wire-Bus an den Auswert-Rechner übermitteln.
Dieser kompakte Edelstahl-Knopf kann also überall in unserer Umwelt „ausgesetzt“ werden und führt dort seine Messaufgaben mit einer garantierten Lebensdauer von bis zu 5 Jahren durch.
Allerdings muss der Anwender für diesen Messknopf ca. 70,00 € investieren (Stand: 01.2010).
Die **Abb.4.5** zeigt die zur Zeit verfügbaren iButtons (insges. 26 Stück):

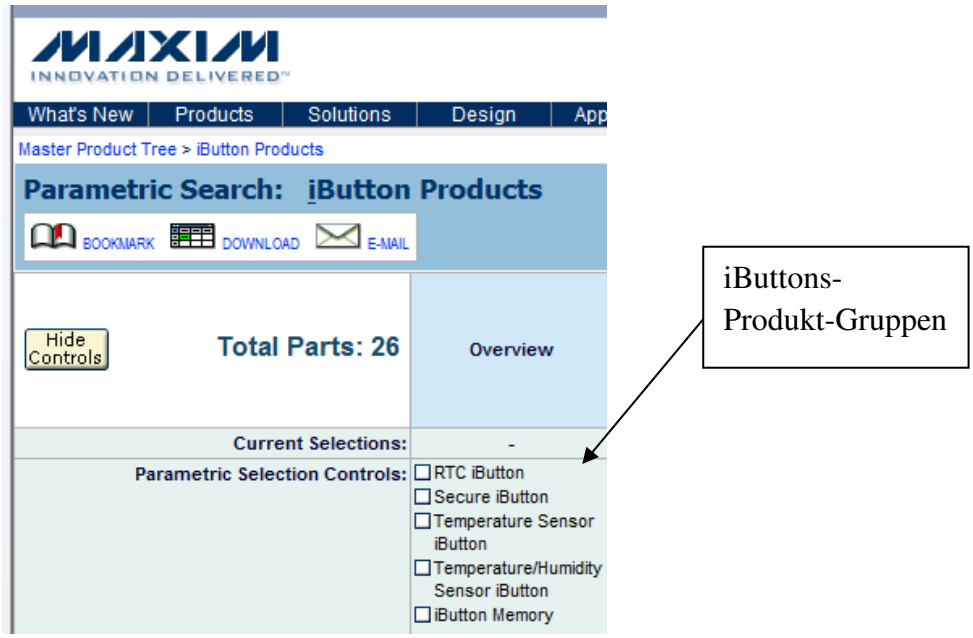


Abb.4.5: Die Familie der iButtons

3. Komplette 1-Wire-Module

Einige Hersteller von Elektronik-Komponenten benutzen nun die vorhandenen 1-Wire-Chips, um komplette eigene Sensor-/Aktor-Module herzustellen, die dann über den 1-Wire-Bus bedient werden können, **Abb.4.6** und **Abb.4.7**:

Abb.4.6: Der DS18S20 (Temperatursensor), schick in Edelstahl
(Firma HYGROSENS INSTRUMENTS, http://www.hygrosens.com/deutsch/home.html)

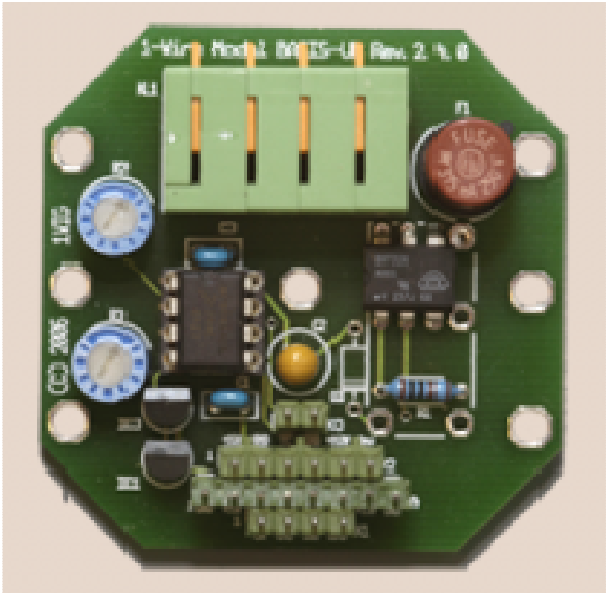


Abb.4.7: Komplettes 1-Wire-Sensor-/Aktor-Modul
(1-Wire Basis UP Modul Relais, von 1-Wire Interest Group, http://www.1-wire.de/1-Wire/1-Wire-Module/1-Wire-Basis-UP-Modul-Relais.html)

Solche Baugruppen sind dann zwar i.a. nicht mehr über den Bus speisbar (also keine parasitäre Speisung mehr möglich), benutzen aber dennoch das einfache und sichere 1-Wire-Datenübertragungsprotokoll (der verwendeten 1-Wire-Chips) mit nur einer Datenleitung und separat geführten Spannungsversorgungsleitungen.

Die **beiden einzigen Probleme** für den engagierten Praktiker, der sich einen 1-Wire-Bus aufbauen will, bestehen

- In der Beschaffung der Chips: hier muss man etwas im Internet suchen, wenn man Lieferanten finden möchte, die auch Einzel-Stückzahlen der ICs verkaufen und nicht gleich Stangen- oder Rollenware mit einigen hundert Stück.

- In der löt-technischen Handhabung der Chips, denn diese werden immer kleiner und man muss sich hier durchaus mit der SMD-Technik anfreunden, wenn man bestimmte Bausteine auf sein eigenes Board löten will.

Für unsere nachfolgenden ersten technischen Realisierungen des 1-Wire-Busses haben wir uns drei interessante, leicht lieferbare 1-Wire-Slaves ausgesucht und für diese eine kleine 1-Wire-Experimental-Platine (das ´**PT-1-Wire-ExBo**´) entwickelt, die über nur drei Leitungen an unser Mikrocontroller-System angeschlossen wird (natürlich funktioniert das Ganze auch mit andern Mikrocontroller-Systemen).
Die hierbei auf dem Board verwendbaren Chips sind:

- bis zu vier **DS18S20** (digitale Temperatursensoren),
- ein **DS2450**: vierfach A/D-Wandler,
- zwei **DS2408**: der eine zur Ansteuerung von Relais/Summer bzw. zur Abfrage von Tastern, der zweite zum Betrieb eines alphanumerischen LC-Displays mit bis zu 4 Zeilen á 20 Zeichen.

Beginnen wir also unseren Praxis-Teil mit einem Blick auf das PT-1-Wire-ExBo.

5. Das PT-1-Wire-Experimentalboard (PT-1-Wire-ExBo)

Das sich das PT-1-Wire-Experimentalbaord noch in der Entwicklungs- und Testphase befindet, werden wir den ersten 1-Wire-Slave, den digitalen Temperatursensor DS18S20, direkt an die 1-Wire-Datenleitung DQ und die Versorgungsspannung anschließen und betrieben.

Das PT-1-Wire-ExBo wird ca. **Mitte Mai 2010** verfügbar sein.

6. Der DS18S20 - Digitaler Temperatursensor

6.1 Allgemeines

Der DS18S20 ist ein digitaler Temperatursensor in einem kleinen TO-92er-Gehäuse, **Abb.6.1.1**:

Abb.6.1.1: Der DS18S20

Dieser Chip beinhaltet alles, was zur Temperaturmessung und zur Aufbereitung des Temperaturmesswertes bis hin zum digitalen Transfer des Ergebnisses über den 1-Wire-Bus notwendig ist.
Die wichtigsten Daten dieses Sensors sind in **Tab.6.1.1** aufgeführt, [4]:

- Temperaturmessbereich: -55°C … +125°C, Auflösung: 0,5°C, Genauigkeit ±0,5°C im Messbereich von -10°C … +85°C.
- 9 Bit Auflösung des Messwertes.
- Erhöhung der Auflösung durch zusätzliche Berechnungen im Mikrocontroller möglich.
- Umwandlungszeit (Zeitbedarf für eine Messung): 750 ms.
- Alarmfunktionen.
- Keine weiteren externen Komponenten erforderlich.
- Betriebsspannungsbereich: 3,0 … 5,0 V.
- Parasitäre Speisung über den 1-Wire-Bus möglich, aber nur bei Messungen bis 100°C. Sollen Temperaturen drüber hinaus gemessen werden, so muß der DS18S20 von extern gespeist werden.
- Einzigartiger fester interner 64 Bit-Identifikations-Code (ROM-Code).
- Datenübertragung über den 1-Wire-Bus.
- Gehäuse: TO-92 oder SO150.
- Bevorzugter Einsatz-Bereich: HVAC(Heating, Ventilating, Air Conditioning)-Systeme und überall dort, wo es um einfache digitale Temperaturmessungen geht.

Tab.6.1.1: Die wichtigsten Daten des DS18S20ers

Die **Abb.6.1.2** zeigt das Blockschaltbild dieses Bausteins und weitere Details können dem Datenblatt entnommen werden, [4].

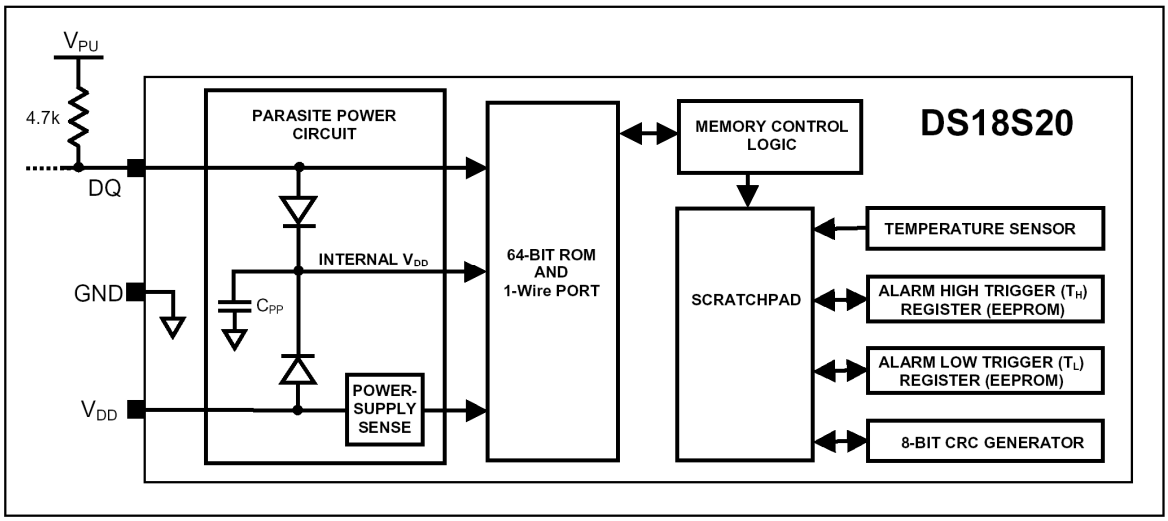


Abb.6.1.2: Das Blockschaltbild des DS18S20ers

Wesentlich sind hier zunächst zwei Funktionsblöcke:

1) Der **64-Bit große ROM-Bereich** in dem der individuelle Chip-ROM-Code fest abgelegt ist.
   Dieser Wert wird dann mit dem ´Read ROM´-Befehl ausgelesen (s. Kap. 2.4.4).

