

*Bernd vom Berg
Peter Groppe*

Projekt Nr. 275:

„Am langen Draht der 1-Wire-Bus“

Grundlagen und Anwendungen des 1-Wire-Busses

(Stand: 07.06.2010, V1.2)



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.

Inhalt

(Stand: 07.06.2010)

- 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
 - 2.4.6 Der 8-Bit-CRC-Check (ROM-Identifizier)
 - 2.5 Weitere 1-Wire-Eigenschaften
- 3. Die 1-Wire-Funktionen in 'C' für 8051er**
- 4. Die verfügbaren 1-Wire-Komponenten**
- 5. Das PT-1-Wire-Experimentalboard (PT-1-Wire-ExBo)**
- 6. Der DS18S20 – Digitaler Temperatur-Sensor**
 - 6.1 Funktion
 - 6.2 Hardware
 - 6.3 Die DS18S20er-API
 - 6.4 Das DS18S20er-Demo-Programm
- 7. Der DS2450 – 4-fach A/D-Wandler**
 - 7.1 Funktion
 - 7.2 Ablauf der Kommunikation
 - 7.3 Der 16-Bit-CRC-Check
 - 7.4 Die DS2450er-API
 - 7.5 Hardware
 - 7.6 Das DS2450er-Demo-Programm

8.	Der DS2408 – 8-fach digitale I/O-Porterweiterung
8.1	Funktion
8.2	Applikation I: Relais, LEDs, Taster, Summer
8.2.1	Hardware
8.2.2	Die DS2408er-API - I/O
8.2.3	Das DS2408er-Demo-Programm - I/O
8.3	Applikation II: Betrieb eines LC-Displays
8.3.1	Hardware
8.3.2	Die DS2408er-API - LCD
8.3.3	Das DS2408er-Demo-Programm - LCD
9.	Die große Gesamt-Demo
10.	Die Projekt-Software
10.1	Die allgemeinen APIs
10.1.1	cc03er.c/.h
10.1.2	terminal.c/.h
10.2	Die 1-Wire-APIs
10.2.1	ow.c/.h
10.2.2	ow_slaves.h
10.2.3	ds1820.c/.h - ALT
10.2.4	ds1820.c/.h - NEU
10.2.5	ds2450.c/.h
10.2.6	ds2408.c/.h - I/O
10.2.7	ds2408.c/.h - LCD
10.3	Die Demo-Programme
10.3.1	DS18S20 - ALT
10.3.2	DS18S20 - NEU
10.3.3	DS2450
10.3.4	DS2408 - I/O
10.3.5	DS2408 - LCD
11.	Bezugsquellen
12.	Literatur
13.	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 I²C-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 I²C-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 I²C-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 I²C-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 I²C-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 I²C- und den CAN-Bus.

3. Die Busausdehnung

Bei den Betrachtungen zur Busausdehnung lassen sich ganz grob zwei Gruppen bilden: SPI- und I²C-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-, I²C- 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 I²C-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 I²C-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 I²C- 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 I²C-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 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.
- **Seriell, 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-Identifizier** genannt. Da sich zwei Slaves der gleichen Art (z.B. zwei Temperatursensoren) in 56 dieser 64 Bits unterscheiden, können „im gesamten bekannten Universum“ insgesamt

$$2^{56} = 72.057.594.037.927.936$$

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:**

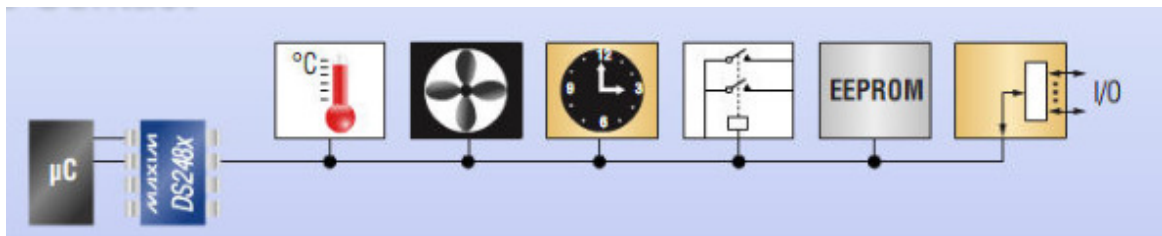


Abb. 2.1.1: Die Vielzahl der 1-Wire-Slave-Funktionen, [2]

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 k Ω -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 I²C-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 k Ω an die Betriebsspannung (+5 V oder +3 V) gelegt wird, **Abb.2.3.1.1**:

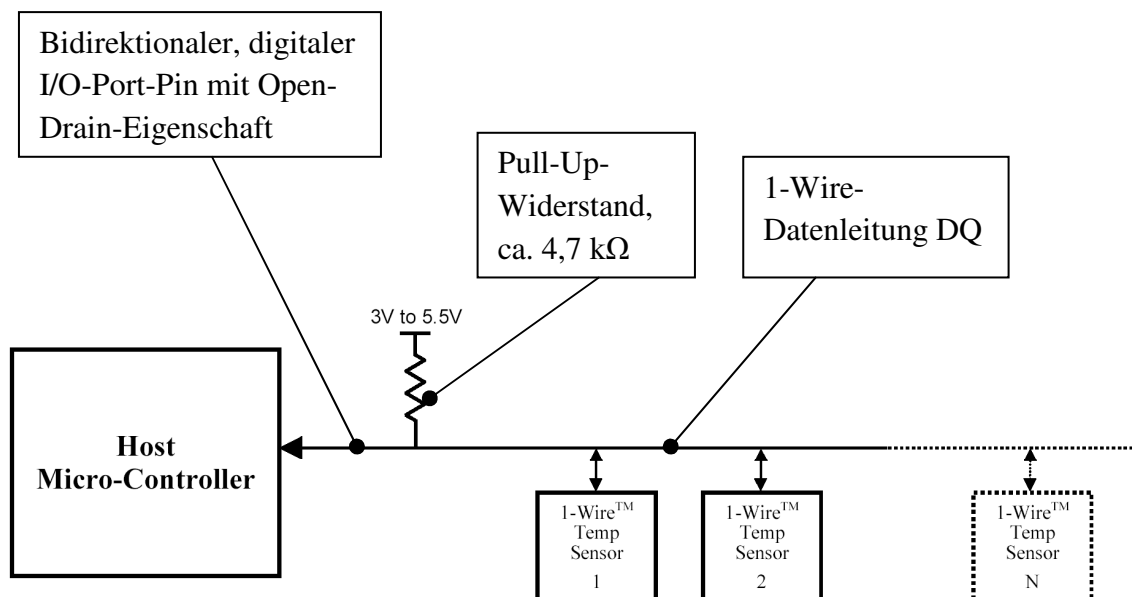


Abb.2.3.1.1: Die einfachste 1-Wire-Busankopplung, [1]

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:**

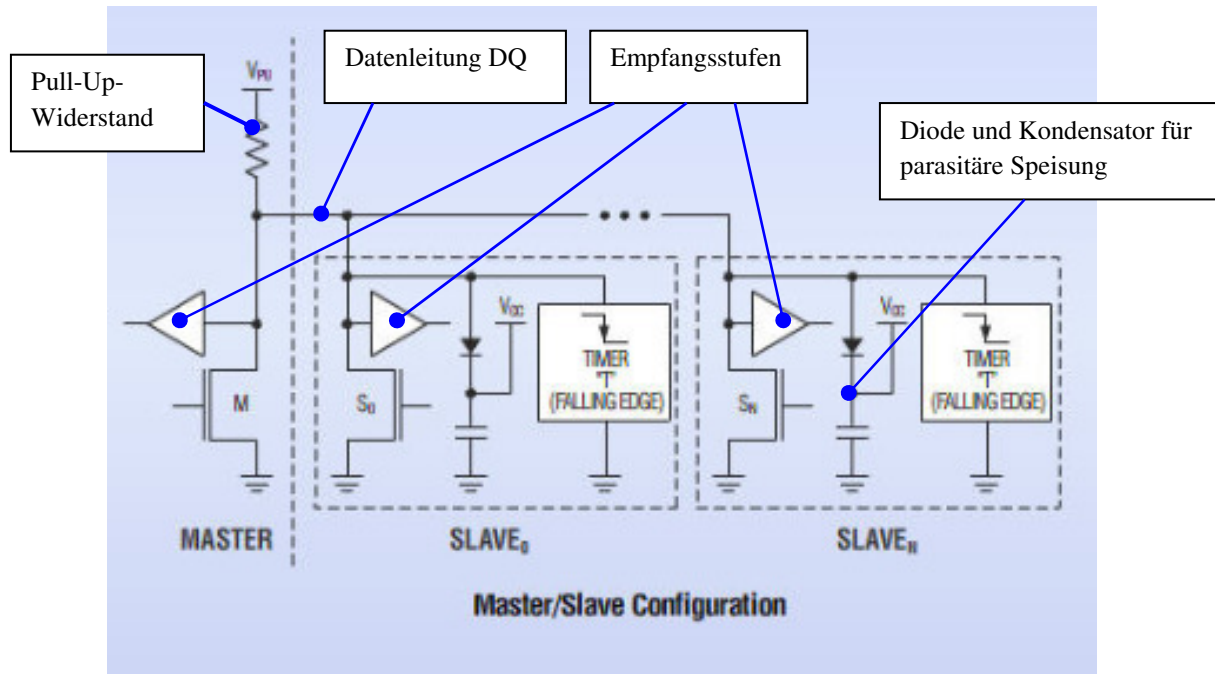


Abb.2.3.1.2: Interna, [2]

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 S₀ ... S_N** im Slave durchgeschaltet.

Soll nun eine log. '1' gesendet werden (\equiv High-Pegel auf DQ \equiv 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 Busanwendungen zu.

Das ist somit die zweite Besonderheit beim 1-Wire-Bus und diese soll nachfolgend kurz einmal erklärt werden, **Abb.2.3.2.1**:

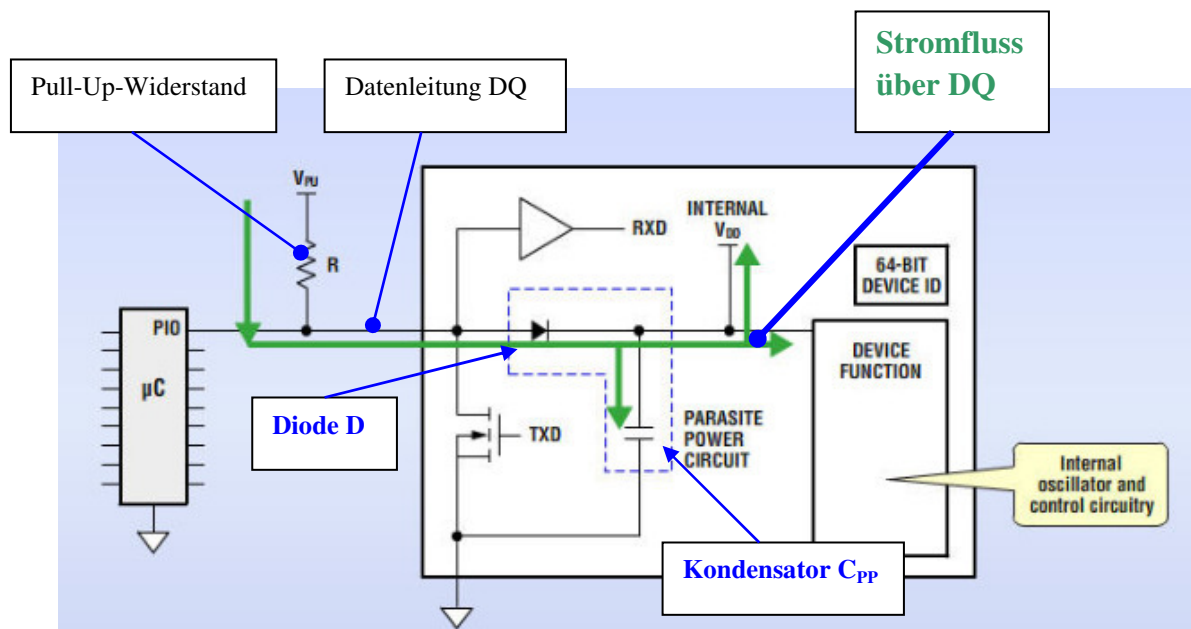


Abb.2.3.2.1: Die parasitäre Speisung von 1-Wire-Slaves über die Datenleitung DQ, [2]

Während des High-Pegels auf DQ (\equiv 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 \equiv 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 Temperatursensor, 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 k Ω 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:**

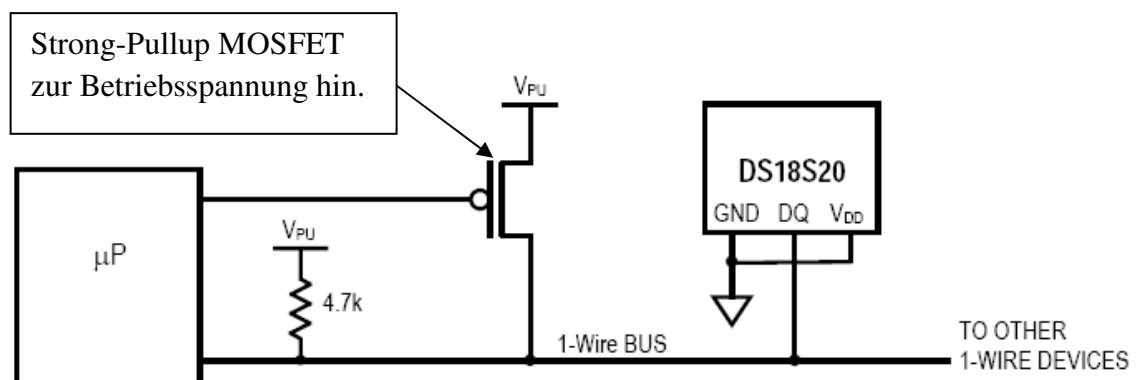


Abb.2.3.2.2: Der Strong-Pullup lässt richtig Strom fließen, [4]

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 Bausteins, 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 \equiv Strong Pullup.
- Versorgung des Chips durch eine externe Spannungsversorgung \equiv 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-Kommunikationssoftware für den Mikrocontroller wird selber geschrieben (einfaches Bit-Banging über diesen Port-Pin).
- Einsatz eines normalen **4,7 k Ω -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 zu 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**:

Anwendungsspezifische Funktionen

Funktionen, die zur Realisierung der ganz konkreten Anwendung mit den 1-Wire-Slaves dienen, also z.B.:

- Temperaturmesswert verarbeiten und darstellen
- Analogmesswerte verarbeiten und abspeichern
- Relais und LEDs ansteuern
- etc.

Slave-spezifische Funktionen (kann nur der jeweilige 1-Wire-Slave ausführen, z.B.):

- Temperatur auslesen aus einem 1-Wire Temperatur-Chip
- Messwert auslesen aus einem 1-Wire-A/D-Wandler-Chip
- Port setzen bei einem 1-Wire-Digital-I/O-Port-Chip
- etc.