2) Der 9 Byte große **Scratchpad-Speicherbereich**: hierin werden die Temperaturmessdaten und weitere zusätzliche interne Informationen vom DS18S20er abgelegt.
   Diese Daten werden nachfolgend vom Mikrocontroller ausgelesen und dann weiter verarbeitet.

Wenn man den DS18S20 in seiner „Ursprungsform“ gemäß Abb.6.1.1 einsetzt, kann man bereits sehr gut Lufttemperaturen messen. Sollen aber Messungen in anderen Umgebungen erfolgen (in Flüssigkeiten (Wasser), in korrosiven Gasen oder in Bereichen mit hohen Feuchtigkeiten), so ist das in dieser Form nicht möglich, da sich der DS18S20er sicherlich recht schnell auflösen oder seine Anschlussbeine wegrosten würden.
Der Anwender muß in solchen Fällen den DS18S20er selber „gut verpacken“ oder direkt eine entsprechend geschützte Version einsetzen.
Wir haben uns für die zweite Möglichkeit entscheiden und einen, von Edelstahl umhüllten, Baustein eingesetzt, **Abb.6.1.3**:

Abb.6.1.3: Der DS18S20er im Edelstahl-Gewand
(Firma HYGROSENS INSTRUMENTS, http://www.hygrosens.com/deutsch/home.html, [7])

An der gesamten Programmierung und Handhabung des DS18S20ers ändert diese Verpackung natürlich nichts.
Am Ende des 2 m langen Anschlusskabels befindet sich ein **RJ12-Steckverbinder**, so das ein einfacher (Steck)Anschluß an jedes Mikrocontroller-System realisiert werden kann (Datenübertragung und Spannungsversorgung erfolgen über diesen Stecker).

Wir werden diesen Sensor mit einer externen Speisung betreiben, da wir durchaus Temperaturen oberhalb von 100°C erfassen wollen.

Der Anschluss des DS18S20ers

Der Anschluss dieses Sensors an den Mikrocontroller kann nun auf drei verschiedene Arten erfolgen:

1. DS18S20er als „nicht verpackter“ Einzelchip gem. Abb.6.1.1

Wir betrieben den Baustein hierbei mit lokaler Speisung und daher sieht der Anschluss wie folgt aus, **Abb.6.1.4**:

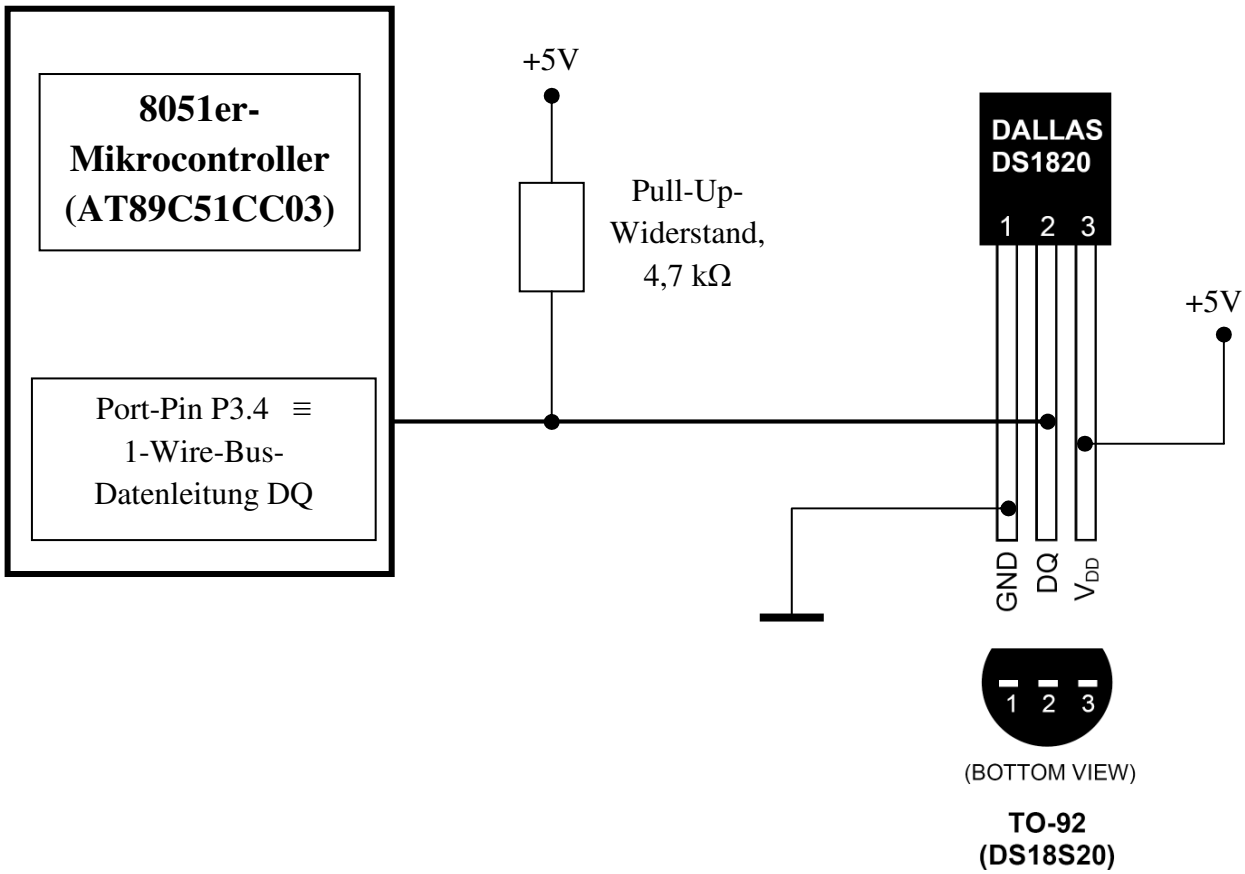


Abb.6.1.4: Der DS18S20er als Einzelchip angeschlossen (TO-92er Gehäuse)

Der notwendige Pull-Up-Widerstand wird hier auf der Master-Seite eingefügt.

2. DS18S20er als „verpackte Einheit“ gem. Abb.6.1.3

Bei dieser Version erfolgt der Anschluss an den Mikrocontroller über einen RJ12-Stecker bzw. über eine RJ12-Buchse, **Abb.6.1.5**:

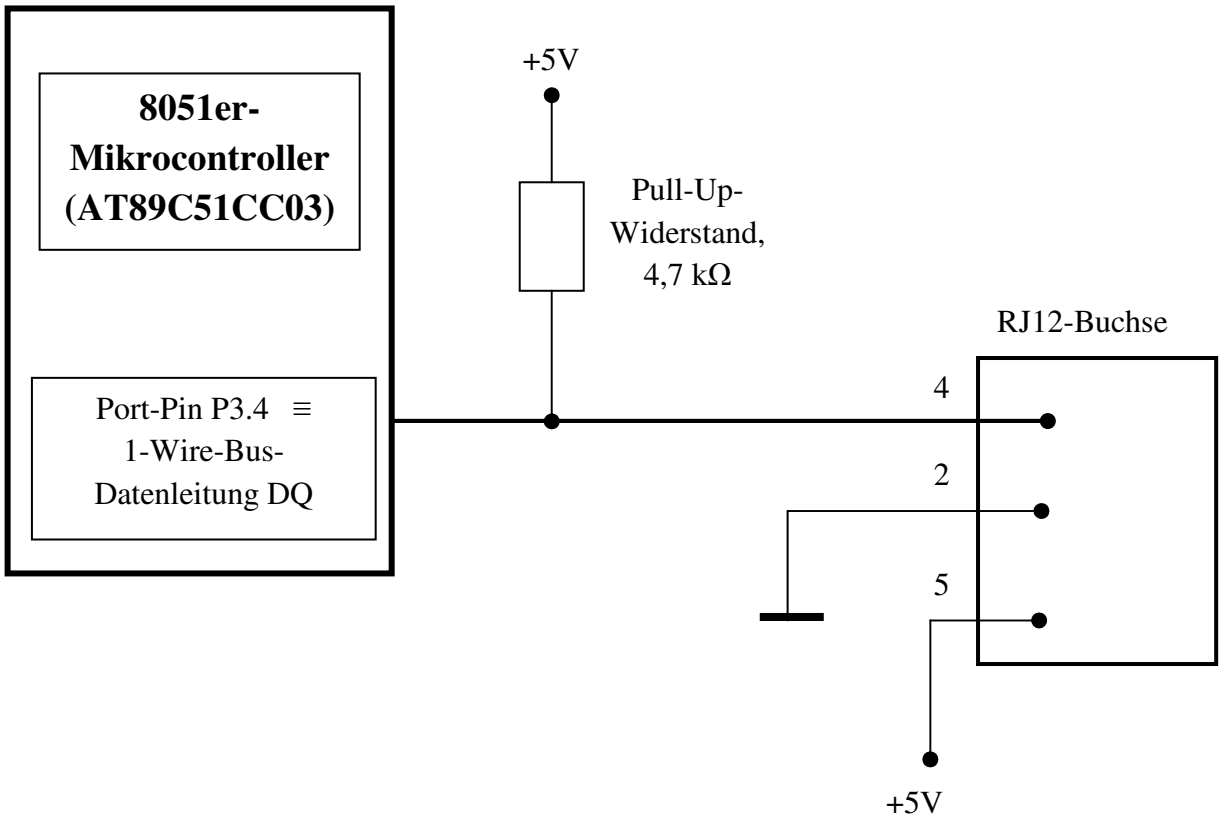


Abb.6.1.5: Anschluss eines eingebauten DS18S20ers über RJ12-Stecker/Buchse

3. DS18S20er-Anschluss am PT-1-Wire-ExBo

Auf diesem Experimentalboard kann sowohl ein einzelner DS18S20er in eine TO-92er-Transistorfassung gesteckt werden als auch ein Abschluss über RJ12-Buchsen erfolgen (s. Kap.5).