Höhere Funktionen (ROM Commands) (Grundbefehle, die jeder 1-Wire-Slave ausführen können muß)

- Read ROM
 - Match ROM
 - Skip ROM
 - Search ROM
- Master-seitig: CRC-Check

Basisfunktionen (muß jeder 1-Wire-Slave beherrschen)

- Erzeugung von Wartezeiten
- Einzel-Bit senden
- Einzel-Bit empfangen
- Master Reset / Slave Presence
- Ein Byte senden
- Ein Byte empfangen

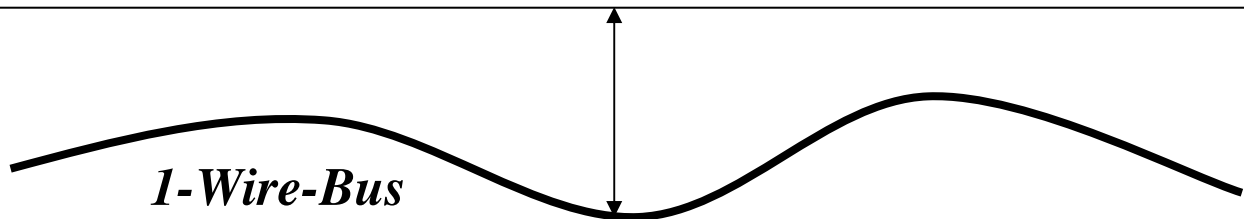


Abb.2.4.1.1: Die 1-Wire-Kommunikationspyramide (Master-seitig)

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 μ s-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 zu 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:

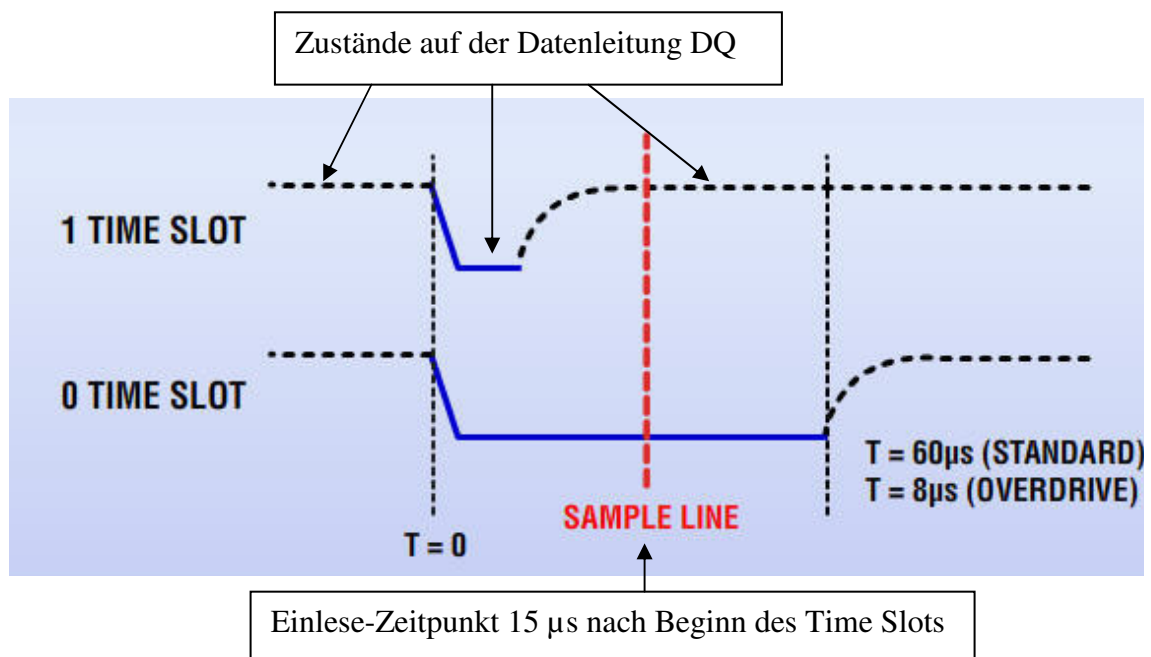


Abb.2.4.2.1: Der 1-Wire Time Slot und die Übertragung der Bits (Master sendet), [2]

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 brauchen 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 (\equiv **1 Time Slot**).

Hält der Master DQ aber bis zum Ende des Time Slots auf Low, so lesen die Slaves eine log. '0' ein (\equiv **0 Time Slot**).

Vereinfacht bedeutet das also:

- Transfer einer log. '1' \equiv kurzer Low-Impuls auf DQ
- Transfer einer log. '0' \equiv 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 Datentransfargeschwindigkeit 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.

Programmierer 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-Kommunikationslaufes 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 (\equiv **Device Selection-Befehl**) , denn jeder Slave besitzt intern eine eigene, individuelle, unverwechselbare Stationsnummer, die der Hersteller dem Slave fest und unveränderbar mitgegeben hat (\equiv **ROM Code**, s. Kap.2.1).
Nachdem der gewünschte Slave so selektiert worden ist, kann der Master den eigentlichen Befehl an den Slave übermitteln \equiv **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 (\equiv 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 $\mu\text{C}/51$** 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 μ s bis ca. 700 μ s (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 benötigte 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 Breite 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 eingesetzt und das Programm dann so kompiliert !), **Tab.2.4.3.1:**

wert t	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:

$$\text{zeitverzögerung} = (\text{wert}-1) * 2,2 + 4,2 \quad \text{in } \mu\text{s}$$

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**:

de- lay	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:

$$\text{zeitverzögerung} = (\text{delay} * 6,55) + 19,4 \quad \text{in } \mu\text{s}$$

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):

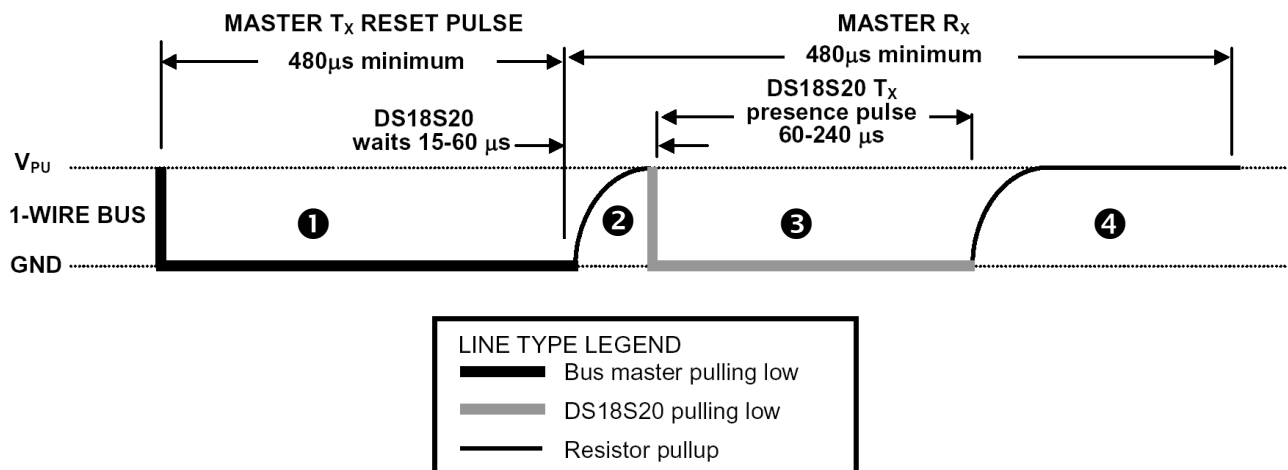


Abb.2.4.3.1: Der Master Reset- und der Slave Presence-Impuls, [4]

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

Zustand ①:

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 \equiv Startzeitpunkt für den Ablauf eines Kommunikationszyklusses (s. Abb.2.3.1.2, Baugruppe „**TIMER “T“ (FALLING EDGE)**“).

Zustand ②:

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 (\equiv Anfang von ②), warten die Slaves, um ihre Anwesenheit kund zu tun, um danach also ihren Presence-Low-Impuls auf DQ auszugeben.

Zustand ③:

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 ④:

Nach Ablauf von insgesamt mindestens 480 μs (ab der steigenden Flanke auf DQ \equiv Anfang von Zustand ②) 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 Abschnittes ❸ einen High-Pegel ein, so weiß 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 Abschnittes ❸ aber dagegen einen Low-Pegel ein, so weiß 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 Slave 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 beschriebenen 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 (\equiv Zustand ④).

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:

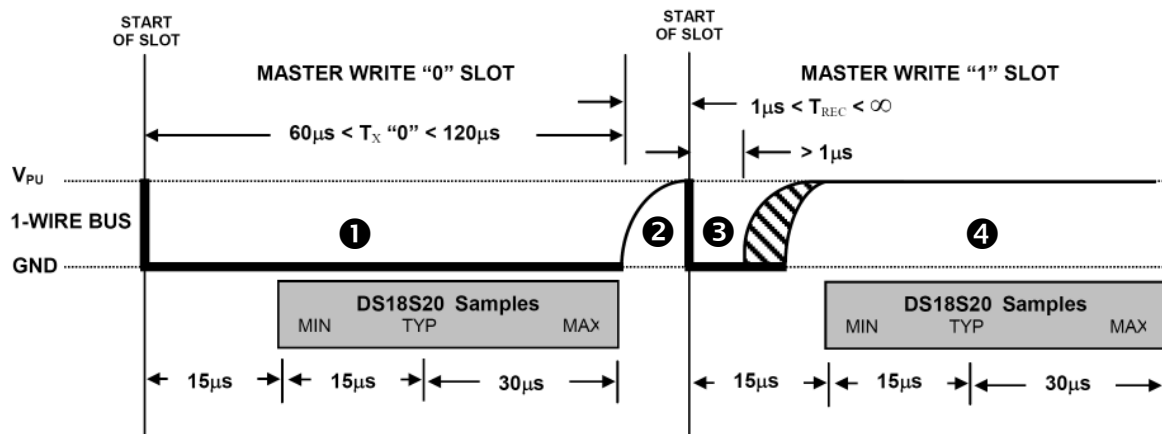


Abb.2.4.3.2: Das Schreiben von Bits, [4]

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 ①:

Der Master setzt DQ auf log. '0' \equiv Start des Time Slots = Start der Zeitsynchronisation aller Zeitgeber 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 ②:

Danach setzt der Master DQ wieder auf log. '1' und wartet mindestens 1 μ s bevor das nächste Bit geschrieben wird bzw. bevor der nächste Time Slot angefangen wird.

Die Slaves tasten nun (frühestens) 15 μ s nach Beginn des Time Slots (also nach Beginn der fallenden Flanke von Zustand ①) die Leitung DQ ab (\equiv **DS18S20 Samples - MIN**) und finden dort eine log. '0', die sie dann einlesen.

Der maximale (ungünstigste) Sample-Zeitpunkt bei den Slave liegt bei 60 μ s nach Beginn des Time Slots (\equiv **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 ③:**

Der Master setzt DQ auf log. '0' \equiv Start des Time Slots.

Nun wird DQ aber **sofort** (frühestens nach Ablauf von 1 μ s, spätestens aber nach 15 μ s) wieder auf log. '1' gesetzt.

Zustand ④:

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

Die Slaves tasten nun wiederum (frühestens) 15 μ s nach Beginn des Time Slots (also nach Beginn der fallenden Flanke von Zustand ③) die Leitung DQ ab (\equiv **DS18S20 Samples - MIN**) und finden dort eine log. '1', die sie dann einlesen.

Zusammen gefasst lässt sich also sagen:

- Master überträgt eine log. '0' \equiv langer Low-Impuls im Time Slot auf DQ
- Master überträgt eine log. '1' \equiv (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 hintereinander 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:

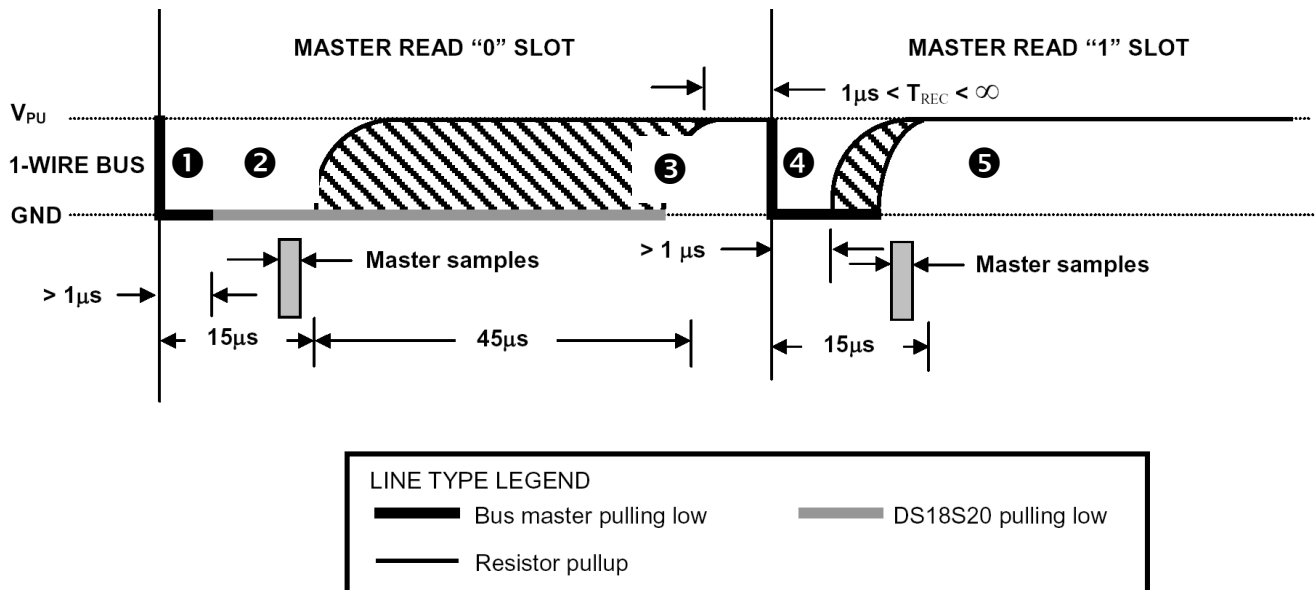


Abb.2.4.3.3: Das Einlesen von Bits, [4]

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 ①:

Der Master setzt dazu DQ auf log. '0' \equiv Start des Time Slots = Start der Zeitsynchronisation aller Zeitgeber in allen Slaves und im Master.

Möglichst schnell, nach $1\mu s$ bis maximal $15\mu s$, 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 ②:

Der Slave zieht nun seinerseits DQ auf Low und hält diesen Pegel auf DQ bis zum Ende des Time Slots, also $60 \dots 120\mu s$ lang (in der Abb.2.4.3.3 ist der Time Slot $60\mu s$ lang, aber die 1-Wire-Norm lässt ja im ungünstigsten Fall eine Länge von bis zu $120\mu s$ für einen Time Slot zu).

Spätestens $15\mu s$ nach Beginn des Time Slots (nach der fallenden Flanke am Anfang von Zustand ①) liest nun der Master den Zustand von DQ ein (\equiv **Master samples**) und erkennt auf DQ einen Low-Pegel, also eine log. '0'.

Zustand ③:

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 ④:**

Der Master setzt dazu DQ auf log. '0' ≡ Start des Time Slots = Start der Zeitsynchronisation aller Zeitähler 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 ⑤:

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 ④) 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 μ s 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 hintereinander 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 dem 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 Befehl 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-Bausteinen 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 „eingeschnitten“ 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 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 (\equiv 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` abgespeichert 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 (\equiv 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 \equiv 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 erhebliche 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-Identifizieren 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 dem Einschalten, erst einmal seinen Bus „untersuchen“ kann, um festzustellen, welche Slaves bzw. welche Slave-Adressen (ROM-Identifizierer) überhaupt angeschlossen sind.

Mit Hilfe eines bestimmten Algorithmus fragt er die Slaves ab und erhält als Ergebnis eine Liste mit den zugehörigen ROM-Codes.

Das 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 jedem 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 beschriebenen 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 $\mu C/51$) 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.

Andererseits 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-Einführungstutorial (in englischer Sprache) und eine Übersicht über die aktuell verfügbaren 1-Wire-Produkte, **Abb.4.1**:

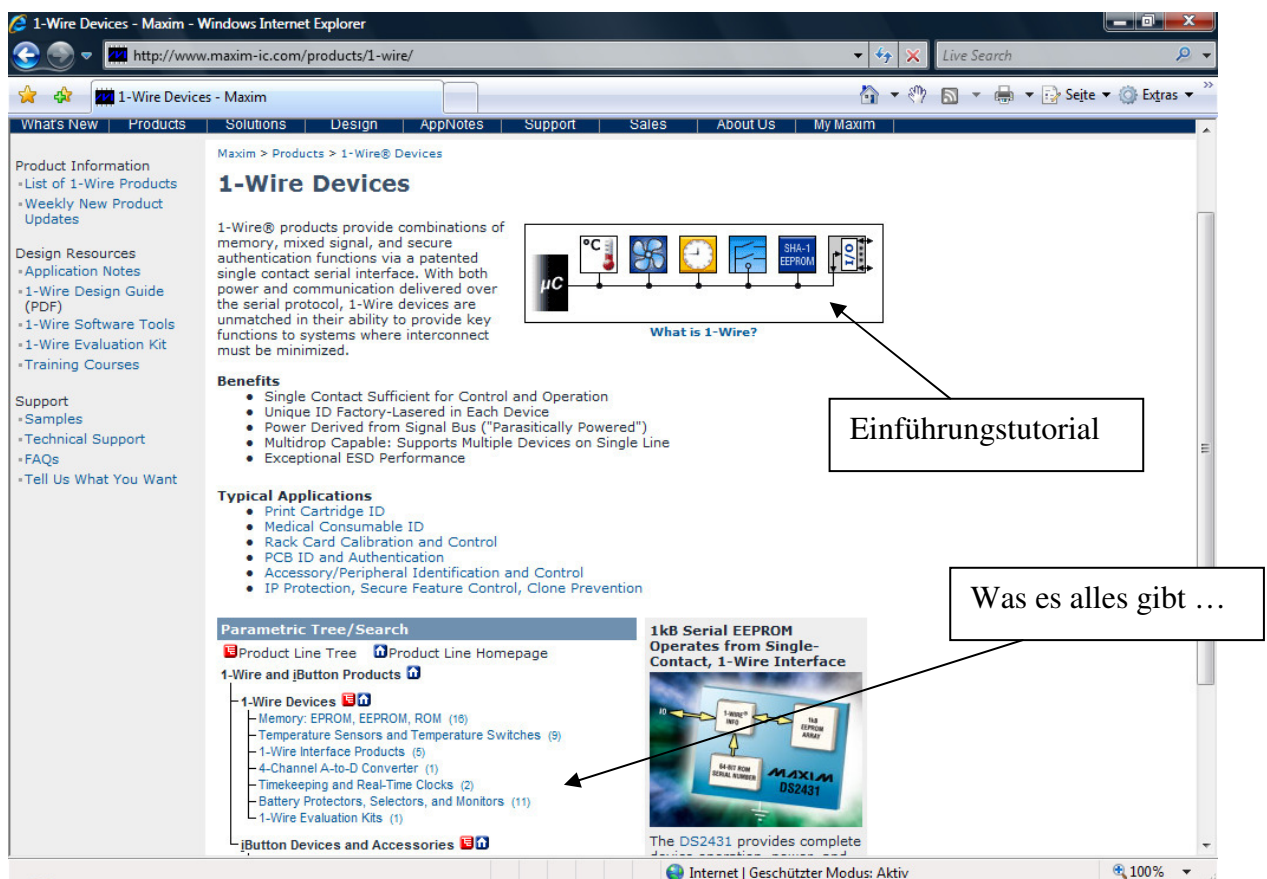


Abb.4.1: Die Welt der 1-Wire-Produkte

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 „reinsten“ 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-Komponenten, 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):

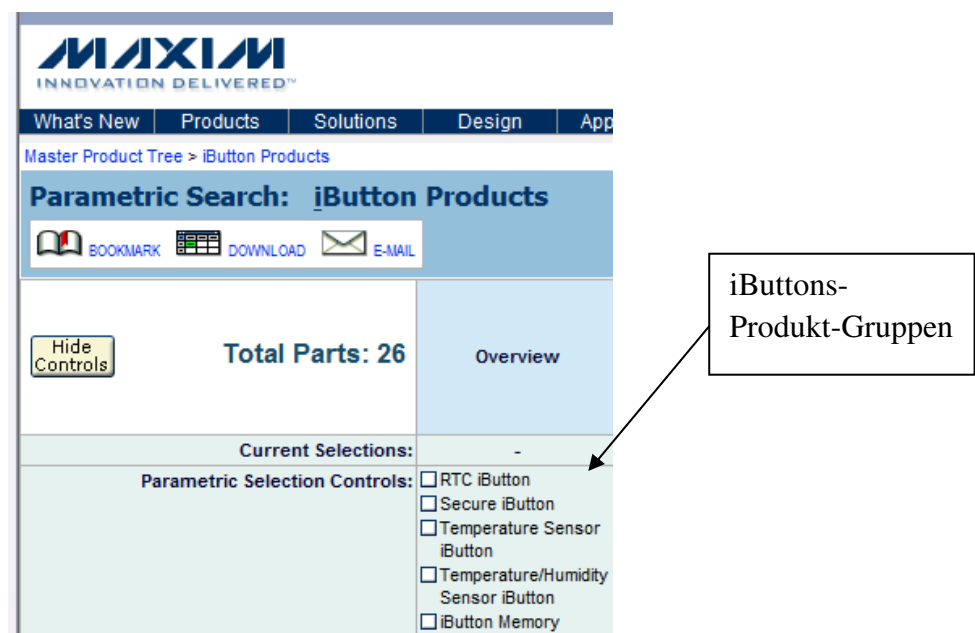


Abb.4.5: Die Familie der iButtons

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>)

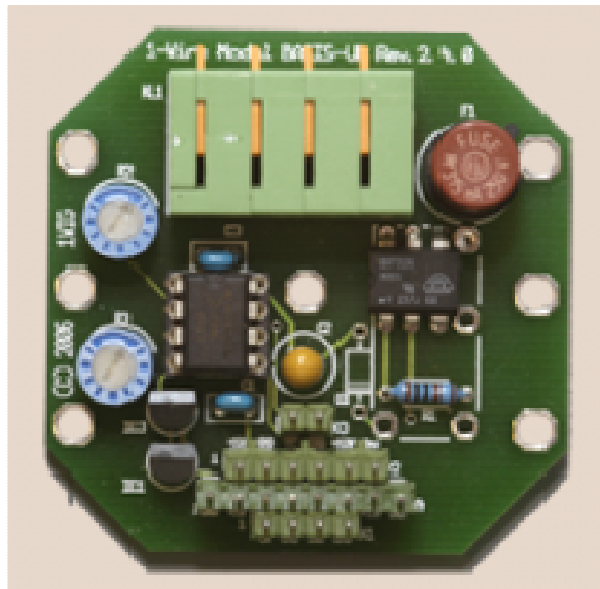


Abb.4.7: Komplettes 1-Wire-Sensor-/Aktor-Modul

(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)

Da sich das PT-1-Wire-Experimentalboard 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 betreiben.

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: $-55^{\circ}\text{C} \dots +125^{\circ}\text{C}$, Auflösung: $0,5^{\circ}\text{C}$, Genauigkeit $\pm 0,5^{\circ}\text{C}$ im Messbereich von $-10^{\circ}\text{C} \dots +85^{\circ}\text{C}$.
- 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].

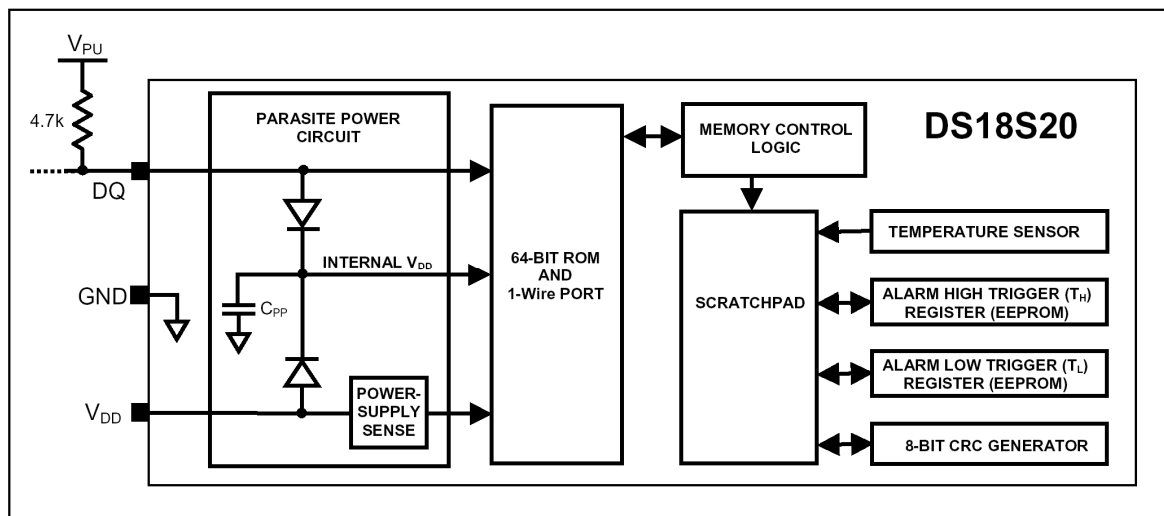


Abb.6.1.2: Das Blockschaltbild des DS18S20ers

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

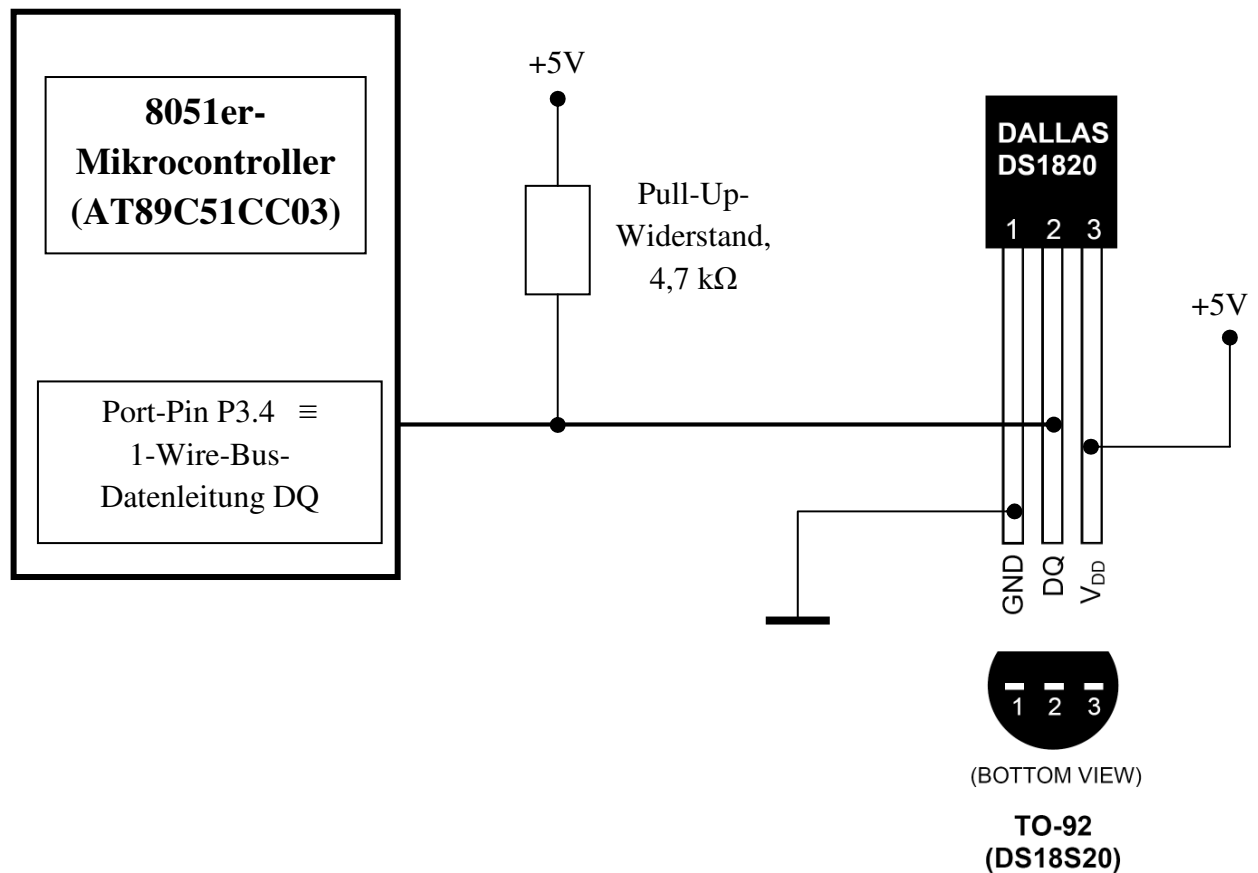


Abb.6.1.4: Der DS18S20er als Einzelchip angeschlossen (TO-92er Gehäuse)

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:**

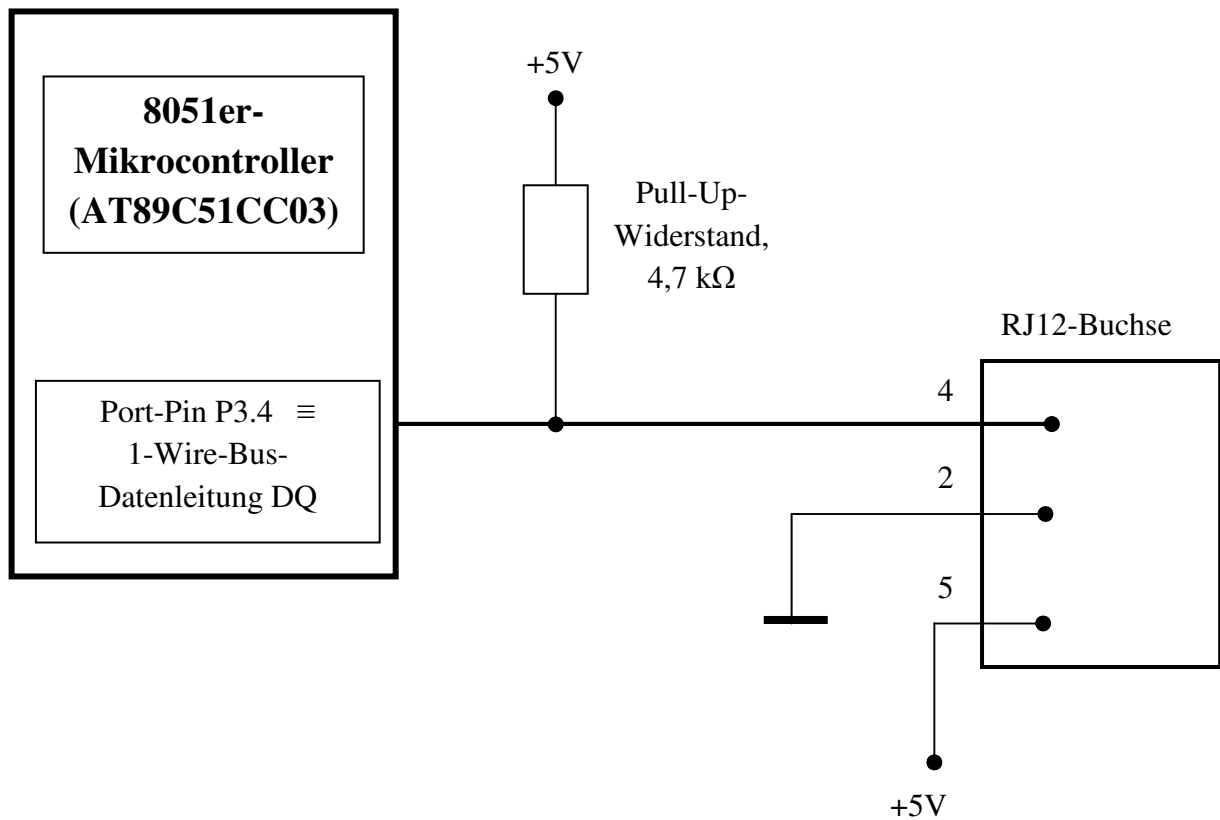


Abb.6.1.5: Anschluss eines eingebauten DS18S20ers über RJ12-Stecker/Buchse

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**:

8-BIT CRC		48-BIT SERIAL NUMBER				8-BIT FAMILY CODE (10h)	
MSB	LSB	MSB		LSB		MSB	LSB

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 Befehlssequenz, 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 ausgegeben
printf("%02x %02x %02x %02x %02x %02x %02x %02x\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-Identifizier 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 DS18S20er misst nun die Temperatur und legt diese in den Scratch Pad-Registern ab.
- Master muss nun noch mindestens 750 ms warten (\equiv 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);

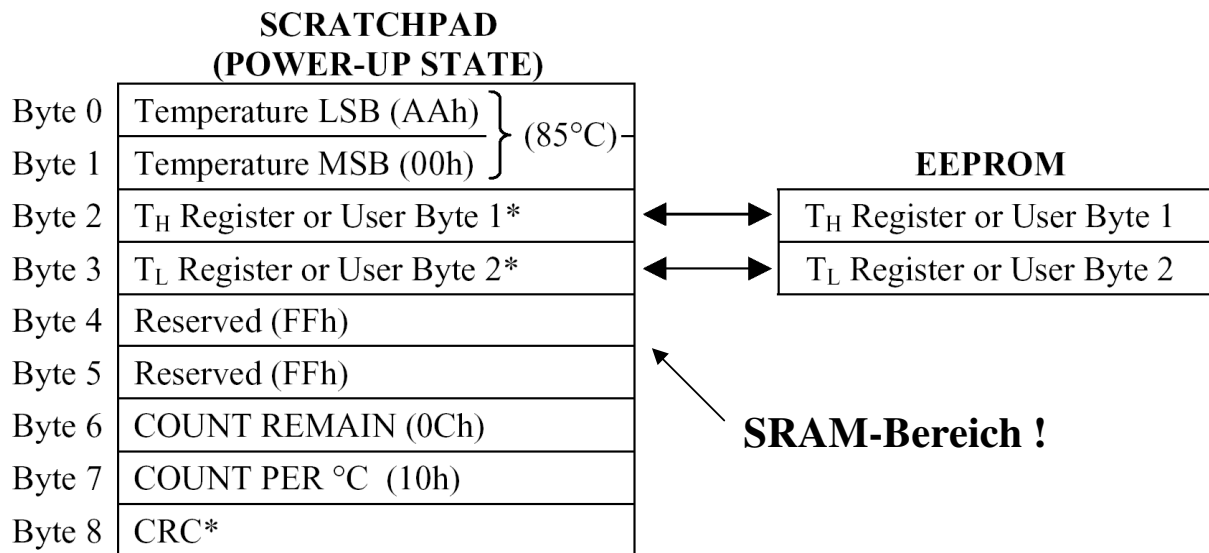
// Antwort einlesen: 9 Byte große Scratch Pad-Inhalt einlesen
for (i=0; i<9; i++)
{
    ds1820[i] = ow_rd_byte();
}
```

}

(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**:



**Power-up state depends on value(s) stored in EEPROM.*

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 mindestens 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, dass 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 ($\equiv 8 \text{ Byte} \equiv 64 \text{ Bit}$ 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 DS18S20er misst nun die Temperatur und legt diese in den Scratch Pad-Registern ab.
- Master muss nun noch mindestens 750 ms warten (\equiv 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}, ← ROM-Code von Station-Nr. 0
  {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. 4
};
```

Wir haben hier ein Feld aus unsigned char-Variablen (\equiv 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 Beispiel 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) \equiv immer den Wert 00h
- T_L (Byte 3) \equiv 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

($\equiv 00h \dots feh \equiv 0 \dots 254 \equiv$ 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 $\mu\text{C}/51$ beim Anlegen eines 'C'-Projektes mittels MakeWiz diese beiden 'C'-Files mit ihren Funktionen eingebunden, **Abb.6.3.1:**

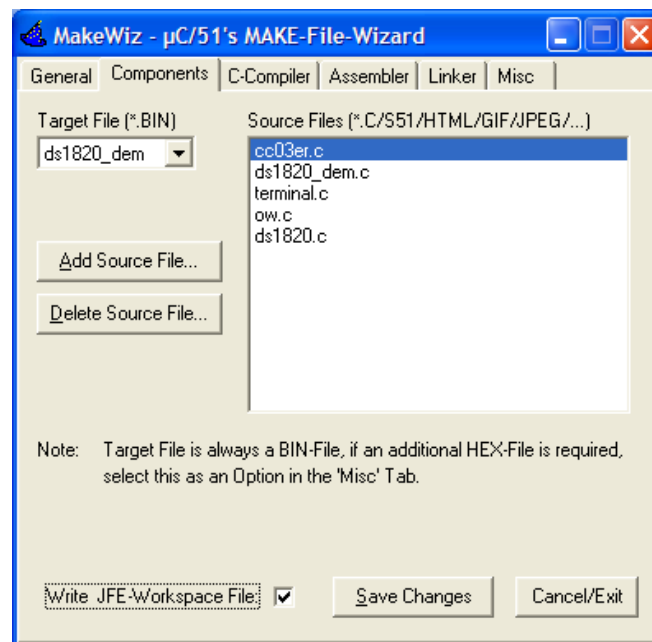


Abb.6.3.1. Das Einfügen der 'C'-Funktions-Files in das Projekt mit MakeWiz ($\mu\text{C}/51$)

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**:

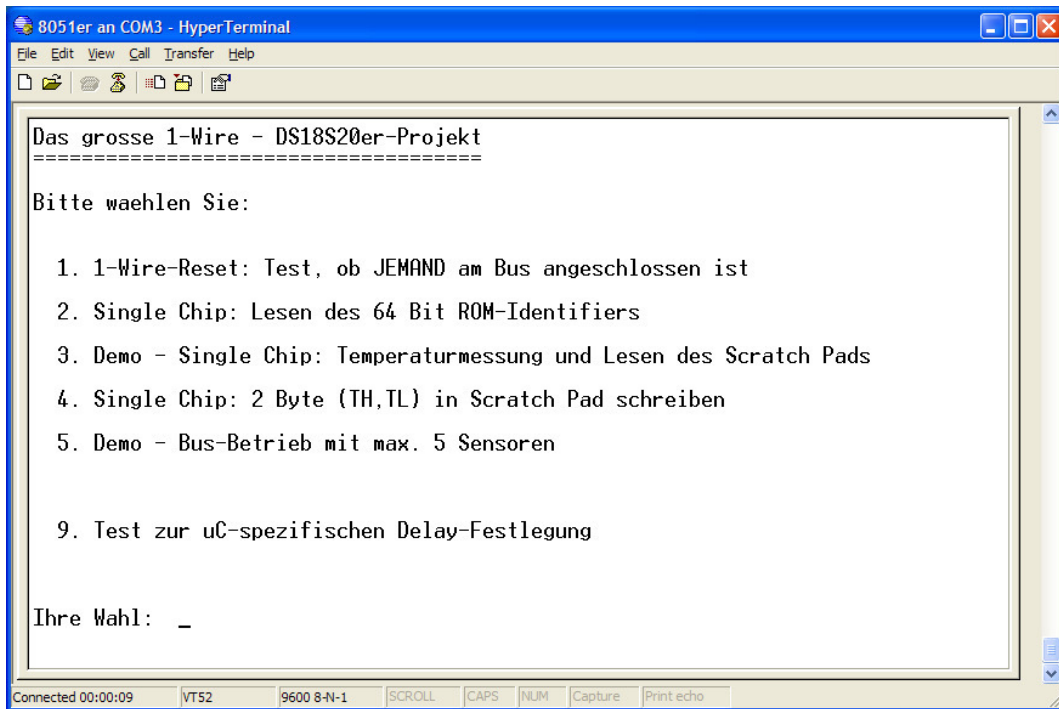


Abb.6.3.2: Das Auswahlmenü des DS18S20er-Demo-Programms

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**:

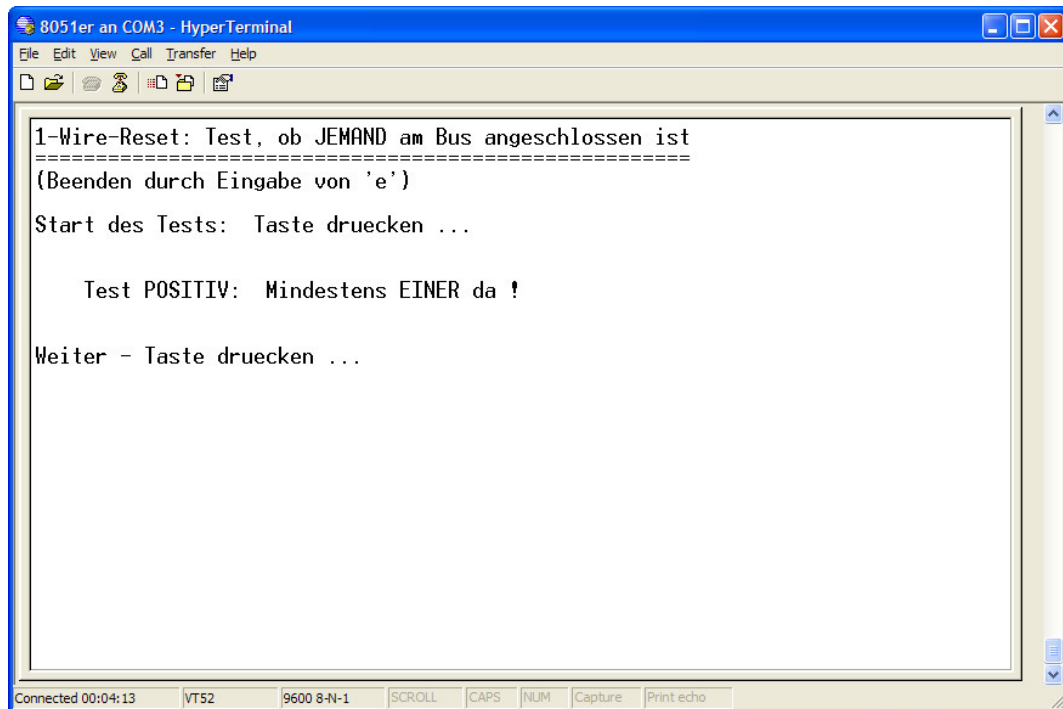


Abb.6.3.3: Mindestens ein Teilnehmer ist am 1-Wire-Bus angeschlossen

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):

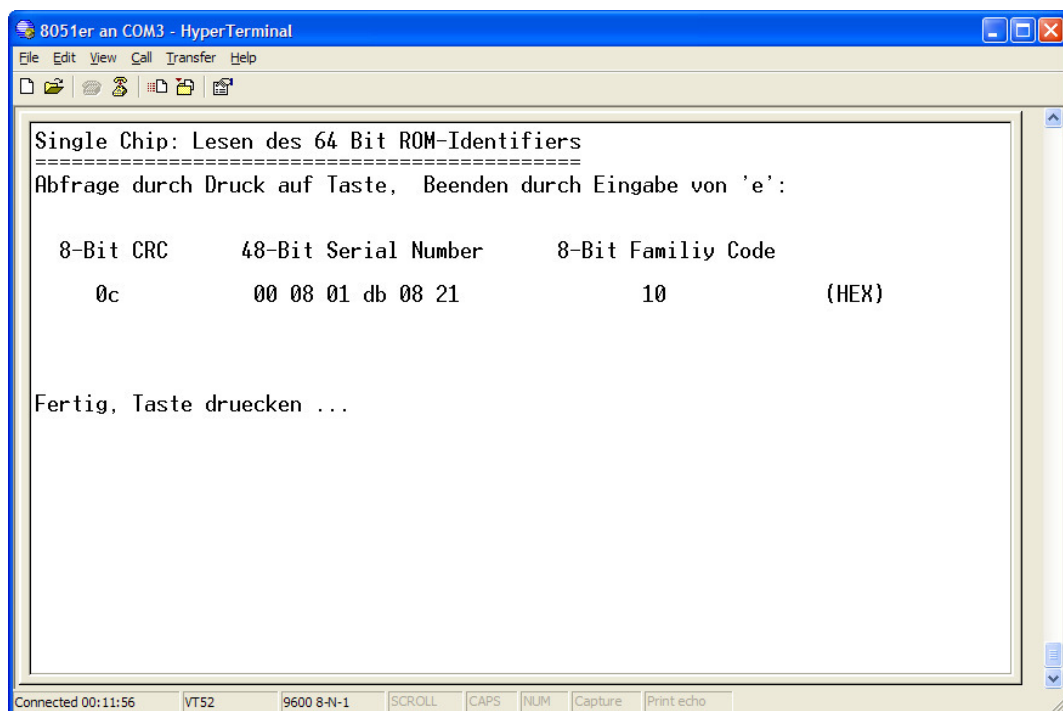
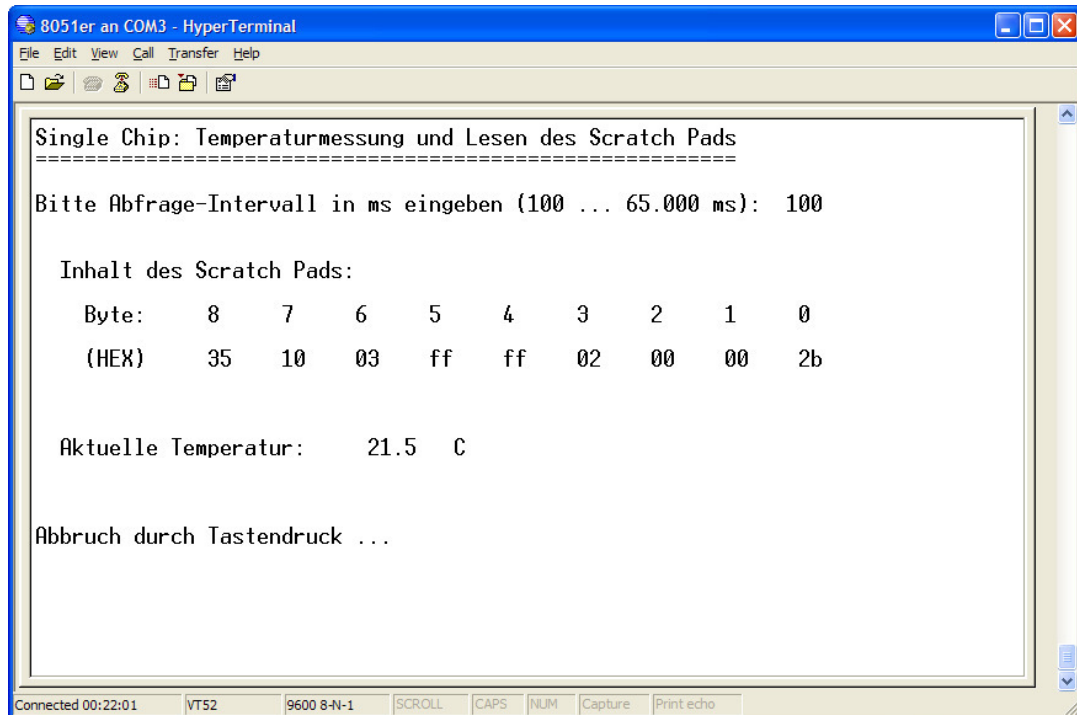