Kommen wir nun zum Betrieb des DS18S20ers.

6.2 Die DS18S20er-API

Unsere Sammlung von ´C´-Betriebsfunktionen für den DS18S20er, unsere **DS18S20er-API** (≡ Application Programming Interface = Schnittstelle zur Anwendungsprogrammierung), besteht im wesentlichen aus speziell für den DS18S20er entwickelten Funktionen, die aber alle auf unseren 1-Wire-Basisfunktionen (s. Kap. 2.4.3) und auf den Höheren Funktionen (s. Kap. 2.4.4) aufbauen.

Read ROM-Code

Als erstes wird zunächst **ein einzelner** DS18S20er an den 1-Wire-Bus angeschlossen, um dessen ROM-Code (≡ Chip-Adresse) auszulesen.
Diese Adresse müssen wir uns dann „irgendwie merken“, um sie später in das Gesamt-Programm einzubauen, wenn mit diesem Sensor ganz gezielt gearbeitet bzw. wenn dieser Sensor ganz gezielt angesprochen werden soll.
Dieser ROM-Code ist 8 Byte lang (≡ 64 Bit) und wie folgt aufgebaut, **Abb.6.2.1**:

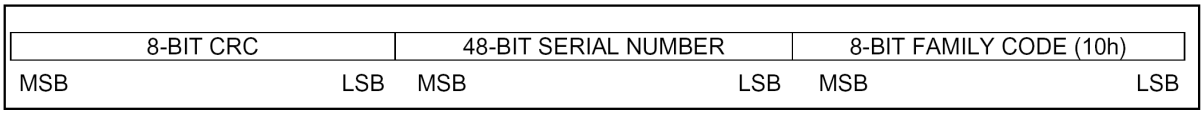


Abb.6.2.1: Der 64 Bit lange ROM-Code eines DS18S20ers

Die oberen 8 Bits bilden das CRC-Datensicherungs-Byte. Danach folgen 6 Bytes mit der eigentlichen Serien-Nummer des Chips und den Abschluss bildet der 8 Bit lange Familien-Code, der letztendlich angibt, um was für einen 1-Wire-Slave-Baustein es sich hier handelt. Der (feste) Wert 10h steht dabei für einen DS18S20er.
Beim Auslesen des ROM-Codes wird immer das niederwertigste Byte zuerst, also der Familien-Code, von Chip ausgesendet.
In ´C´ könnte eine Befehlsequenz, die diesen ROM-Code ausliest, unter Verwendung der bereits erstellten 1-Wire-Funktionen, nun wie folgt aussehen:

```
…….
// Einlesen des ROM-Codes
ow_rd_rom();          // Höhere 1-Wire-Funktion

// Ausgabe der 8 Bytes des ROM-Codes
// Als erster Wert wird das CRC ausgeben
printf("%02x  %02x  %02x  %02x  %02x  %02x  %02x  %02x
(HEX)\n",rom_c[7],rom_c[6],rom_c[5],rom_c[4],rom_c[3],
rom_c[2],rom_c[1],rom_c[0]);
…….
…….
```

Durch Aufruf der Funktion ´**ow_rd_rom()**´ wird der ROM-Code ausgelesen und im globalen Array ´rom_c[ ]´ abgespeichert.
Die nachfolgende Ausgabe stellt dann diese Werte auf dem Terminal-Bildschirm dar.

Um diese Adresse auszulesen, haben wir keine besondere DS1820er-API-Funktion geschrieben: die obigen beiden Programmzeilen werden einfach in das jeweilige Programm integriert, wenn es darum geht, die zugehörigen ROM-Identifier aus den 1-Wire-Slaves auszulesen.

Bevor wir nun den DS18S20er gezielt programmieren, hier zunächst ein Überblick, über den spezifischen Befehlsumfang, den dieser Chip versteht, **Tab.6.2.1**:

COMMAND	DESCRIPTION	PROTOCOL	1-Wire BUS ACTIVITY AFTER COMMAND IS ISSUED	NOTES
TEMPERATURE CONVERSION COMMANDS				
Convert T	Initiates temperature conversion.	44h	DS18S20 transmits conversion status to master (not applicable for parasite-powered DS18S20s).	1
MEMORY COMMANDS				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18S20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2 and 3 (T_H and T_L).	4Eh	Master transmits 2 data bytes to DS18S20.	3
Copy Scratchpad	Copies T_H and T_L data from the scratchpad to EEPROM.	48h	None	1
Recall E^2	Recalls T_H and T_L data from EEPROM to the scratchpad.	B8h	DS18S20 transmits recall status to master.	
Read Power Supply	Signals DS18S20 power supply mode to the master.	B4h	DS18S20 transmits supply status to master.	

Note 1: For parasite-powered DS18S20s, the master must enable a strong pullup on the 1-Wire bus during temperature conversions and copies from the scratchpad to EEPROM. No other bus activity may take place during this time.
Note 2: The master can interrupt the transmission of data at any time by issuing a reset.
Note 3: Both bytes must be written before a reset is issued.

Tab.6.2.1: Der Befehlsumfang des DS18S20ers

Beim jetzt folgenden Betrieb des DS18S20ers, bei der ganz konkreten Temperaturmessung, muss man **zwei verschiedene Fälle** unterscheiden:

- Nur ein einziger DS18S20er ist am Bus angeschlossen: in diesem Fall kann man auf eine explizite (individuelle) Adressierung des Chips verzichten und mit der **Skip-ROM**-Anweisung arbeiten.

- Sind dagegen mehrere DS18S20er am Bus aktiv, so muß jeder Sensor individuell über eine eigene Adresse mittels **Match ROM** angesprochen werden, damit sich auch nur dieser Sensor angesprochen fühlt.

Und nun die Details:

Nur ein DS18S20er am Bus

Die Temperatur-Messung mit dem Chip teilt sich in drei Schritte auf:

- **Start der Temperaturmessung** auslösen: dieses geschieht durch Aussendung des Befehls: „**Convert T**“.

- Da nun die **Conversion Time** (Umwandlungszeit) 750 ms dauert, muß auf der Master-Seite mindestens so lange gewartet werden, bis der Messwert ausgelesen werden kann.

- Nach Beendigung der Messung (Ablauf der Conversion Time): **Temperaturmesswert aus dem Scratch Pad, dem internen SRAM-Datenspeicher des DS18S20ers, auslesen**.
  Hierzu dient die Anweisung „Read Scratchpad“ mit dem Befehlsbyte beh.
  Nachdem der DS18S20er diesen Befehl erhalten hat, kann der Master die 9 Bytes des Scratch Pads nacheinander auslesen.

Somit sieht der etwas detaillierter beschriebene Ablauf zur Temperatur-Messung mit dem DS18S20er wie folgt aus:

- **Start des 1. Befehlszyklusses**:
  Master sendet Reset-Impuls, DS18S20er antwortet mit Presence-Impuls.

- Master sendet ´Skip ROM´-Befehl, da hier nur ein DS18S20er angeschlossen ist (Befehls-Code: cch).

- Master sendet den nächsten Befehl an den DS18S20er: Start der Messung ´Convert T´, Befehls-Code 44h.
  Damit ist für den Master der 1. Befehlszyklus beendet.

- Der DS182S20er misst nun die Temperatur und legt diese in den Scratch Pad-Registern ab.

- Master muss nun noch mindestens 750 ms warten (≡ Conversion Time), bevor er den 2. Befehlszyklus startet.

- **Start des 2. Befehlszyklusses**:
  Master sendet Reset-Impuls, DS18S20er antwortet mit Presence-Impuls.

- Master sendet ´Skip ROM´-Befehl, da hier nur ein DS18S20er angeschlossen ist (Befehls-Code: cch).

- Master sendet den nächsten Befehl an den DS18S20er: Auslesen des Scratch Pads, ´Read Scratch Pad´, Befehlscode: beh.

- Danach kann der Master sofort die 9 Bytes des Scratch Pads vom DS18S20er auslesen und zwar mit Hilfe der Funktion ´**ow_rd_byte()**´ und die Werte werden **im neu angelegten globalen Array ´ds1820[ ]´** abgelegt.

- Damit ist der 2. Befehlszyklus und die gesamte Temperaturmessung abgeschlossen

Man muß hier also immer beachten, das die Kommunikation zwischen Master und Slave in einzelnen Befehlszyklen abläuft und jeder Befehlszyklus mit einem Reset-Impuls beginnt, auf den der (funktionierende) Slave mit einem Presence-Impuls antwortet.
Daraufhin kann der Master einen Befehl senden, auf den der Slave reagiert. Danach ist dann dieser Befehlszyklus abgeschlossen und ein neuer Zyklus kann vom Master initiiert werden.