Abb.6.3.4: Das Auslesen des ROM-Identifiers

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**:

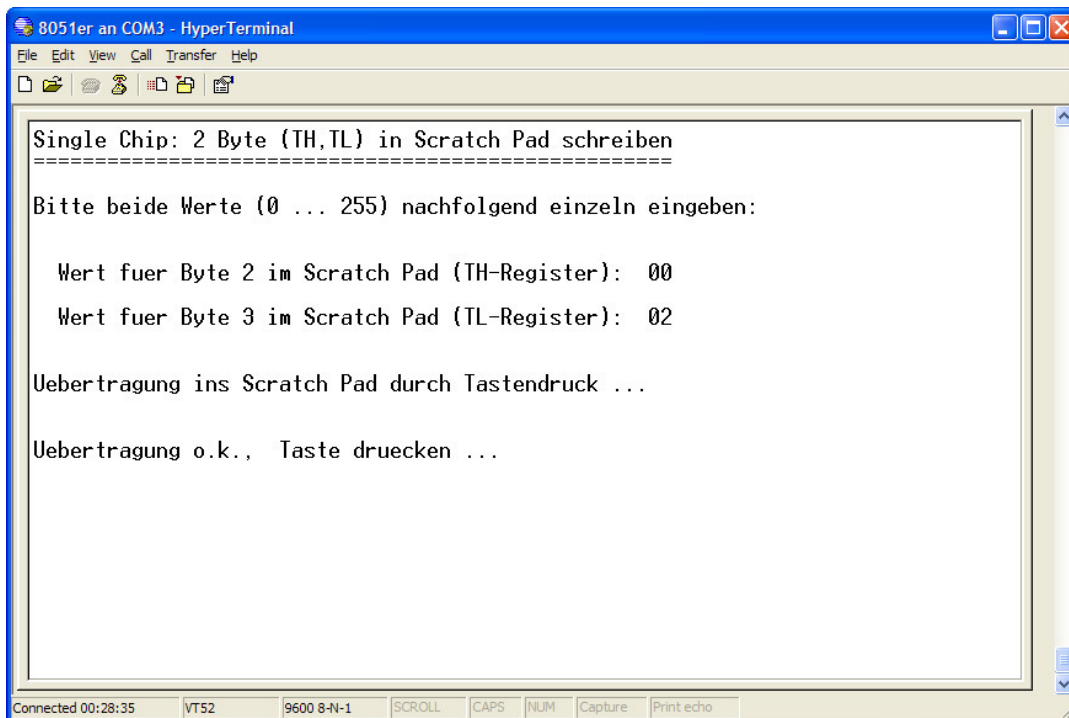


Abb.6.3.6: Das Beschreiben der TH- und TL-Register

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**:

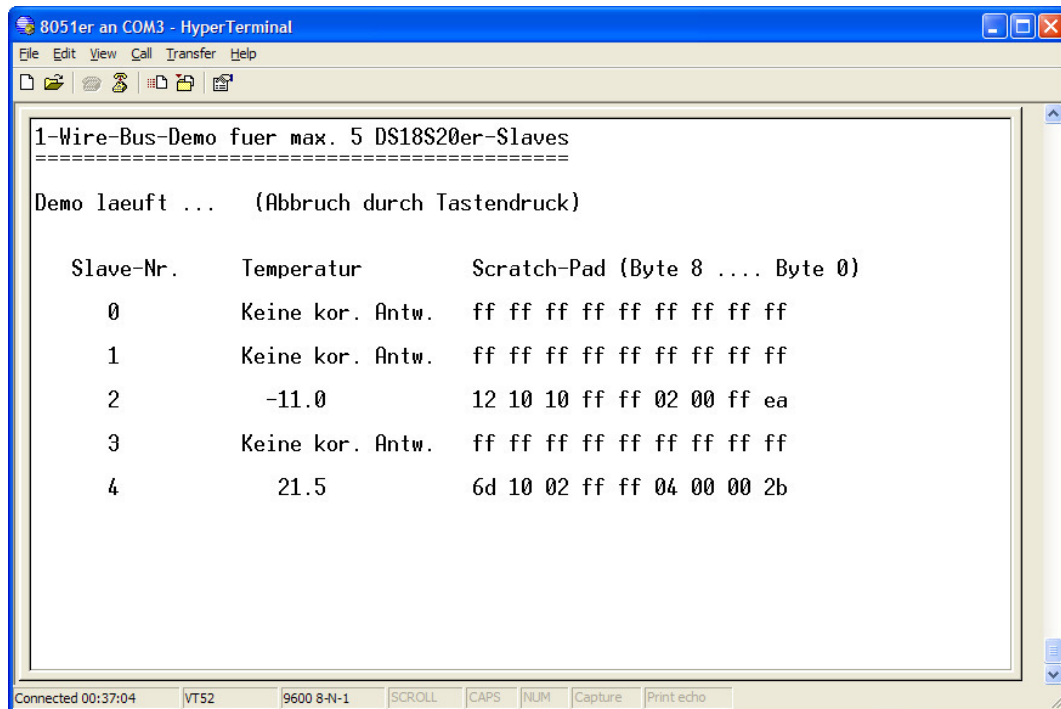


Abb.6.3.7: Demo-Betrieb mit bis zu fünf DS18S20ern

Die ROM-Identifizierer 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:**

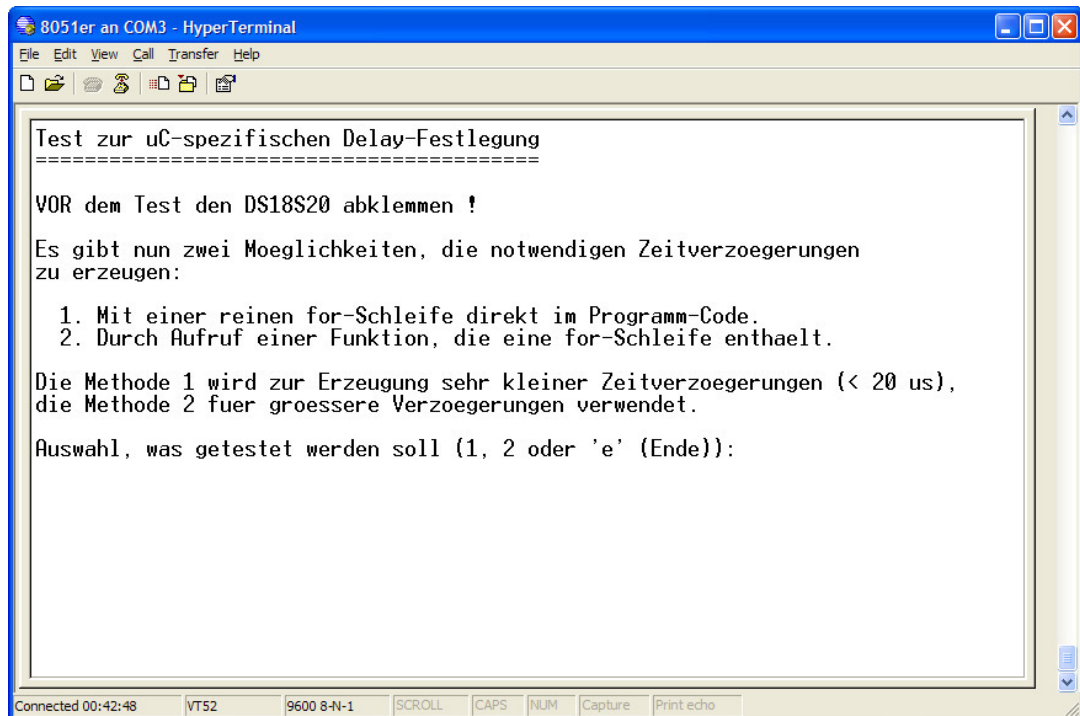


Abb.6.3.8: Ein einfaches Hilfsmittel zum „Zeit-Tuning“

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 !

7. Der DS2450 - 4-fach A/D-Wandler

Der zweite Baustein in unserem 1-Wire-System ist der DS2450er, der dem Anwender einen 4-kanaligen A/D-Wandler mit einstellbaren Auflösungen und zwei unterschiedlichen Referenzspannungen zur Verfügung stellt.

Nachdem wir die Funktion, die Ansteuerung und den Hardware-Anschluss des Chips ausführlich beschrieben haben, entwickeln wir die zugehörige DS2450er-API und schließen dieses Kapitel mit einer kleinen DS2450er-Demo-Software ab.

7.1 Funktion

Das Blockschaltbild des **internen Aufbaus** des DS2450ers zeigt die **Abb.7.1.1**:

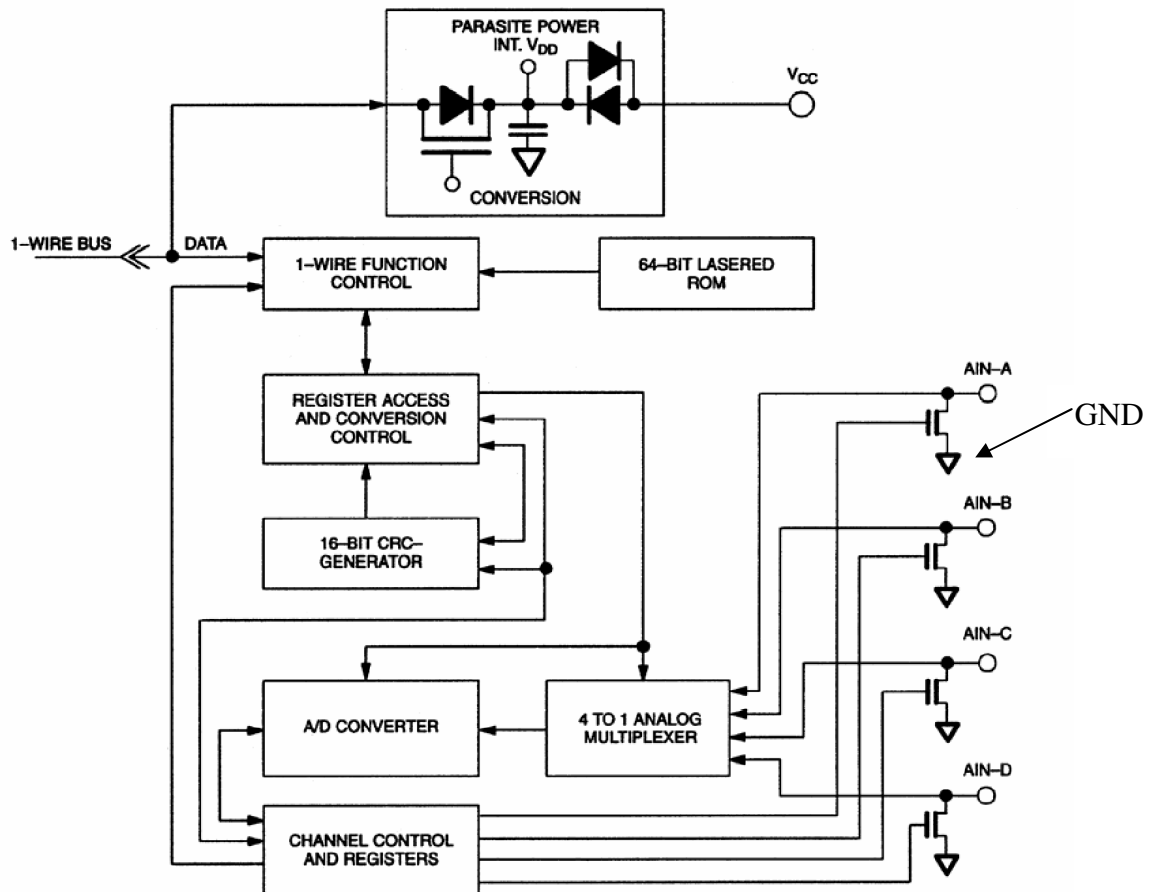


Abb.7.1.1: Das Blockschaltbild des DS2450ers

Viele Funktionsblöcke des DS2450er sind schon vom DS18S20er her bekannt:

- Baugruppe zur parasitären Speisung ('Parasite Power'),
- '1-Wire Function Control',
- '64-Bit Lasered ROM'.

Neu hinzugekommen sind:

- 'Register Access an Conversion Control' zur Steuerung der A/D-Wandlung und zur Abspeichern der Ergebnisse.,
- '16-Bit CRC-Generator zur Realisierung einer verbesserten Datensicherung,
- Der 'A/D Converter' selber als Herz der Baugruppe mit dem 4-fach Eingangsumschalter zur Auswahl von einem der vier Kanäle ('4 To 1 Analog Multiplexer').

- 'Channel Control and Registers' zur weiteren Kontrolle der A/D-Wandler-Baugruppe.

Die wesentlichen Kerndaten des A/D-Wandler-Bausteins haben wir nachfolgend einmal zusammen gefasst:

- Vier hochohmige analoge Eingänge A, B, C und D (Eingangsimpedanz: 0,5 M Ω ... 1,5 M Ω).
- Zwei für jeden Kanal unabhängig programmierbare Referenzspannungen: 2,56 V oder 5,12 V verfügbar.
- Programmierbare Auflösung, für jeden Kanal unabhängig einstellbar, zwischen 1 und 16 Bits.
- Nicht benutzte analoge Eingänge können alternativ als digitale Open-Drain-Ausgänge verwendet werden.
- Einfache +5 V-Betriebsspannungsversorgung.
- Komplette Kommunikation über den 1-Wire-Bus: Standard-Datenübertragungsrate von bis zu 16,3 kBit/s, Overdrive-Geschwindigkeit bis zu 142 kBit/s, parasitäre oder lokale Speisung möglich, eindeutiger 64 Bit-ON-Chip-ROM-Identifizier in jeden Chip fest einprogrammiert.
- Insgesamt 4 interne Register-Bänke bzw Register-Seiten (Pages) zu je 8 Byte (insgesamt also 32 Byte) steuern die gesamten Funktionen des Chips.

Beim praktischen Betrieb des DS2450er sind noch zwei Punkte zu beachten:

- 1) Die **normale parasitäre Speisung** des Chips über die Datenleitung DQ (mit 4,7 k Ω -Pull-Up-Widerstand nach +5 V, s. Kap.2.3.2) reicht lediglich für die Abwicklung von reinen Kommunikationsaufgaben aus.
Sollen dagegen A/D-Wandlungen durchgeführt werden, so muß der DS2450er auf jeden Fall durch einen „**Strong-Pull-Up**“ versorgt werden (s. Kap.2.3.2).
In unserer Applikation auf dem PT-1-Wire-ExBo betreiben wir den Baustein daher mit einer lokalen Speisung.
- 2) Der DS2450er behält nach dem Abschalten der Betriebsspannung nicht seine zuvor programmierten Betriebs-Parameter, d.h. nach jedem Einschalten der Betriebsspannung muß der DS2450er zuerst immer **neu parametrisiert werden**.

Die 1-Wire-Kommandos bzw. die Baustein-spezifischen Anweisungen, die der DS2450er „versteh“ sind in der **Abb.7.1.2** aufgeführt:

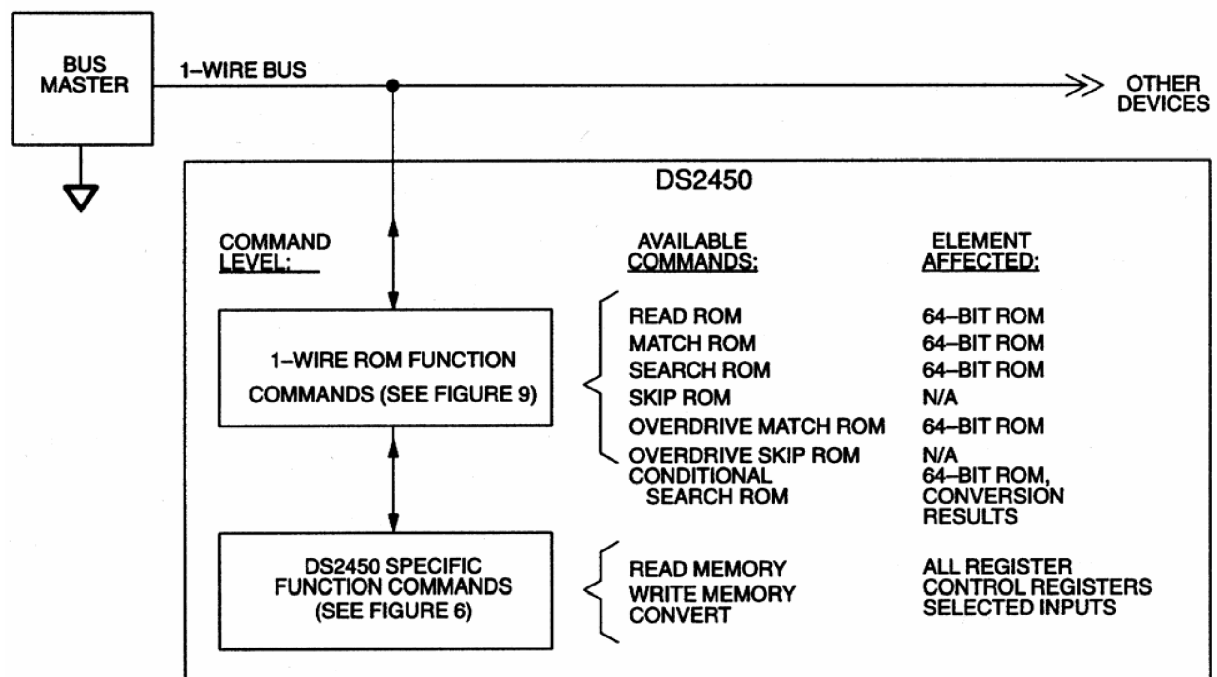


Abb.7.1.2: Der Befehlsumfang des DS2450er

Unsere nachfolgend entwickelte DS2450er-API bzw. die bereits beim DS18S20er erstellte 1-Wire-API unterstützen nun eine (wesentliche) Teilmenge dieser Befehle:

1-Wire-API ('ow.c/h', bereits vorhanden):

- Read ROM
- Match ROM
- Skip ROM

DS2450-API ('ds2450.c/h', wird nachfolgend neu entwickelt):

- Read Memory
- Write Memory
- Convert

Der ROM-Identifizierer der DS2450er-Basuteine hat den bereits bekannten Aufbau von 1-Wire-Identifizierern **Abb.7.1.3:**

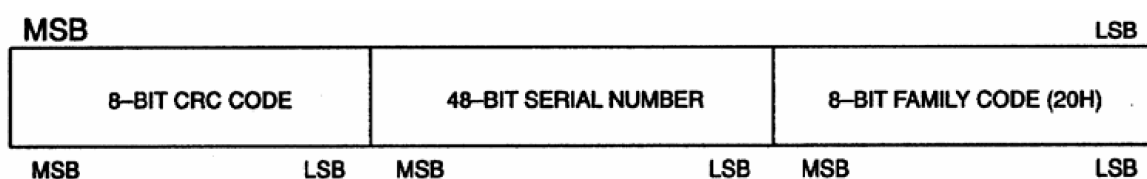


Abb.7.1.3: Der ROM-Identifizierer der DS2450er

Der einheitliche „Familien-Mitglieds-Identifikations-Code“ der DS2450er ist also 20h und der 8 Bit-CRC-Code wird, wie in Kapitel 2.4.6 beschrieben, gebildet.

Für die Erstellung der Anwendungs- und Betriebssoftware ist es nun zuerst einmal wesentlich, den Aufbau und die Bedeutung der **vier internen Speicher-Seiten (Pages)** mit ihren **insgesamt 32 Bytes** zu verstehen.

Beginnen wir daher mit der Erläuterung der

Page 0

In den 8 Bytes dieser Seite sind die Wandlungsergebnisse der vier Kanäle abgelegt,

Abb.7.1.4:

Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
00	A	A	A	A	A	A	A	LSBIT A
01	MSBIT A	A	A	A	A	A	A	A
02	B	B	B	B	B	B	B	LSBIT B
03	MSBIT B	B	B	B	B	B	B	B
04	C	C	C	C	C	C	C	LSBIT C
05	MSBIT C	C	C	C	C	C	C	C
06	D	D	D	D	D	D	D	LSBIT D
07	MSBIT D	D	D	D	D	D	D	D

Abb.7.1.4: Die Memory Page 0: „Wandlungsergebnisse“

Die internen Adressen dieser Register-Bank laufen von 00h bis 07h, d.h. über diese Adressen werden die einzelnen Bytes dann später über die 1-Wire-Betriebsssoftware angesprochen, wenn die Wandlungsergebnisse ausgelesen werden sollen.

Da die maximale Auflösung eines jeden Kanals 16 Bit beträgt, stehen für jeden Kanal genau 2 Bytes zur Aufnahme des Wandlungsergebnisses zur Verfügung: das jeweilige High-Byte und das Low-Byte für die Kanäle A, B, C und D.

Nach dem Power ON haben diese Page-Bytes den Wert 00h.

Da die gewünschte Auflösung vom Anwender aus im Bereich von 1 bis 16 Bits programmiert werden kann, muß man hier allerdings genauestens beachten, wie die Ergebnisbits in den beiden Bytes positioniert werden, wenn eine Auflösung kleiner als 16 Bit gewählt wird,

Abb.7.1.5:

High-Byte, Adr. 01h								Low-Byte, Adr. 00h							
MSBIT feste Stelle															LSBIT variable Stelle
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

(X ≡ maßgebliche Ergebnis-Bits)

Abb.7.1.5: Die Abspeicherung des Wandlungsergebnisses beim Kanal A

Am Beispiel von Kanal A (≡ Ergebnis-Bytes mit den Adressen 00h und 01h in der Page 0) sollen die beim DS2450er geltenden Abspeicher-Regeln näher erläutert werden:

Das High-Byte des Ergebnisses wird unter der höheren Adresse, das Low-Byte unter der niedrigeren Adresse abgespeichert.

Während das höherwertigste Bit, das **MSBIT**, des Ergebnisses immer an der gleichen Stelle stehen bleibt, nämlich als höherwertigstes Bit des High-Bytes, ändert das niederwertigste Bit, das **LSBIT**, des Ergebnisses seine Stellung im Low-Byte je nach gewählte Auflösung: ist die Auflösung kleiner als 16 Bit, so "wandert" das LSBIT des Ergebnisses um die entsprechende Bitzahl, die bis zu 16 Bits fehlen, nach links.

Die dann rechts vom LSBIT liegenden, jetzt nicht benötigten Bits, werden vom DS2450er automatisch auf log. '0' gesetzt.

In der **Abb.7.1.6** sind dazu drei Beispiele gezeigt:

Auflösung: 12 Bit

High-Byte, Adr. 01h								Low-Byte, Adr. 00h							
MSBIT feste Stelle											LSBIT				
X	X	X	X	X	X	X	X	X	X	X	X	0	0	0	0

(X ≡ maßgebliche Ergebnis-Bits)

Auflösung: 8 Bit

High-Byte, Adr. 01h								Low-Byte, Adr. 00h							
MSBIT feste Stelle							LSBIT								
X	X	X	X	X	X	X	X	0	0	0	0	0	0	0	0

(X ≡ maßgebliche Ergebnis-Bits)

Auflösung: 4 Bit

High-Byte, Adr. 01h								Low-Byte, Adr. 00h							
MSBIT feste Stelle			LSBIT												
X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0

(X ≡ maßgebliche Ergebnis-Bits)

Abb.7.1.6: Die unterschiedlichen Auflösungen

Nur bei der maximal möglichen Auflösung von 16 Bit werden dementsprechend alle 16 Bits gebraucht, sind also alle 16 Bits gültig.

Will man nun aus den beiden Ergebnis-Bytes das komplette Endergebnis „zusammenbauen“ so muß man ein wenig geschickt nachdenken.

Für eine Realisierung dieser Zusammenfassung in der Programmiersprache 'C' ergibt sich daher:

```
unsigned int wert;
.....
wert = (High_Byte * 256) + Low_Byte;
wert = wert >> (16 - Aufloesung);
```

Zuerst werden beide Bytes zu einer unsigned int Zahl namens 'wert' zusammengefasst und dann nach rechts verschoben, um die überflüssigen Nullen herausfallen zu lassen und um das Ergebnis in die richtige Stellenwertigkeit zu bringen.

Die Anzahl der Stellen, um die die Zahl nach rechts geschoben werden muß, ergibt sich aus: (16 - gewählte Aufloesung), s. dazu Abb.7.1.6. Bei einer Auflösung von 12 Bit muß die Zahl um 4 Stellen, bei einer Auflösung von 8 Bit um 8 Stellen und bei einer Auflösung von 4 Bit um 12 Stellen nach rechts geschoben werden. Bei einer 16 Bit Auflösung wird nicht geschoben, denn dann ist ja (16 - Aufloesung) auch gleich Null.

Und schon hat man das zusammengefasste Wandlungsergebnis des entsprechenden Kanals.

Page 1

In den 8 Bytes dieser Seite (mit den Adressen 08h ... 0fh) sind die Kontroll- und Statusinformationen zu jedem der vier Kanäle abgelegt, **Abb.7.1.7:**

Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
08	OE-A	OC-A	0	0	RC3-A	RC2-A	RC1-A	RC0-A
09	POR	0	AFH-A	AFL-A	AEH-A	AEL-A	0	IR-A
0A	OE-B	OC-B	0	0	RC3-B	RC2-B	RC1-B	RC0-B
0B	POR	0	AFH-B	AFL-B	AEH-B	AEL-B	0	IR-B
0C	OE-C	OC-C	0	0	RC3-C	RC2-C	RC1-C	RC0-C
0D	POR	0	AFH-C	AFL-C	AEH-C	AEL-C	0	IR-C
0E	OE-D	OC-D	0	0	RC3-D	RC2-D	RC1-D	RC0-D
0F	POR	0	AFH-D	AFL-D	AEH-D	AEL-D	0	IR-D

Abb.7.1.7: Die Memory Page 1: „Kontroll- und Statusinformationen“

Für jeden Kanal existieren je zwei Bytes, die für jeden Kanal den gleichen Aufbau und den gleichen Inhalt haben.

Für den Kanal A ist diese Struktur einmal detaillierter in der **Abb.7.1.8** dargestellt.

High-Byte, Adr. 09h								Low-Byte, Adr. 08h							
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
PO R	0	AFH- A	AFL- A	AEH- A	AEL- A	0	IR- A	OE- A	OC- A	0	0	RC3- A	RC2- A	RC1- A	RC0- A
X	0	X	X	X	X	0	X	X	X	0	0	X	X	X	X

(der Kennbuchstabe 'A' steht hier für 'Kanal A')

Abb.7.1.8: Der Aufbau der beiden Kontroll- und Status-Bytes beim Kanal A

Es bedeuten hierbei:

POR

Power ON Reset: wird vom DS2450er automatisch auf log. '1' gesetzt, wenn der Chip zurück gesetzt wurde (nach dem Einschalten der Betriebsspannung (Power On)).

Das ist dann für den Anwender ein Zeichen dafür (wenn der das High-Byte ausliest und das POR-Bit untersucht), dass der Baustein komplett neu initialisiert werden muß, denn alle Einstellungen gehen verloren, wenn die Betriebsspannung abgeschaltet und dann wieder eingeschaltet wird.

Der 1-Wire-Master muß also die Grundinitialisierung durchführen und dabei das POR-Bit auf log. '0' zurück setzen. **Erst dann funktioniert der DS2450er einwandfrei !**

AFH

Alarm Flag 'High': dieses Flag-Bit wird vom DS2450er automatisch gesetzt, wenn der vom Anwender gesetzte High-Grenzwert vom Wandlungswert des Kanals überschritten wird (s. Beschreibung zur Page 2).

Dieses Bit wird nach jeder Wandlung neu gesetzt bzw. zurückgesetzt, je nach Größe des aktuellen Wandlungswertes.

AFL

Alarm Flag 'Low': dieses Flag-Bit wird vom DS2450er automatisch gesetzt, wenn der vom Anwender gesetzte Low-Grenzwert vom Wandlungswert des Kanals unterschritten wird (s. Beschreibung zur Page 2).

Dieses Bit wird nach jeder Wandlung neu gesetzt bzw. zurückgesetzt, je nach Größe des aktuellen Wandlungswertes.

AEH

Alarm enable 'High': dieses Bit legt fest, ob sich der Baustein beim „Conditional Search“-Algorithmus meldet, wenn der Master alle Chips abfragt, ob bei ihnen der High- oder der Low-Grenzwert über- bzw. unterschritten wurde.

AEH = 1 bedeutet, dass sich der Chip beim Master-Suchdurchlauf meldet, wenn der High-Grenzwert überschritten wurde, wenn also auch gleichzeitig das Flag-Bit AFH gesetzt wurde. Bei AEH = 0 meldet sich der Chip nicht während des Master-Suchdurchlaufes, egal ob der High-Grenzwert überschritten wurde oder nicht.

So kann der 1-Wire-Master sehr einfach all diejenigen Stationen am Bus herausuchen, bei denen der zuvor eingestellte High-Grenzwert überschritten wurde.

Diese Möglichkeit des Betriebes haben wir allerdings bei unserer nachfolgend erstellten Demo-Software für den DS2450er (noch) nicht implementiert.

AEL

Alarm enable 'Low': dieses Bit legt fest, ob sich der Baustein beim „Conditional Search“-Algorithmus meldet, wenn der Master alle Chips abfragt, ob bei ihnen der High- oder der Low-Grenzwert über- bzw. unterschritten wurde.

AEL = 1 bedeutet, dass sich der Chip beim Master-Suchdurchlauf meldet, wenn der Low-Grenzwert unterschritten wurde, wenn also auch gleichzeitig das Flag-Bit AFL gesetzt wurde. Bei AEL = 0 meldet sich der Chip nicht während des Master-Suchdurchlaufes, egal ob der Low-Grenzwert unterschritten wurde oder nicht.

So kann der 1-Wire-Master sehr einfach all diejenigen Stationen am Bus herausuchen, bei denen der zuvor eingestellte Low-Grenzwert unterschritten wurde.

Diese Möglichkeit des Betriebes haben wir allerdings bei unserer nachfolgend erstellten Demo-Software für den DS2450er (noch) nicht implementiert.

IR

Input voltage range: mit diesem Bit wird die jeweilige Referenzspannung für den entsprechenden Kanal festgelegt: ISR = 0 wählt 2,56 V und ISR = 1 wählt 5,12 V als Referenzspannung aus.

Und **das Besondere** hierbei ist: der DS2450er erzeugt sich intern diese beiden Referenzspannungen selber, eine irgendwie geartete externe Beschaltung durch den Anwender ist hier nicht erforderlich !

OE

Output enable: über dieses Bit wird die grundlegende Funktion des jeweiligen Port-Pins AIN-A, AIN-B, AIN-C und AIN-D festgelegt, s. Abb.7.1.1.