Das Ganze haben wir in ´C´ nun so realisiert:

```
// 1. Befehlszyklus: Messung starten
// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
if(ow_reset())
{
  printf("\n\n\n\n  Fehler beim 1. Reset: KEIN Presence vom Slave !");
  printf("\n\n  Reset druecken ... ");
  while(1);
}

// Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
ow_wr_byte(0xcc);

// Befehl: Start Conversion an DS1820
ow_wr_byte(0x44);

// Warte-Zeit, da Chip eine max. Conversion Time von 750 ms hat !
_wait_ms(800);              // 800 ms warten

// 2. Befehlszyklus: Messwerte (Scratch Pad) auslesen
// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
if(ow_reset())
{
  printf("\n\n\n\n  Fehler beim 2. Reset: KEIN Presence vom Slave !");
  printf("\n\n  Reset druecken ... ");
  while(1);
}

// Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
ow_wr_byte(0xcc);

// Befehl: Auslesen des Scratch Pads vom DS1820
ow_wr_byte(0xbe);

// Antwort einlesen: 9 Byte große Scratch Pad-Inhalt einlesen
for (i=0; i<9; i++)
{
  ds1820[i] = ow_rd_byte();
```

```
}
```

(Diese Programm-Zeilen sind ein Teil der DS18S20er-API-Funktion, die wir nachfolgend noch komplett entwickeln werden)

Kommen wir jetzt noch zur **Auswertung der Messwerte**.
Den Aufbau des DS18S20ers Scratch Pads zeigt die **Abb.6.2.2**:

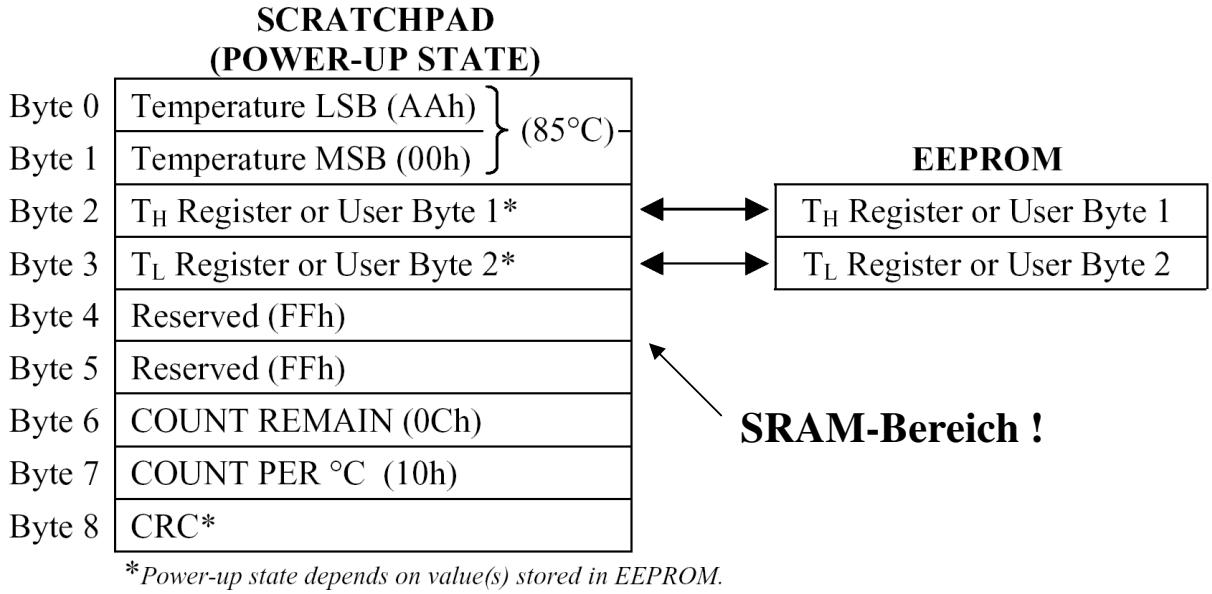


Abb.6.2.2: Das DS18S20er Scratch Pad

Das Scratch Pad ist im DS18S20er als **SRAM-Speicher** (statisches RAM) realisiert, 9 Byte groß und die einzelnen Bytes bedeuten:

- **Byte 0 und Byte 1**: hierin steht der Temperaturmesswert, aufgeteilt in LSB (≡ Betrag der Temperatur) und MSB (≡ Vorzeichen der Temperatur).
  Ist der Temperaturmesswert negativ, so erfolgt die Darstellung des Wertes im so genannten 2er-Komplement (s. nachfolgend).

- **Byte 2 und Byte 3**: in diesen beiden Bytes können entweder Grenztemperaturen (High- und Low-Temperatur) eingeschrieben werden oder der Anwender kann diese beiden Bytes zur Abspeicherung beliebiger Werte (z.B. einer Stations-Nummer) benutzen.
  Der DS18S20er vergleicht jede gemessene Temperatur mit diesen beiden Werten und wird die untere Grenze T_L unterschritten oder die obere Grenze T_H überschritten, so wird intern ein Alarm-Bit gesetzt, das vom Anwender abgefragt und dann weiter ausgewertet werden kann (≡ Kennzeichen dafür, dass eine Grenztemperatur unter- bzw. überschritten wurde).
  (Hierauf werden wir allerdings nicht näher eingehen, weitere Informationen dazu fin-

den sich im Datenblatt zum DS18S20).

- **Byte 4 und Byte 5**: sind reserviert für weitere zukünftige Anwendungen und somit für uns zur Zeit nicht weiter interessant.

- **Byte 6 und Byte 7**: hierin sind Zusatzwerte enthalten, mit denen man durch weiter gehende Berechnungen die Auslösung der Messung (des Messwertes) erhöhen kann. (Hierauf werden wir allerdings nicht näher eingehen, weitere Informationen dazu finden sich im Datenblatt zum DS18S20).

- **Byte 8**: in diesem Byte ist die CRC-Sicherungssumme über alle vorhergehenden Bytes enthalten. Dieses CRC-Byte kann zur Überprüfung von Datenübertragungsfehlern herangezogen werden.
  (Hierauf werden wir allerdings nicht näher eingehen, weitere Informationen dazu finden sich im Datenblatt zum DS18S20).

Beim Auslesen dieses Speicherbereichs überträgt der DS18S20er das Byte 0 als Erstes.

Zwei Besonderheiten sind hier noch zu beachten:

1. Der Temperaturmesswert „85°C“

Bei den beiden Bytes 0 und 1 ist der Temperaturmesswert 85°C (= 00aah) angegeben und dieser Wert hat eine besondere Bedeutung, er ist nämlich ein „Fehler-Anzeige-Wert“ und erscheint immer bei folgenden Problemen (wenn die externe Temperatur eben garantiert nicht 85°C ist, der DS18S20 diesen Wert aber zurückgibt):
Der Wert „85“ ist der **Power-On-Wert** des Sensors, d.h. immer wenn der DS18S20er eingeschaltet, also mit Spannung versorgt wird, wird dieser Wert als definierter Anfangswert in die beiden Register geschrieben.
Wenn nun KEINE korrekte oder eine fehlerhafte Wandlung durchgeführt wird, so bleibt dieser Wert erhalten und der Anwender liest immer nur „85“ aus.
Die Gründe für eine solche Fehlbedienung können sein:

- Es wird einfach nur der Inhalt des Scratch Pads ausgelesen, **ohne** vorher den Befehl zur Wandlung zu geben, die ´Convert T´-Anweisung wurde einfach vergessen.
  Daher wandelt der DS18S20er gar nicht, sondern behält immer nur die „85“ in den Ergebnisregistern.
  Vergessen Sie also nie, eine Temperaturmessung mit dem ´Convert T´-Befehl zu beginnen.

- Die korrekt Einhaltung der notwendigen Conversion Time wurde nicht beachtet: der DS18S20er benötigt (mindestens) 750 ms Wandlungszeit, d.h.: nachdem der Convert T-Befehl gesendet wurde, muss man mindestes 750 ms warten, ehe man den Scratch Pad mit dem Wandlungsergebnis auslesen kann.
  Wird früher ausgelesen, z.B. nach 500 ms, so ist die Wandlung noch nicht korrekt er-

folgt und in den Ergebnis-Registern steht wiederum nur „85“ (oder noch der alte, vorherige richtig gewandelte Wert).

- Bei der Wandlung der Temperatur benötigt der DS18S20er mehr Strom als im Ruhezustand. Erhält der Baustein also während der Wandlung „zu wenig“ Strom, so bricht er die Wandlung ab und in den Registern steht wiederum nur „85“.
  Die Ursachen für dieses Problem können sein: ein falsch dimensionierter Pull-Up-Widerstand bei der parasitären Speisung des Chips oder eine falsch ausgelegte externe Versorgung für den Baustein (s. Kap. 2.3.2).

Eines ist natürlich hier zu beachten: der Wert „85“ kann auch ein richtiger, echter Messwert sein. Wenn nämlich alle Messungen bei andere Temperaturen (z.B. bei Raumtemperatur) korrekt sind und man taucht dann den Sensor z.B. in eine heiße Flüssigkeit ein und er gibt darauf hin den Wert „85“ zurück, so ist das sicherlich der aktuelle Messwert.
Nur wenn bei allen Messungen (permanent) der Wert „85“ zurück kommt, ist etwas „faul“.

2. Die Register T_H und T_L

Da der Scratch Pad des Ds18S20ers intern als SRAM realisiert ist, verliert dieser Bereich beim Ausschalten der Betriebsspannung seinen Inhalt.
Für den Messwert in den Bytes 0 und 1 ist das sicherlich kein Problem, denn die Messungen sind ja beendet.
Anders sieht das mit den Werten in den Registern T_H und T_L aus: hierin sind ja entweder Grenztemperaturen oder z.B. eine Stations-Nummer hinterlegt und diese Angaben sollten sinnvoller Weise, beim Abschalten der Betriebsspannung, erhalten bleiben.
Daher hat der Anwender die Möglichkeit, diese Werte durch einen besonderen Befehl, ganz bewusst und gezielt, in zwei einzelne EEPROM-Stellen im DS18S20er abzuspeichern.
Dort liegen sie dann Spannungsausfall-gesichert und nach jedem Einschalten der Spannungsversorgung lädt der DS18S20er, selbsttätig und ohne Zutun des Anwenders, diese beiden Bytes aus dem EEPROM-Bereich in die zugehörigen Register T_H und T_L des Scratch Pads zurück.
Darüber hinaus gibt es auch noch einen eigenen Befehl (´Recall E^2´, s. Tab.6.2.1), um dieses Rückladen, während des Betriebs, vom Anwender-Programm aus zu initiieren.

Wichtig ist nun noch …

Die Umrechnung der Messwerte

Bei **positiven Temperatur-Werten** (Byte 1 (MSB) im Scratch Pad hat den Wert 0) ist die Umrechnung recht einfach: man teilt den Wert aus Byte 0 (LSB) des Scratch Pads einfach durch 2. Man muß hier allerdings beachten, das dieses nur beim durchgängigen Arbeiten mit float-Variablen funktioniert, denn bei int-Variablen fällt ja der Nachkomma-Anteil immer ersatzlos weg.

Bei **negativen Temperatur-Werten** (Byte 1 (MSB) im Scratch Pad hat den Wert ffh) ist zu beachten, dass die Temperaturdarstellung im so genannten **2er-Komplement** erfolgt, der Betrag des Wertes also insgesamt wie folgt bestimmt werden muß:

- der Wert aus Byte 0 (LSB) wird Bit-weise invertiert,
- dann wird die Zahl ´1´ hinzu addiert,
- das Ergebnis dann durch 2 geteilt und abschließend
- wird noch mit (-1) multipliziert.