Bei OE = 0 gilt: der Port-Pin wird als **analoger Eingang** für den internen A/D-Wandler betrieben: es können also analoge Spannungen angeschlossen werden, die dann gewandelt werden.

Das Bit OC hat in diesem Falle keine Bedeutung.

Bei OE = 1 wird der Port-Pin als **digitaler Open Drain Ausgang** betrieben, und über das Bit OC können die Zustände log. '0' oder log. '1' am Pin ausgegeben werden.

OC

Output Control: dieses Bit ist nur dann wirksam, wenn das Bit OE auf log. '1' gesetzt ist (\equiv Pin als digitaler Open Drain Ausgang betrieben), s. Abb.7.1.1: die jeweiligen Transistoren an den Ausgängen spielen jetzt eine wesentliche Rolle.

Es gilt hierbei:

OC = 0 \equiv der jeweilige Ausgangstransistor wird leitend geschaltet, am (Open Drain) Anschlusspin erscheint eine log. '0'.

OC = 1 \equiv der jeweilige Ausgangstransistor wird gesperrt, am (Open Drain) Anschlusspin erscheint eine log. '1'.

(Beim Betrieb als digitaler Open Drain-Ausgang ist also ein Pull-Up-Widerstand nach +5 V erforderlich und der Ausgangszustand entspricht dann direkt dem Zustand des Bits OC).

In unsere Applikation auf dem PT-1-Wire-ExBo betreiben wir allerdings alle vier Kanäle als analoge Eingänge für den A/D-Wandler.

RC3 – RC0

Read converted: diese vier Bit legen die Auflösung fest, mit der der entsprechende Kanal gewandelt wird.

Es gilt dabei der folgende Zusammenhang, **Tab.7.1.1**:

RC3	RC2	RC1	RC0	Auflösung / Bit
0	0	0	0	16
0	0	0	1	1
0	0	1	0	2
....
1	1	1	0	14
1	1	1	1	15

Tab.7.1.1: Die Einstellung der Auflösung

Die Anordnung der Wandlungs-Ergebnisbits auf der Page 0 bei eingestellten Auflösungen unterhalb von 16 Bits wurde ja bereits schon bei der Beschreibung zur Page 0 näher erläutert.

Ein kleines Beispiel soll die Einstellungen auf der Page 1 verdeutlichen:

Beispiel für Kanal A:

- Betrieb als analoger Eingang,
- keine Überwachung von Grenzwertverletzungen,
- 5,12 V Referenzspannung,
- Auflösung: 8 Bit,
- POR-Bit auf log. '0' setzen.

Mit diesen Vorgaben erhält man für die beiden Kanal A-Kontroll- und Status-Bytes:

High-Byte, Adr. 09h								Low-Byte, Adr. 08h							
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
PO R	0	AFH- A	AFL- A	AEH- A	AEL- A	0	IR- A	OE- A	OC- A	0	0	RC3- A	RC2- A	RC1- A	RC0- A
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0

Somit ergeben sich folgende Einträge in die internen SFRs des DS2450:

- SFR mit der Adresse 09h: 01h
- SRF mit der Adresse 08h: 08h

Die Default-Werte nach dem Einschalten der Betriebsspannung für diese beiden Kontroll- und Status-Bytes für jeden Kanal sind:

- High-Byte: 8ch
- Low-Byte: 08h

Page 2

In den 8 Bytes dieser Seite (mit den Adressen 10h ... 17h) werden die High- und Low-Alarmgrenzen, mit denen die gewandelten Messwerte verglichen werden können, abgelegt, **Abb.7.1.9:**

Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
10	MSBL-A	A	A	A	A	A	A	LSBL-A
11	MSBH-A	A	A	A	A	A	A	LSBH-A
12	MSBL-B	B	B	B	B	B	B	LSBL-B
13	MSBH-B	B	B	B	B	B	B	LSBH-B
14	MSBL-C	C	C	C	C	C	C	LSBL-C
15	MSBH-C	C	C	C	C	C	C	LSBH-C
16	MSBL-D	D	D	D	D	D	D	LSBL-D
17	MSBH-D	D	D	D	D	D	D	LSBH-D

Abb.7.1.9: Die Memory Page 2: „Festlegung der High- und Low-Alarmgrenzen“

Für jeden Kanal ist ein Byte zur Festlegung der unteren und der oberen Alarmgrenze vorhanden: das jeweilige L- und das jeweilige H-Byte

Hierbei ist ein **wichtiger Punkt** zu beachten:

Die Auflösung bei der Wandlung durch den DS2450er kann ja maximal 16 Bit betragen, die beiden Vergleichsgrenzwerte werden aber (aus Speicher-Platzgründen) nur 8 Bit breit angegeben.

Daher gilt hier folgende Festlegung:

Gewählte Auflösung \geq 8 Bit

In diesen Fall stellen die im Register eingetragenen 8 Alarmgrenzenbits die 8 MSBits des Ergebnisses dar, mit anderen Worten: der High-Alarm wird ausgelöst (das entsprechende Flag-Bit gesetzt), wenn die MSBits des Ergebnisses größer sind als die festgelegten Bits im H-Byte-Register.

Analog: der Low-Alarm wird ausgelöst, wenn die 8 MSBits des Wandlungsergebnisses kleiner sind als der im L-Register eingetragene Wert.

Gewählte Auflösung $<$ 8 Bit

Jetzt werden die nicht benötigten LSBits des Grenzregister-Wertes ignoriert und nur die MSBits werden zum Vergleich herangezogen.

Beträgt die gewählte Auflösung z.B. 4 Bit, so werden vom Alarmregister auch **nur die vier oberen Bits** für den Vergleich verwendet.

Nach dem Einschalten der Betriebsspannung (Power On) besitzen diese beiden Grenzwert-Bytes für jeden Kanal die folgenden Grundwerte:

- Untere Grenze (L-Byte): 00h
- Obere Grenze (H-Byte): ffh

Page 3

Die 8 Bytes auf dieser Seite (Adressen 18h ... 1fh) sind nur für den Hersteller von Interesse, denn hier werden bei der Chipfertigung wichtige Kalibrierungswerte eingetragen, **Abb.7.1.10:**

Address	
18	Factory Calibration (do not change)
19	Factory Calibration (do not change)
1A	Factory Calibration (do not change)
1B	Factory Calibration (do not change)
1C	Set to 40 hex if VCC powered
1D	Factory Calibration (do not change)
1E	Factory Calibration (do not change)
1F	Factory Calibration (do not change)

Abb.7.1.10: Die Memory Page 3: „Wichtige Kalibrierungsdaten“

Mit anderen Worten: der Anwender sollte hier absolut nichts eintragen (obwohl dieses möglich ist), weil dadurch die ordnungsgemäße Funktion des DS2450ers gestört wird.

Nur durch einen Power On-Reset kann dann die korrekte Arbeitsweise des Bausteins wieder hergestellt werden.

Es gibt hierbei allerdings eine extrem wichtige Ausnahme:

In das Byte mit der Adresse 1ch **MUSS unbedingt** der Wert 40h eingetragen werden, wenn der DS2450er lokal gespeist wird („VCC powered“), weil sonst der gesamte Chip NICHT ordnungsgemäß funktioniert !

Da wir auf unserem PT-1-Wire-ExBo den DS2450er auf jeden Fall lokal speisen, dürfen wir bei der Initialisierung des Chips, nach dem Einschalten der Betriebsspannung, diesen (Einzel)Eintrag in dieses SFR auf der Page 3 nicht vergessen !

Damit haben wir nun die Grundfunktionen und die wesentlichen Einstellmöglichkeiten des DS2450ers beschrieben und können uns jetzt etwas näher mit dem Kommunikationsablauf zwischen dem 1-Wire- Master und diesem Baustein beschäftigen.

7.2 Ablauf der Kommunikation

In den allgemeine Betrachtungen zum 1-Wire-Bus haben wir ja ausgeführt, dass der Vorteil eines (jeden) Busses darin besteht, dass alle Stationen Hardware-mäßig gleich angeschlossen und Software-mäßig gleich betrieben werden.

Gerade dadurch ergeben sich ja die großen „Vereinheitlichungs“-Vorteile für den Anwender beim Einsatz eines (genormten) Bussystems.

So sollte es eigentlich auch beim 1-Wire-Bus sein.

Sie ahnen es aber sicherlich schon: dieses ist bei den verschiedenen 1-Wire-Slave-Bausteinen nicht durchgängig der Fall !

In Kapitel 2.4.1 haben wir dargelegt, dass es im wesentlichen zwei große Gruppen von 1-Wire-Funktionen gibt:

- die höheren 1-Wire-Funktionen, die so genannten **ROM Commands** und die
- **Slave-(Baustein-)spezifischen Funktionen**.

Während die ROM Commands „Kommunikations-mäßig“ alle gleich abgewickelt werden, gibt es bei der Realisierung bzw. bei der Kommunikations-mäßigen Abwicklung der Baustein-spezifischen Funktionen teilweise (erhebliche) Unterschiede bei den einzelnen 1-Wire-Slave-Chips.

Zwei Beispiele sollen das vorab erläutern:

Beispiel 1:

Bei Auslesen der erfassten Temperaturwerte aus dem DS18S20er (s. Abb.6.2.2, Inhalt des Scratch Pads) werden diese Daten durch ein 8-Bit-CRC-Sicherungsbyte gegen Datenübertragungsfehler abgesichert.

Nun hat MAXIM/Dallas wohl im Laufe der Entwicklungsgeschichte der 1-Wire-Bausteine festgestellt, dass eine 8-Bit-CRC-Sicherung nicht ausreichend ist bzw. dass man den Datentransfer von und zu den 1-Wire-Bausteinen noch besser schützen sollte, und das es daher von Vorteil ist, eine höhere (optimalere) CRC-Sicherung durchzuführen.

Daher wird die Datenübertragung beim DS2450er durch eine **16-Bit-CRC-Checksumme** gesichert: es werden also 2 CRC-Sicherungsbytes mit gesendet und der DS2450 hat dafür zusätzlich einen neuen 16-Bit-CRC-Generator auf seinen Chip bekommen, s. Abb.7.1.1.

Der 1-Wire-Master muss also jetzt (zusätzlich) einen solchen 16-Bit-CRC-Check durchführen können, um die Datenintegrität (Datensicherheit) zu überprüfen.

Der „alte“ 8-Bit-CRC-Check auf der Master-Seite muss allerdings auch noch erhalten bleiben (und bei Bedarf durchgeführt werden), denn die Sicherung des ROM-Identifiers (ROM-Code, s. Abb.7.1.3) auf der Slave-Seite wird wie bisher mit dem 8-Bit-CRC durchgeführt (das gehört ja zu den 1-Wire-ROM Commands, die auf jeden Fall immer gleich ausgeführt werden).

Beispiel 2:

Die Baustein-spezifischen Kommunikationszyklen beim DS18S20er, z.B. 'Read Scratch Pad' enden einfach nachdem alle Daten übertragen worden sind.

Beim DS2450er ist am Ende **eines jeden** Kommunikationszyklusses immer noch ein Master-Reset-Impuls notwendig, auf den der Slave dann mit einem Presence-Impuls antwortet.

Man muß also immer ganz genau hinsehen, wie die einzelnen Zyklen ablaufen. Oder anders formuliert: die reine (Nutz)Datenübertragung vom/zum DS18S20er läuft anders ab als die Datenübertragung vom/zum DS2450er.

Von Vorteil ist hier allerdings, dass im Datenblatt des DS2450ers sehr ausführliche Flussdiagramme zur Programmierung des Datentransfers vorhanden sind und dass wir bereits alle dazu notwendigen 1-Wire-Grundfunktionen in unserer API erstellt haben. Wir müssen diese nur noch in die richtige Reihenfolge bringen.

Zum Betrieb des DS2450ers benötigen wir eigentlich nur drei Kommunikationszyklen:

- Auslesen von Inhalten aus den internen (Memory)Pages: „**Read Memory**“
- Einschreiben von Inhalten in die internen (Memory)Pages: „**Write Memory**“
- Befehl für den Start einer A/D-Wandlung mit Übergabe des gewünschten Kanals: „**Convert**“

Das jedes Byte auf den internen Pages eine eigene, eindeutige Adresse besitzt, kann man beim Speicher-Lesen bzw. beim Speicher-Schreiben noch zwei verschiedenen Arten von Zugriffen unterscheiden:

- Lesen einer gesamten Seite (Page) auf einmal, d.h. Auslesen aller 8 Bytes (Register) einer Seite hintereinander **oder** gezieltes Auslesen nur eines einzelnen Registers.
- Beschreiben einer gesamten Seite (Page) auf einmal, d.h. Beschreiben aller 8 Bytes (Register) einer Seite hintereinander **oder** gezieltes Beschreiben nur eines einzelnen Registers.

Da die 16-Bit-CRC-Datensicherung beim DS2450er beim Auslesen nur dann korrekt funktioniert, wenn eine Page komplett ausgelesen wird, programmieren wir alle nachfolgenden API-Routinen so, dass immer Seiten-weise geschrieben bzw. gelesen wird.

Der **Zeitverlust**, der entsteht, wenn in diesen Fällen eigentlich nicht benötigte Werte erneut geschrieben bzw. erneut ausgelesen werden, ist hier sicherlich vernachlässigbar, da die Datenübertragung über den 1-Wire-Bus für viele Anwendungsfälle schnell genug.

Kommt es allerdings im Zeitbedarf auf „die letzte Mikrosekunde“ an, so muß der Anwender jeweils ganz konkret optimierte Kommunikationszyklen entwickeln, die nur mit den wirklich benötigten Daten ablaufen.

Globale Variablen, I

Für die ersten Implementierungsschritte der DS2450er-API definieren wir uns zunächst **am Programm-Anfang** drei globale Variablen:

```

/***** Globale Variablen *****/
// Array, um die 8 Byte DS2450er-Page-Daten abzuspeichern: Sende/Empfangs-Richtung
unsigned char ds2450_p[8];

// Abspeicherung des 16-Bit-CRC-Wertes, zerlegt in zwei Bytes
unsigned char crc1,crc2;

```

Um die Daten einer Page aufzunehmen, verwenden wir ein 8 Byte großes Array aus unsigned char-Daten, das wir **ds2450_p[.]** nennen.

Dieses Feld werden wir sowohl für den Sende- als auch für den Empfangsbetrieb benutzen.

Weiterhin benötigen wir zwei unsigned char-Variablen, um den 16-Bit-CRC-Wert abzuspeichern, der vom DS2450er in Form von 2 einzelnen Bytes ausgesendet wird.

Diese Variablen nennen wir **‘crc1’** und **‘crc2’**.

Hinweis:

Zur Zeit werden diese beiden CRC-Informationen lediglich empfangen, aber nicht weiter ausgewertet; wir verzichten hier also zunächst auf eine Überprüfung der Datensicherung.

Kommunikationszyklus 1: Das Einschreiben in eine DS2450er-Page

Das Einschreiben in eine Seite ist nur sinnvoll für die internen Pages 1 und 2, bzw. für die Bytes mit den Adressen 08h .. 17h, s. Kap.7.1:

- Page 1: Festlegung der Kontroll-/Status-Daten für die vier Kanäle,
- Page 2: Festlegung der Grenzwertdaten für die vier Kanäle.