Somit könnte die gesamte Umrechnung des Messwertes in ´C´ wie folgt aussehen:

```
float ftemp;                // Temperatur als float-Var.
…….
…….
// Die Scratch Pad Werte sind im globalen Array ´ds1820[ ]´
// abgelegt
…….
// Temperatur berechnen, als float-Wert
if(ds1820[1] == 0)              // Positive Temp.-Werte
{
  ftemp = ds1820[0]/2.0;
}
else                            // Neg. Temp.-Werte
{
  ftemp = (~ds1820[0])+1;       // 2er-Komplement
  ftemp = (ftemp*(-1))/2;
}

// Ausgabe
printf("\n%5.1f",ftemp);
```

Mehrere DS18S20er am Bus

Sind nun mehrere DS18S20er am 1-Wire-Bus angeschlossen, so ist der zuvor beschriebene Ablauf zu ändern bzw. zu ergänzen, da ja nun die einzelnen Slaves gezielt über ihre individuelle Adresse (ROM-Code) angesprochen werden müssen, damit sie eindeutig reagieren.
Wir gehen einmal davon aus, das die ROM-Codes der einzelnen Slaves bereits, wie am Anfang dieses Kapitel beschrieben, ermittelt worden und somit bekannt sind.
Dann muß der Kommunikationsablauf insofern geändert werden, dass die ´Skip ROM´-Anweisung ersetzt wird durch den ´Match-ROM-Befehl, gefolgt durch das Aussenden des gewünschten ROM-Codes (≡ 8 Byte ≡ 64 Bit lang).
Die Ersetzung sieht dann so aus:

- **Start des 1. Befehlszyklusses**:
  Master sendet Reset-Impuls, DS18S20er antwortet mit Presence-Impuls.

- ***Master sendet ´Match ROM´-Befehl aus (Befehls-Code: 55h), gefolgt vom gewünschten ROM-Code.***

- Master sendet den ersten eigentlichen Befehl an den DS18S20er: Start der Messung ´Convert T´, Befehls-Code: 44h
  Damit ist für den Master der 1. Befehlszyklus beendet.

- Der DS182S20er misst nun die Temperatur und legt diese in den Scratch Pad-Registern ab.

- Master muss nun noch mindestens 750 ms warten (≡ Conversion Time), bevor er den 2. Befehlszyklus startet.

- **Start des 2. Befehlszyklusses**:
  Master sendet Reset-Impuls, DS18S20er antwortet mit Presence-Impuls.

- ***Master sendet ´Match ROM´-Befehl aus (Befehls-Code: 55h), gefolgt vom gewünschten ROM-Code.***

- Master sendet den zweiten eigentlichen Befehl an den DS18S20er: Auslesen des Scratch Pads, ´Read Scratch Pad´, Befehls-Code: beh.

- Danach kann der Master sofort die 9 Bytes des Scratch Pads vom DS18S20er auslesen und zwar mit Hilfe der Funktion ´**ow_rd_byte()**´ und die Werte werden im globalen Array ´ds1820[ ]´ abgelegt.

- Damit ist der 2. Befehlszyklus und die gesamte Temperaturmessung abgeschlossen

In ´C´ sieht diese geänderte Stelle dann wie folgt aus:

```
// Match ROM-Befehl und ROM-Code aussenden
ow_wr_byte(0x55);
for (i=0; i<8; i++) ow_wr_byte(ds1820_mat[s_adr][7-i])
```

wobei wir hier davon ausgehen, dass die ROM-Codes der einzelnen, am Bus angeschlossenen DS18S20er in einem globalen Feld (Array, Matrix) namens ´**ds1820_mat[ ]**´ hinterlegt sind, das folgenden Aufbau hat:

```
// Matrix für bis zu 5 angeschlossenen DS18S20er-Slaves:
//   -  Stations-Nummern:  0 - 4
//   -  Jeder Slave hat einen 8 Byte großen ROM-Code.
//   -  Jede Zeile in der Matrix entspricht dem ROM-Code eines Slaves.
//   -  Eintragungen fangen beim CRC-Code an, letztes Byte: Family-Code.
code unsigned char ds1820_mat[5][8] = {
{0x3f,0x00,0x08,0x01,0xdb,0x13,0x6c,0x10},
{0,0,0,0,0,0,0,0},
{0x0c,0x00,0x08,0x01,0xdb,0x08,0x21,0x10},
{0,0,0,0,0,0,0,0},
{0x54,0x00,0x08,0x01,0x86,0x70,0x0a,0x10}
};
```

ROM-Code von Station-Nr. 0

ROM-Code von Station-Nr. 4

Wir haben hier ein Feld aus unsigned char-Variablen (≡ Byte-Variablen) für maximal 5 DS18S20er am Bus angelegt. Diese Matrix hat somit die Dimension 5 * 8, da der ROM-Code von jeden DS18S20er aus 8 Bytes besteht.
Jede Zeile des Arrays entspricht somit dem ROM-Code eine Slave-Station.
Sind in einer Zeile lediglich acht Nullen eingetragen, so ist diese Zeile nicht durch einen ROM-Code besetzt, d.h. zu dieser Zeile existiert kein DS18S20er.
In unserem Bespiel haben wir also drei DS18S20er am Bus angeschlossen und ihre ROM-Codes in die Matrix eingetragen.
Der Zusatz ´**code**´ in der Definitionszeile des Arrays weist den Compiler an, diese Matrix, die ja im laufenden Programm nie geändert wird, fest in den Programmspeicher abzulegen. Damit spart man wertvollen Datenspeicher (im unserem Mikrocontroller befinden sich zwar ON-Chip 64 kByte Programmspeicher, aber „nur“ 2 kByte Datenspeicher).

Um die Adressierung des einzelnen Slaves im Programm einfacher durchführen zu können, haben wir jedem DS18S20er eine **Stations-Nummer** zugeordnet, die seiner Zeilennummer in der ROM-Code-Matrix entspricht: also die Station, deren ROM-Code in der Zeile 0 der Matrix steht, bekommt die Stationsnummer 0 zugeordnet, usw. bis hin zur Stationsnummer 4.
Diese Stations-Nummer wird auch fest im jeweiligen DS18S20er einprogrammiert und zwar in den Bytes T_H und T_L im internen DS18S20er-Scratch Pad (es gibt einen speziellen Befehl zum Beschreiben von T_H und T_L, den wir nachfolgend noch besprechen werden).
Und zwar haben wir festgelegt:

- T_H (Byte 2) ≡ immer den Wert 00h
- T_L (Byte 3) ≡ Stationsnummer: 00h - feh (0 - 254)
- Auf das Arbeiten mit Grenztemperaturen verzichten wir hier also.

Wenn wir dann später die verschiedenen DS18S20er am Bus abfragen, so arbeiten wir vorzugsweise mit diesen Stationsnummern und nicht mit den etwas unhandlicheren 8 Byte langen ROM-Codes.

Wir haben nun den Betrieb (die Abfrage) eines einzigen DS18S20ers und den Betrieb von mehreren DS18S20ern am 1-Wire-Bus in einer einzigen Funktion namens ´**ds1820_rd_temp(…)**´ zusammen gefasst, an die immer die entsprechende Stationsnummer

(≡ 00h … feh ≡ 0 … 254 ≡ Zeilen-Nummer des jeweiligen DS18S20ers in der Matrix) übergeben wird. In unserem Beispiel können also die Stations-Nummern 0 … 4 verwendet werden.
Übergibt man dagegen die Nummer ffh (255), so ist das für die Funktion das Zeichen, dass nur ein einziger DS18S20er am Bus angeschlossen ist und somit mit ´Skip ROM´ anstelle von ´Match ROM´ gearbeitet wird.
Diese **erste Funktion unserer DS18S20er-API** hat demnach das folgende Aussehen:

```
/*** Abfrage eines DS18S20ers mit der Slave-Adresse s_adr ***/

void ds1820_rd_temp(unsigned char s_adr)
{
  unsigned char i;

  // 1. Befehlszyklus: Messung starten
  // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
  if(ow_reset())
  {
    printf("\n\n\n\n  Fehler beim 1. Reset: KEIN Presence vom Slave !");
    printf("\n\n  Reset druecken ... ");
    while(1);
  }

  // Prüfen der Slave-Adresse
  if(s_adr == 255)                // Nur ein Slave am Bus angeschlossen
  {
    // Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
    ow_wr_byte(0xcc);
  }
  else                           // Mehrere Slaves am Bus
  {
    // Match ROM-Befehl und ROM-Code aussenden
    ow_wr_byte(0x55);
    for (i=0; i<8; i++) ow_wr_byte(ds1820_mat[s_adr][7-i]);
  }

  // Befehl: Start Conversion an DS1820
  ow_wr_byte(0x44);

  _wait_ms(800);        // 800 ms Warte-Zeit, da Chip eine max. Conversion
                        // Time von 750 ms hat !

  // 2. Befehlszyklus: Messwerte (Scratch Pad) auslesen
  // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
  if(ow_reset())
  {
    printf("\n\n\n\n  Fehler beim 2. Reset: KEIN Presence vom Slave !");
    printf("\n\n  Reset druecken ... ");
    while(1);
  }

  // Prüfen der Slave-Adresse
  if(s_adr == 255)                // Nur ein Slave am Bus angeschlossen
  {
    // Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
    ow_wr_byte(0xcc);
  }
```

```
  else                          // Mehrere Slaves am Bus
  {
    // Match ROM-Befehl und ROM-Code aussenden
    ow_wr_byte(0x55);
    for (i=0; i<8; i++) ow_wr_byte(ds1820_mat[s_adr][7-i]);
  }

  // Befehl: Auslesen des Scratch Pads vom DS1820
  ow_wr_byte(0xbe);

  // Antwort einlesen: 9 Byte große Scratch Pad-Inhalt einlesen
  for (i=0; i<9; i++)
  {
    ds1820[i] = ow_rd_byte();
  }
}
```

Das Beschreiben der Register T_H und T_L im Scratch Pad

Als nächstes benötigen wir noch eine Funktion, um die Register T_H und T_L im Scratch Pad des DS18S20ers zu beschreiben.
Wir gehen hier zunächst einmal davon aus, dass nur ein einziger DS18S20er am Bus angeschlossen ist und wir somit vereinfachend mit der ´Skip ROM´-Anweisung arbeiten können. Eine Erweiterung auf die Verwendung von ´Match ROM´, mit nachfolgender Übertragung des ROM-Codes, wenn mehrere DS18S20er angeschlossen sind, ist später dann einfach durchführbar.
Zunächst einmal eine Übersicht über den Ablauf der notwendigen Befehlsfolgen:

- **Start des 1. Befehlszyklusses**:
  Master sendet Reset-Impuls, DS18S20er antwortet mit Presence-Impuls.

- Master sendet ´Skip ROM´-Befehl, da hier nur ein DS18S20er angeschlossen ist, Befehls-Code: cch.

- Master sendet den Schreibbefehl für das Scratch Pad: ´Write Scratch Pad´, Befehls-Code 4eh.

- Danach sendet der Master unmittelbar zuerst den Wert für das Register T_H und dann als zweites den Wert für das Register T_L.

- Zum Abschluss dieses Transfers ist es nun erforderlich, dass der Master einen Reset-Impuls sendet (und der Slave mit einem Presence-Impuls antwortet).

  Nun befinden sich die beiden Werte in den Registern T_H und T_L (s. Abb.6.2.2), allerdings erst im SRAM-Bereich !
  Und das bedeutet: wenn man jetzt die Betriebsspannung des DS18S20ers ausschaltet, sind diese Werte wieder weg.
  Man muß diese beiden Daten also erst noch fest, Spannungsausfall-geschützt, in den

EEPROM-Bereich (s. Abb.6.2.2) einschreiben.
Und dazu ist jetzt noch ein zweiter Befehlszyklus notwendig.
Da der Master als Letztes bereits einen Reset-Impuls gesendet und einen Presence-Impuls vom DS18S20er erhalten hat, kann es hier sofort weiter gehen mit:

- **Start des 2. Befehlszyklusses**:
- Master sendet ´Skip ROM´-Befehl, da hier nur ein DS18S20er angeschlossen ist, Befehls-Code: cch.

- Master sendet den Befehl zum Kopieren der beide Werte vom SRAM-Bereich in den EEPROM-Bereich: ´**Copy Scratch Pad**´, Befehls-Code 48h.

- Abschließend müssen mindestens 10 ms gewartet werden, bis der Kopiervorgang intern beim DS18S20er beendet worden ist, bevor der Master einen neuen Befehlszyklus initiieren kann.

Die Implementierung dieses Ablaufs in eine entsprechende ´C´-Funktion namens ´**ds1820_wr_t_reg(…)**´, an die die beiden Werte für die Register T_H und T_L übergeben werden, ist jetzt nicht mehr besonders schwierig:

```
/* Übertragung von 2 Bytes in die Register TH und TL des Scratch Pads */

void ds1820_wr_t_reg(unsigned char TH, unsigned char TL)
{
  // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
  if(ow_reset())
  {
    printf("\n\n\n\n  Fehler beim Schreiben des Scratch Pads 1: KEIN
            Presence vom Slave !");
    printf("\n\n  Reset druecken ... ");
    while(1);
  }

  /*****************************************/
  /*** 1. Werte in Scratch Pad einschreiben ***/
  /*****************************************/

  // Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
  ow_wr_byte(0xcc);

  // Schreib-Befehl senden: "WRITE SCRATCHPAD" = 0x4e
  ow_wr_byte(0x4e);

  // Byte 1 senden: Reg TH
  ow_wr_byte(TH);

  // Byte 2 senden: Reg TL
  ow_wr_byte(TL);

  // Master sendet abschließenden Reset-Impuls
  if(ow_reset())
  {
    printf("\n\n\n\n  Fehler beim Schreiben des Scratch Pads 2: KEIN
```

```
            Presence vom Slave !");
    printf("\n\n  Reset druecken ... ");
    while(1);
  }

  /*************************************************/
  /*** 2. Werte vom Scratch Pad ins EEPROM kopieren ***/
  /*************************************************/

  // Skip ROM-Befehl, da nur ein DS1820 angeschlossen ist
  ow_wr_byte(0xcc);

  // Kopieren ins Scratch Pad (EEPROM)-Befehl senden:
  // "COPY SCRATCHPAD" = 0x48
  ow_wr_byte(0x48);

  _wait_ms(15);           // Wartezeit f. EPROM-Programmierung
}
```

Damit haben wir es geschafft, unser erster Entwurf der **DS18S20er-API** ist fertig und beinhaltet:

- die Funktion `ds1820_rd_temp(…)`
- die Funktion `ds1820_wr_t_reg(…)`
- das globale Array `ds1820[.]` und
- das globale Array `ds1820_mat[.][.]`.

Aus diesen vier Grundelementen „bauen“ wir uns nun die beiden API-Files ´**ds1820.c**´ und ´**ds1820.h**´ zusammen und können damit ab jetzt sehr einfach in unseren Anwendungsprogrammen arbeiten.
Das wollen wir auch gleich einmal vorführen …

6.3 Das DS18S20er-Demo-Programm

Mit den beiden API-Funktionssammlungen:

- ´**ow.c**´ und ´**ow.h**´ für den 1-Wire-Bus
  und
- ´**ds1820.c**´ und ´**ds1820.h**´ für den DS18S20er

haben wir nun ein kleines, aber bereits sehr leistungsfähiges Demo-Programm namens ´**ds1820_dem.c**´ zusammengestellt.
Zur Realisierung dieser Software werden unter µC/51 beim Anlegen eines ´C´-Projektes mittels MakeWiz diese beiden ´C´-Files mit ihren Funktionen eingebunden, **Abb.6.3.1**:

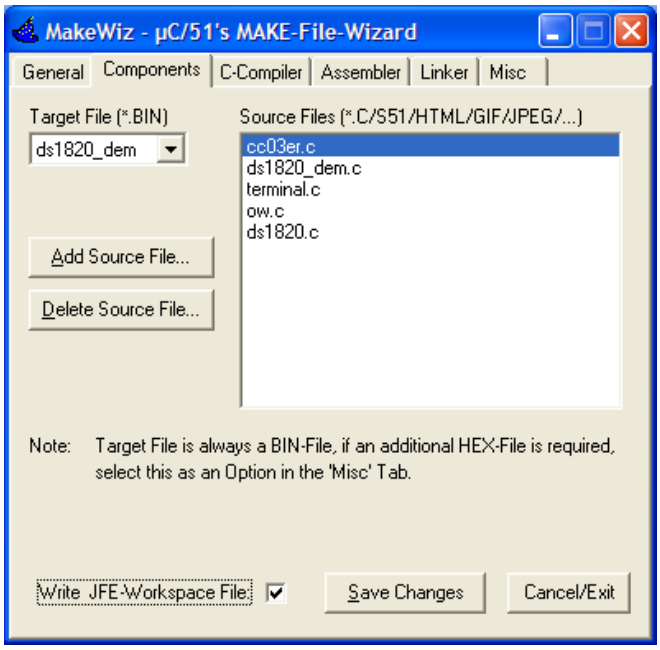


Abb.6.3.1. Das Einfügen der ´C´-Funktions-Files in das Projekt mit MakeWiz (µC/51)

Und im eigentlichen ´C´-Source-File fügt man die Header-Dateien mit der #include-Anweisung ein:

```
/*** Einbinden von Include-Dateien ***/
#include <stdio.h>          // Standard Ein-/Ausgabefunktionen
#include <at89c51cc03.h>    // CC03er-Register

#include "cc03er.h"         // CC03er-Funktionen
#include "terminal.h"       // Terminal-Funktionen

#include "ow.h"             // 1-Wire-Bus-API
#include "ds1820.h"         // DS1820er-API
```

Damit stehen dem Anwender jetzt in seinem Programm alle 1-Wire- und alle DS18S20er-Funktionen zur freien Verfügung.

Nach dem Download (File: ´**ds1820_dem.hex**´) und dem Start des DS18S20er-Demo-Programms erscheint auf dem Terminal-Bildschirm (z.B. HyperTerm) ein kleines Auswahl-Menü, **Abb.6.3.2**:

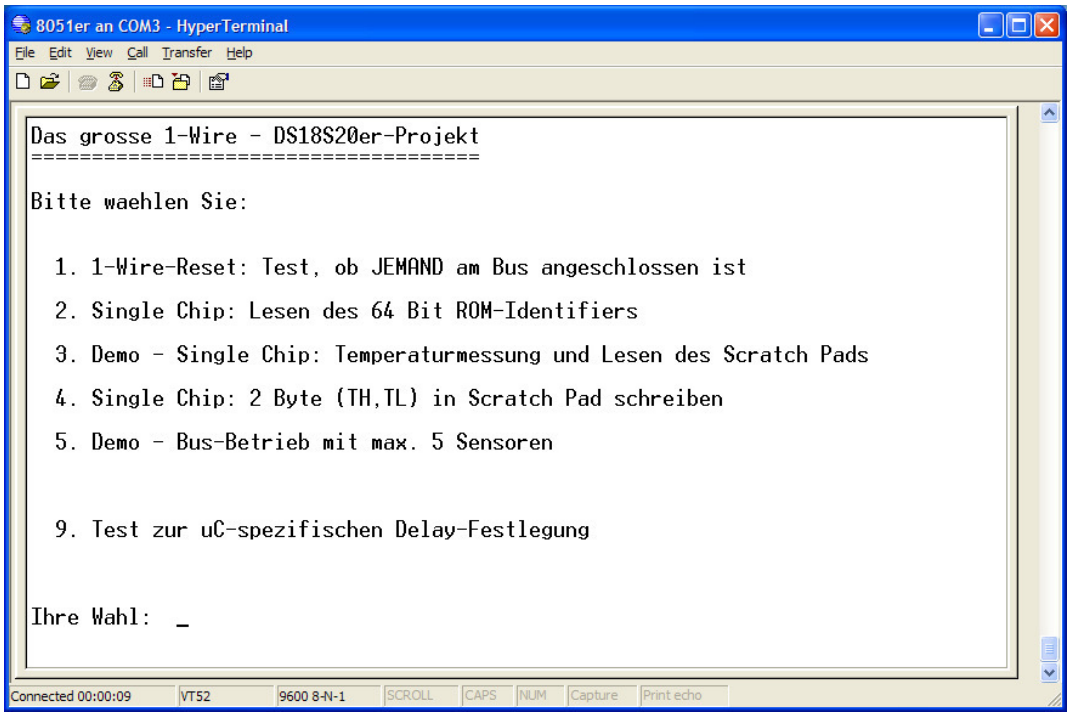


Abb.6.3.2: Das Auswahlmenü des DS18S20er-Demo-Programms

1. 1-Wire-Reset: Test, ob JEMAND am Bus angeschlossen ist

Unter diesem Menü-Punkt lässt sich feststellen, ob der 1-Wire-Bus in Ordnung ist und ob sich mindestens ein angeschlossener Slave zurück meldet:
Der Master sendet dazu einen Reset-Impuls und überprüft dann, ob er (mindestens) einen Presence-Impuls zurück erhält, **Abb.6.3.3**:

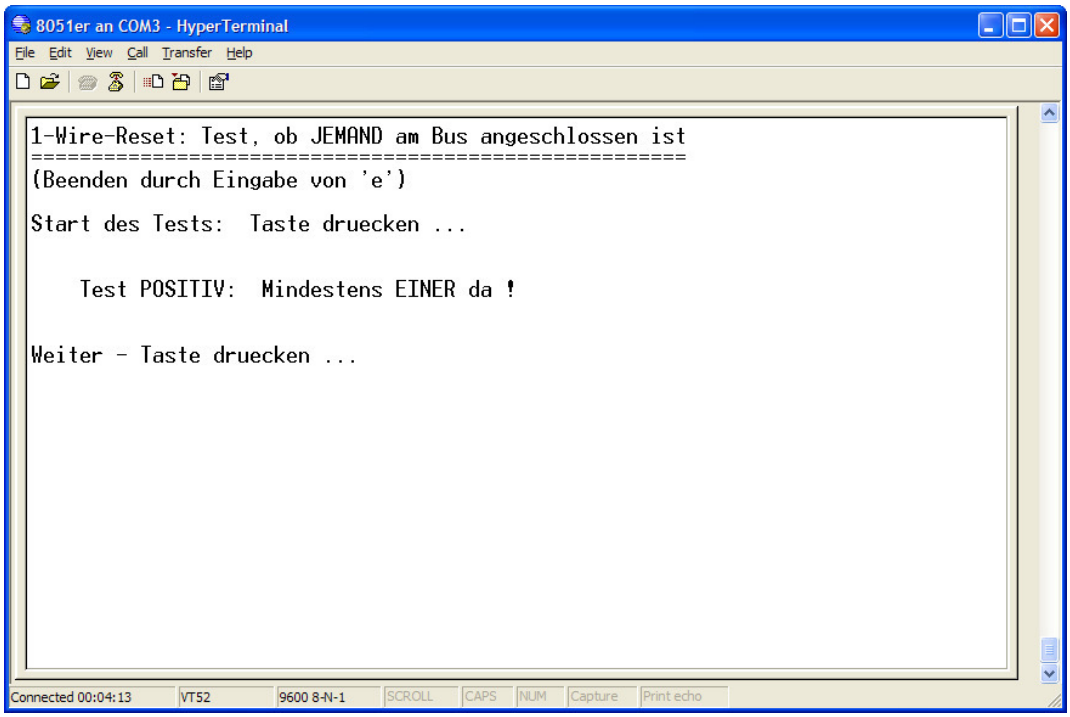


Abb.6.3.3: Mindestens ein Teilnehmer ist am 1-Wire-Bus angeschlossen

2. Single Chip: Lesen des 64 Bit ROM-Identifiers

VOR Auswahl dieses Menü-Punktes müssen Sie unbedingt sicherstellen, dass nur genau EIN 1-Wire-Slave angeschlossen ist, denn dessen ROM-Identifier wird hier ausgelesen, **Abb.6.3.4** (der 1-Wire ´Read ROM´-Befehl funktioniert ja nur bei einem einzigen Slave am Bus):

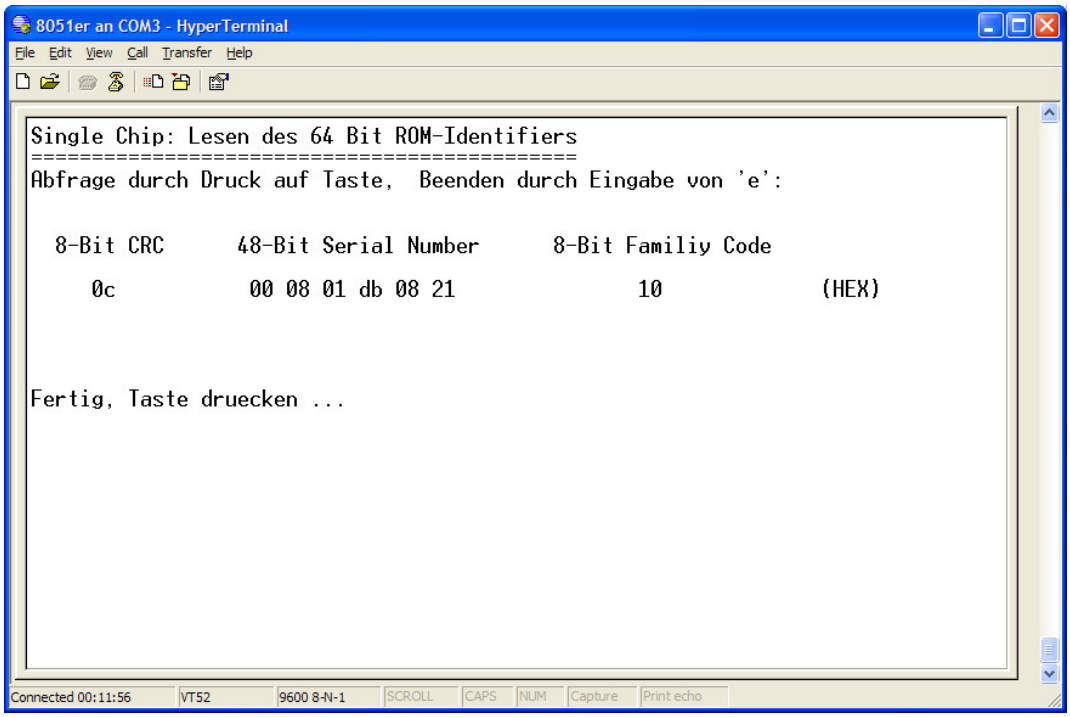
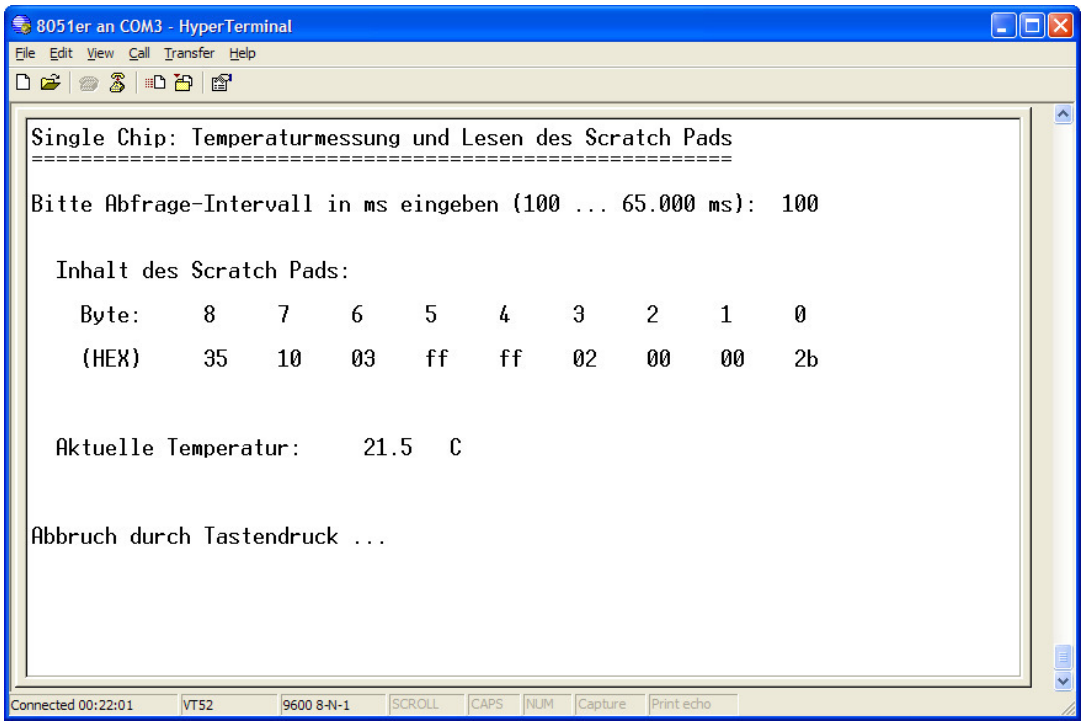


Abb.6.3.4: Das Auslesen des ROM-Identifiers

3. Demo - Single Chip: Temperaturmessung und Lesen des Scratch Pads

In diesem Punkt startet nun die erste Demo-Messung mit dem DS18S20er.
Aber auch hier darf NUR genau EIN DS18S20er am Bus angeschlossen sein, da wir, der Einfachheit halber, mit der ´Skip ROM´-Anweisung arbeiten, **Abb.6.3.5**:



6.3.5: Das Abfragen eines einzigen DS18S20ers am Bus

4. Single Chip: 2 Byte (TH,TL) in Scratch Pad schreiben

Mit diesem Programm-Punkt können Sie die beiden beschreibbaren SRAM/EEPROM-Speicherstellen im Scratch Pad-Bereich des DS18S20ers mit beliebigen 8-Bit-Werten beschreiben, **Abb.6.3.6**:

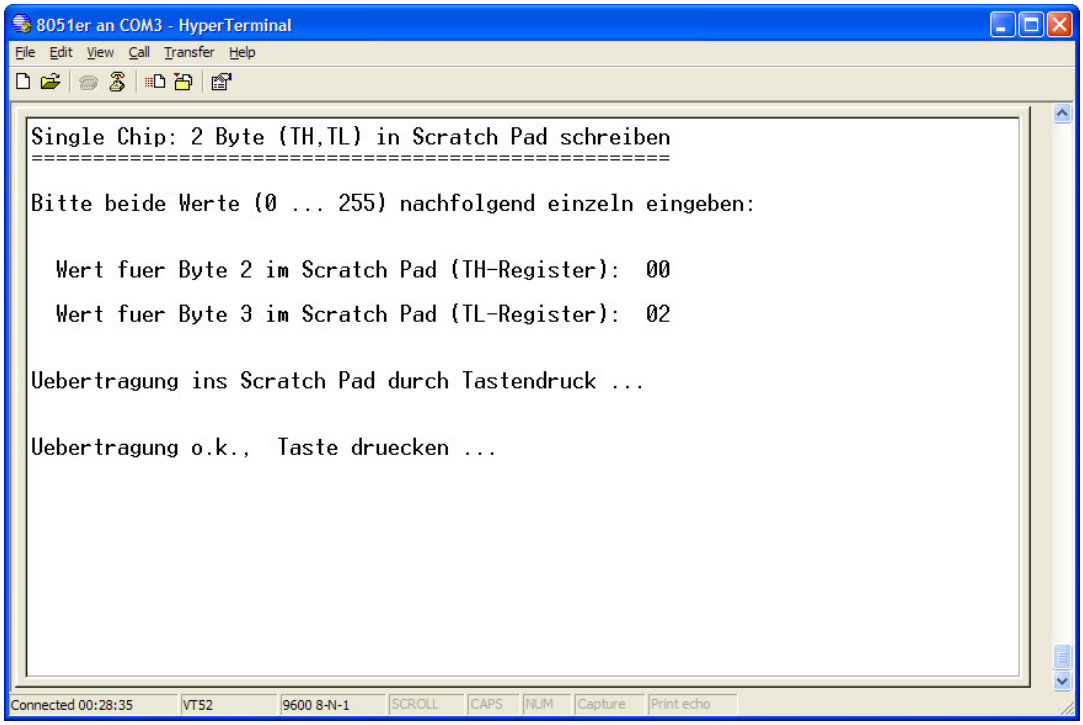


Abb.6.3.6: Das Beschreiben der TH- und TL-Register

Auch hier darf wieder nur ein einziger DS18S20er am Bus angeschlossen sein, da der ´Skip ROM´-Befehl zum Einsatz kommt.
Unter dem Menü-Punkt 3) kann dann überprüft werden, ob die Einstellung korrekt erfolgte, denn hier wird ja immer der gesamte Inhalt des Scratch Pads ausgelesen und dargestellt.

5. Demo - Bus-Betrieb mit max. 5 Sensoren

Und nun „geht´s richtig zur Sache“: in diesem Menü-Punkt läuft eine Demo mit bis zu fünf DS18S20ern am Bus ab, **Abb.6.3.7**:

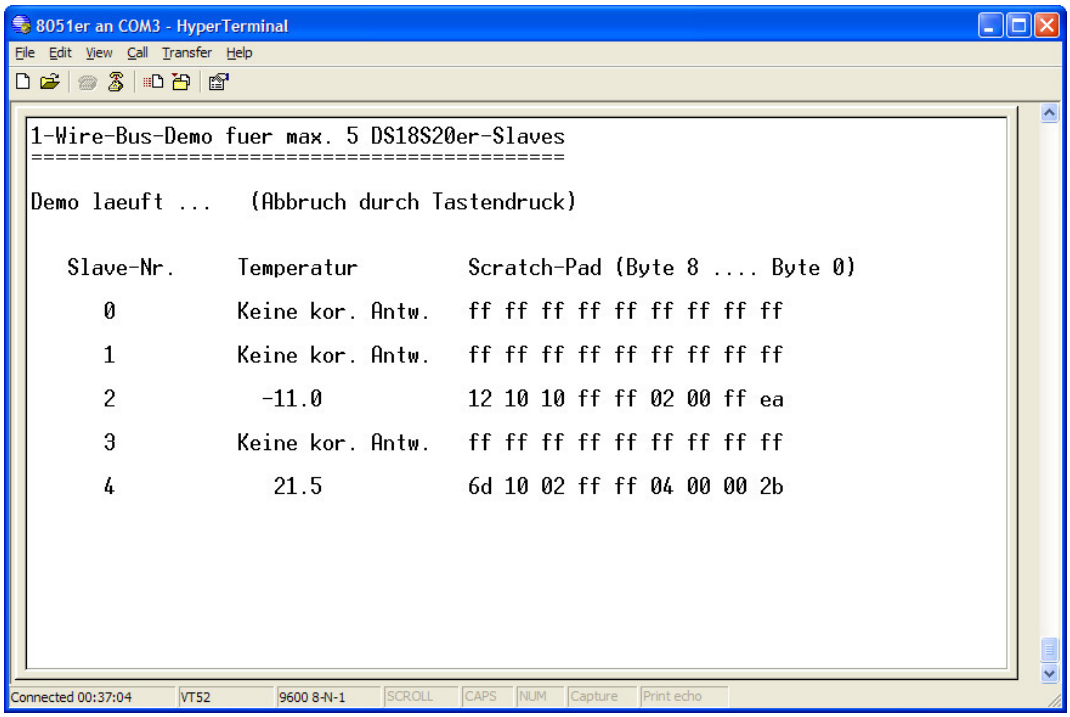


Abb.6.3.7: Demo-Betrieb mit bis zu fünf DS18S20ern

Die ROM-Identifier der einzelnen DS18S20er werden dabei in das globale Array ´ds1820_mat[.] [.]´ eingetragen, so wie in Kapitel 6.2 beschrieben.

Unsere Demo läuft hier allerdings nur mit zwei DS18S20ern, da wir nicht mehr zur Verfügung hatten, aber es klappt auch mit bis zu fünf Stück am Bus !

9. Test zur uC-spezifischen Delay-Festlegung

Im letzten Menü-Punkt befindet sich eine kleine Hilfs-Routine, um die notwendigen Zeitverzögerungen, in Abhängigkeit vom Mikrocontroller-Typ und von der Taktfrequenz zu bestimmen.
Dazu klemmen Sie alle DS18S20er vom Bus ab und schließen ein Oszilloskop am 1-Wire-Port-Pin des Mikrocontrollers an, **Abb.6.3.8**:

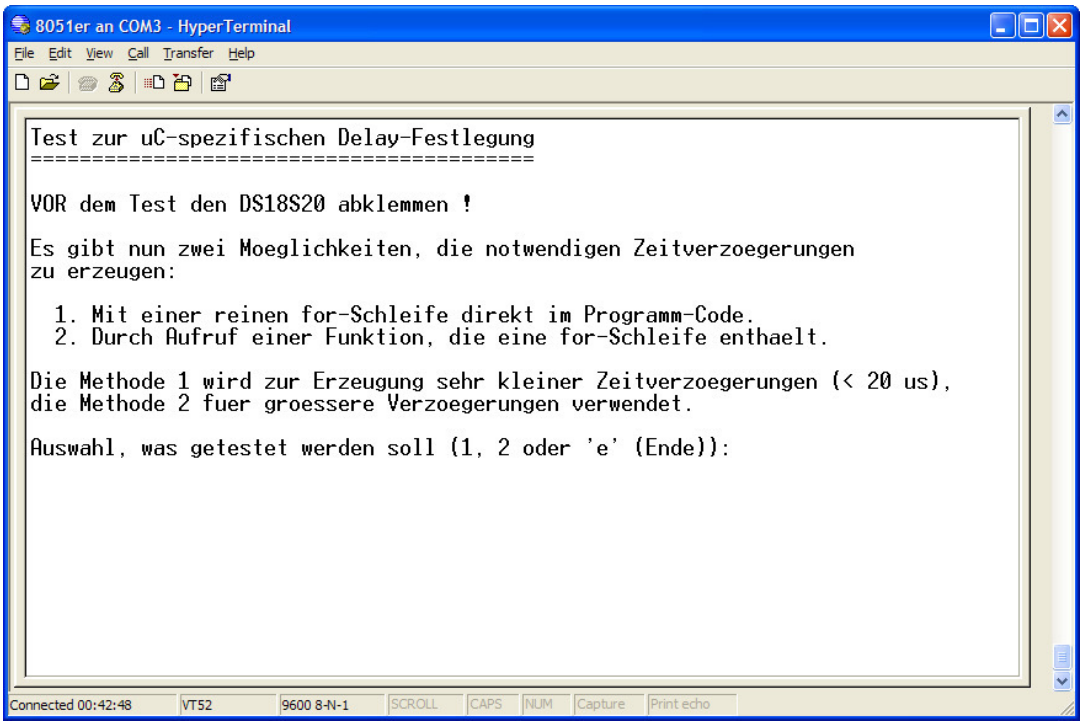


Abb.6.3.8: Ein einfaches Hilfsmittel zum „Zeit-Tuning“

So können Sie jetzt die Angaben in den Tabellen Tab.2.4.3.1 und Tab.2.4.3.2 nachvollziehen und mit solchen Werten ihr eigenes System zeitmäßig optimal auf den 1-Wire-Bus einstellen.

Und nun stehen Ihnen „alle Türen“ offen: da wir sämtliche Quell-Codes frei zugänglich machen, haben Sie alle Möglichkeiten, selber weiter zu experimentieren, selber Ihre eigene Software zu entwerfen und Ihr 1-Wire-Bussystem weiter auszubauen.

Ausblick …

Die im Inhaltsverzeichnis erwähnten und hier noch fehlenden 1-Wire-Bausteine ***DS2450*** (4-fach A/D-Wandler) und ***DS2408*** (8-fach digitale I/O-Port-Erweiterung) werden wir selbstverständlich auch ausführlich beschreiben und in Betrieb nehmen (in Verbindung mit dem PT-1-Wire-ExBo), denn das war hier erst nur der Anfang und dieses 1-Wire-Projekt wird ja noch weiter fortgesetzt ….. !

10. Literatur

[0] 1-Wire-Homepage bei MAXIM/DALLAS:
http://www.maxim-ic.com/products/1-wire

[1] MAXIM/DALLAS Application Note 162:
INTERFACING THE DS18X20/DS1822 1-WIRE TEMPERATURE SENSOR IN A MICRO-CONTROLLER ENVIRONMENT

[2] MAXIM/DALLAS On Line Tutorial zum 1-Wire-Bus: s. [0]

[3] IDE µC/51 für 8051er-Mikrocontroller:
www.wickenhaeuser.de

[4] MAXIM/DALLAS Datenblatt zum DS18S20: s. [0]

[5] MAXIM/DALLAS Application Note 187:
1-Wire Search Algorithm

[6] MAXIM/DALLAS Application Note 148:
Guidelines for Reliable Long Line 1-Wire® Networks

[7] Datenblatt von Hygrosens Instruments GmbH:
DRUCKFESTER TEMPERATURFÜHLER DS 1820 MIT GEWINDE M10
www.hygrosens.com

[8] MAXIM/DALLAS Application Note 4206:
Choosing the Right 1-Wire® Master for Embedded Applications

11. Version History

Version 1.0

Veröffentlicht am 08.03.2010.
Teil 1 des Projektes: Kapitel 1 bis Kapitel 3.

Version 1.1

Veröffentlicht am 18.03.2010.
Teil 1 und Teil 2 des Projektes: Kapitel 1 bis Kapitel 3, Kapitel 4, Kapitel 6.