Für unsere DS2450er-API entwickeln wir nun eine Funktion namens

ds2450_write_page(...) an die die Nummer der gewünschten zu beschreibenden Page übergeben wird, also der Wert 1 oder 2. Somit hat der Kopf dieser Funktion das folgende Aussehen:

```

/** Schreiben von 8 Byte-Daten in eine DS2450er-Page; sinnvoll nur für Page 1 und 2 */
/** Inhalt des 8-Byte-Arrays ds2450_p[.] wird in die angegebene Page geschrieben. */
/** LSByte der Page aus der Array-Stelle 0, usw. */
/** p_nr = 1 oder 2 */

void ds2450_write_page(unsigned char p_nr)

```

Die eigentlichen Daten, die in die Speicherstellen der ausgewählten Page geschrieben werden, müssen vorher im globalen Array (Vektor) `ds2450_p[.]` abgelegt werden und zwar gilt: der Wert für die erste Speicherstelle der Page steht in `ds2450_p[0]`, ..., der Wert für die letzte Speicherstelle der Page steht in `ds2450_p[7]`.

Die Funktion `ds2450_write_page(...)` überträgt nun die Werte aus dem Vektor `ds2450_p[.]` an den DS2450er, wo sie dann in die gewünschte Page eingeschrieben werden.

Bevor jetzt der ganz konkrete Schreibvorgang abläuft, sind zuerst noch die bekannten allgemeinen „einführenden“ 1-Wire-Anfangsbefehle vorzuschalten:

- Der Master sendet einen Reset-Impuls und der Slave antwortet mit einem Presence-Impuls.
Hierzu verwenden wir die neue Funktion `ow_pres_check(...)`.
- Der Master sendet ein ROM Command zur Stationsadressierung: entweder `'Skip ROM'`, wenn nur ein DS2450er angeschlossen ist oder `'Match ROM'` (mit passender Stations-Adresse), wenn sich mehrere DS2450er am Bus befinden.
Dazu verwenden wir die neue Funktion `ow_match()` aus der 1-Wire-API.

Wichtiger Hinweis:

Wir verwenden hier, abweichend vom Betrieb des DS18S20ers, zwei neue Funktionen aus der 1-Wire-Grund-API:

- `ow_pres_check(...)`

und

- `ow_match()`

denn wir haben festgestellt, dass man damit die allgemeine Adressierung von 1-Wire-Slave-Bausteinen optimaler gestalten kann.

Wir werden diese neuen API-Elemente im Kapitel 7.4 näher erläutern; sie kommen allerdings hier beim Betrieb des DS2450er schon (kommentarlos) zu Einsatz.

Somit sehen die ersten Anweisungen in unserer Funktion `ds2450_write_page(...)` wie folgt aus:

```
// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(20);

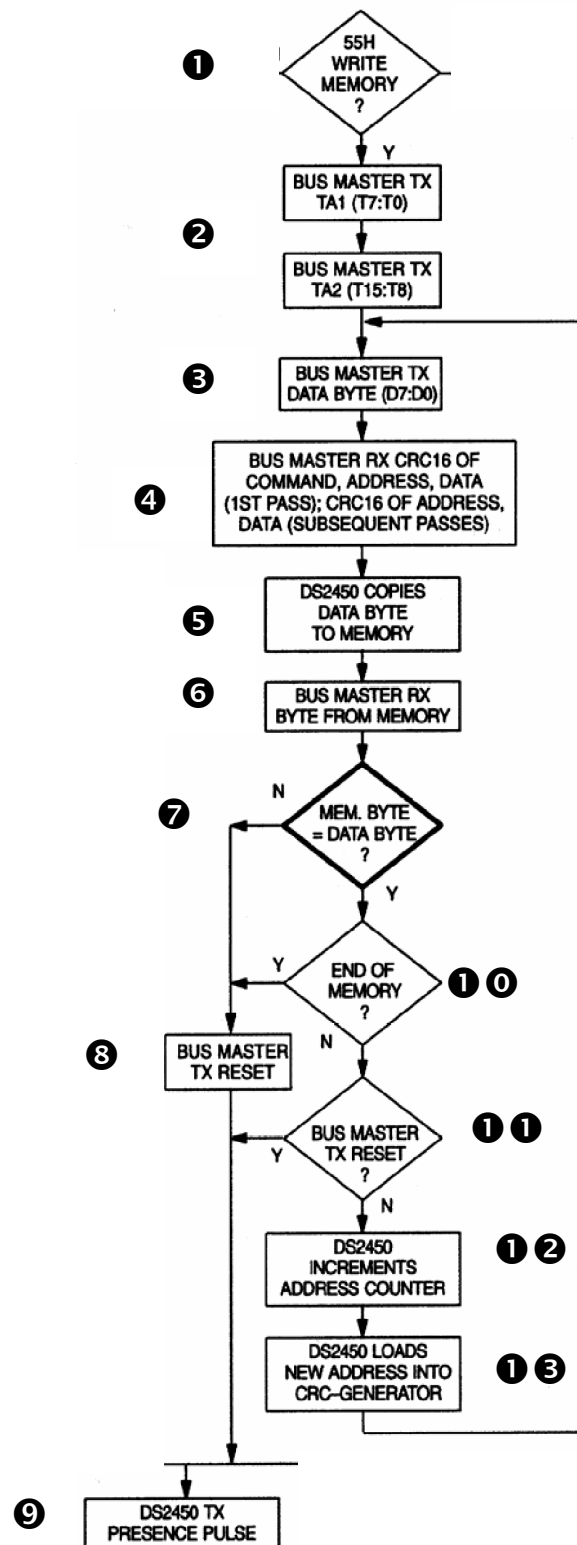
/**** Werte in übergebene Page einschreiben ****/
/**** Adressierung des jeweils als aktiv ausgewählten Slaves
```

```
ow_match();
```

Als Fehlermeldung bei der Slave-Presence-Abfrage wird hier der Text 20 ausgewählt und die globale Variable 'ow_sl_aktiv' wird zuvor im main-Teil des Programms auf den Wert 255 gesetzt, d.h. wir haben im 1-Wire-System nur einen einzigen DS2450er und adressieren diesen nicht über seine eindeutige interne Adresse (ROM Identifier) sondern über die allgemeine 'Skip ROM'-Anweisung.

In diesem Fall müssen natürlich alle anderen 1-Wire-Slaves vom Bus entfernt worden sein !

Im Datenblatt des DS2450er findet man nun zum Einschreiben in eine Page das folgende Flussdiagramm, **Abb.7.2.1**:



(TX ≡ sendet, RX ≡ empfängt)

Abb.7.2.1: Flussdiagramm zum Einschreiben in eine Page

Und es bedeuten nun:

①:

Der Master sendet zuerst das zugehörige 'Write Memory'-Befehlswort: 55h.

In unserer neuen Funktion 'ds2450_write_page(...)' geht es daher weiter mit:

```
// Schreib-Befehl senden: "WRITE Memory" = 0x55
ow_wr_byte(0x55);
```

②:

Danach muß die Adresse des internen DS2450er-Registers übertragen werden, dessen Inhalt als erstes beschrieben werden soll.

Das ist in unserem Fall immer die Adresse des ersten Bytes auf der Page, also die Startadresse einer Page, somit die Startadresse der Page 1 (= 08h) oder die der Page 2 (= 10h).

Und hier ergibt sich die **erste Besonderheit**: der DS2450er hat ja intern (nur) 32 adressierbare Register mit den Adressen von 00h bis 1fh.

Somit sind zu deren Adressierung eigentlich insgesamt nur 5 Adress-Bits notwendig.

Um nun aber das Adressierungsschema bei allen 1-Wire-Slaves mit (großen) internen Speichern gleich zu realisieren, verwendet MAXIM/Dallas grundsätzlich immer 16 Bit große Adressen, ob sie nun komplett benötigt werden oder nicht (so gibt es z.B. den 1-Wire-Speicherchip DS2506 der einen 64 kBit großen internen Speicher besitzt und zu dessen Adressierung werden dann so große Adressen benötigt).

In unserem Fall, beim DS2450er, bedeutet das aber: von den 16 Adressbits werden nur 5 benötigt, die restlichen 11 Bits liegen immer konstant auf log. '0' und müssen aber trotzdem komplett mit zum DS2450er übertragen werden.

Dazu dienen die zwei Bytes **TA1** und **TA2**: TA1 enthält die Adressbits 7 bis 0 (T7:T0), mit dem Wertebereich 0 – 31, und wird als Erstes an den DS2450er gesendet.

TA2 enthält die Adressbits 15 bis 8 (T15:T8), die hier immer den Wert log. '0' haben. Dieses Byte wird als Zweites gesendet.

Daher ergänzen wir jetzt unsere Funktion um:

```
// Page-Adresse übermitteln
ow_wr_byte(p_nr*8);           // TA1 = niederwertig = 08h, 10h
ow_wr_byte(0x00);           // TA2 = höherwertig = 00 (immer fest)
```

Den Wert für TA1 berechnen wir hier einfach aus der an die Funktion übergebenen Page-Nummer 'p_nr': beim Schreiben der Page 1 wird ja der Wert '1' übergeben, somit ergibt sich für TA1 der Wert 8 (≡ Startadresse der Page 1) und für die Page 2 übergibt man ja den Wert 2 und daher ergibt sich in diesem Falle der Start-Speicherwert von 10h für diese Seite.

③:

Nun werden die 8 Datenbytes für die jeweilige Page nacheinander aus dem zuvor beschriebenen Vektor 'ds2450_p[.]' ausgegeben.

Wir beginnen also in unserem Programm mit der for-Schleife und senden das erste Datenbyte aus 'ds2450_p[0]' aus:

```
// Die 8 Bytes aus dem Vektor ds2450_p[.] für die Page aussenden
for (i=0; i<8; i++)
{
    // Byte senden
    ow_wr_byte(ds2450_p[i]);
}
```

Nachdem nun das erste Nutzdatenbyte gesendet wurde, „erscheint“ die **nächste Besonderheit** bei der Datenübertragung zum DS2450er hin:

④:

Der DS2450er berechnet nun (nachdem der CRC-Generator zuvor gelöscht worden ist) aus den bisher vom Master gesendeten Daten, also vom

- Command-Byte 55h,
- von den beiden Adress-Bytes TA1 und TA2 und
- vom ersten vom Master gesendeten Datenbyte

die 16-Bit-CRC-Check-Summe und diese muß jetzt zuerst einmal vom Master abgeholt werden, bevor es mit der Übertragung des zweiten Nutzdatenbytes weiter gehen kann, mit anderen Worten:

Der Master muß als nächstes erst einmal die beiden CRC-Bytes abholen (die 16-Bit-CRC-Summe ist ja in zwei Bytes aufgeteilt), d.h. 2 Bytes müssen aus dem DS2450er auslesen werden !

Somit geht unsere Funktion also wie folgt weiter:

```
// 16-Bit-CRC vom DS2450er zurück lesen
crc1=ow_rd_byte();           // CRC-1
crc2=ow_rd_byte();           // CRC-2
```

Diese beiden CRC-Byte werden zunächst unter den globalen Variablen 'crc1' und 'crc2' abgespeichert und können nachfolgend ausgewertet werden.

Die „Sicherungs-Aktionen“ des DS2450ers sind aber noch lange nicht abgeschlossen, denn ...

⑤:

Der Baustein speichert intern das vom Master empfangene Nutzdatenbyte in der zuvor adressierten Speicherstelle ab.

⑥:

Danach liest der DS2450er den Inhalt dieser Speicherstelle wieder zurück und stellt diesen Wert dem Master für eine Rücklese-Aktion zur Verfügung.

Mit andere Worten: der Master kann nun immer noch nicht das zweite Nutzdatenbyte aussenden, sondern muß erst einmal das zuvor gesendet Byte wieder aus dem DS2450er zurück lesen.

Danach kann der Master nun auf zwei verschiedenen Arten überprüfen, ob ein Fehler bei der Übertragung des ersten Nutzdaten-Bytes aufgetreten ist:

- 1) Er kann intern selber eine 16-Bit-CRC-Checksumme aus den bisher gesendeten Daten berechnen und diese mit der vom DS2450er empfangenen 16-Bit-CRC-Checksumme vergleichen

und / oder

- 2) Er vergleicht das zurück gelesene Nutzdaten-Byte mit dem zuvor ausgesendeten Byte.

Tritt bei einem der beiden Vergleiche ein Fehler auf, so ...

⑦:

bricht der Master die weitere Datenübertragung ab, sendet ein Master-Reset-Signal **⑧**, worauf der DS2450er sich zurück setzt, einen Presence-Impuls aussendet, **⑨**, und die gesamte Datenkommunikation von vorne beginnt.

Daher geht es in unserer Funktion wie folgt weiter:

```
// Kopie d. gesendeten Datums vom DS2450er zurück lesen u. vergl.
zw=ow_rd_byte();
if (zw != ds2450_p[i])          // Fehler
{
    ow_fem(21);                // Fehlermeldung ausgeben
}

// CRC-Überprüfung; entfällt zur Zeit !!
```

Zwei Besonderheiten bzw. Abweichungen vom (vorgeschlagenen) Flussdiagramm-Verlauf haben wir hier programmiert:

- 1) Beim Auftreten eines Fehlers beim Vergleich von gesendeten und wieder zurück empfangen Nutzdaten-Byte geben wir hier die Fehlermeldung Nummer 21 aus und das Programm wird in einer Endlosschleife angehalten.
Zur erneuten Programmausführung ist also ein kompletter Master-Reset notwendig.

- 2) Ein 16-Bit-CRC-Check auf der Master-Seite wird zur Zeit hier nicht ausgeführt.

1 0:

Ist das Ende des gesamten internen Speichers des DS2450ers oder das Ende der zu beschreibenden Page erreicht worden, so muß der Master ebenfalls einen Reset- Impuls senden und über die Punkte **8** und **9** wird der Kommunikationszyklus ordnungsgemäß beendet. Sollen dagegen noch weitere Nutzdaten-Bytes für die ausgewählte Page gesendet werden, so geht es bei

1 1:

weiter: der Master kann jeder Zeit, auch ohne dass ein Fehler vorliegt, durch Senden eines Reset- Impulses die Kommunikation mit dem DS2450er beendet. Über den Punkt **9** bestätigt der DS2450er dann diesen vorzeitigen Abbruch. Die bisher übertragenen Daten bleiben dann im DS2450er erhalten. Das wollen wir hier aber nicht durchführen. Also geht es für uns im Flussdiagramm-Ablauf weiter mit ...

1 2:

Der DS2450er erhöht nun seinen internen Adresszähler und adressiert dadurch den nächsten Speicherplatz, in den der Master nachfolgend etwas einschreiben kann.

Gleichzeitig wird diese neue Adresse in den internen CRC-Generator des DS2450er geladen (**1 3**) und dient so als Grundlage für die nachfolgende, neu zu berechnende CRC-Check-Summe.

Der Master kann also jetzt (endlich) das nächste Nutzdaten-Byte senden, unser Programm muß daher zum Punkt **3** verzweigen, also zum Anfang der for-Schleife.

Dort erwartet uns nun aber **eine kleine Änderung** bei der CRC-Berechnung durch den DS2450er (**4**): die Sicherungssumme wird ab jetzt immer nur aus den beiden Adressbytes der neu beschriebenen Speicherstelle und aus dem zugehörigen Nutzdaten-Byte gebildet, das Command-Byte fällt also aus der Bildung der CRC-Summe heraus (es wurde ja schon bei der allerersten CRC-Berechnung mit verwendet).

Der DS2450er generiert so **nach jedem** geschriebenen Byte ein CRC-Summe und stellt diese dem Master zur Verfügung.

Aber es kommt noch eine weitere **Besonderheit** bei der Kommunikation mit dem DS2450er auf uns zu:

Wenn alle 8 Nutzdaten-Bytes gesendet worden sind, wenn also die for-Schleife abgearbeitet worden ist, so beendet der Master diesen gesamten Kommunikationszyklus über den Punkt **1 0** bzw. über den Punkt **1 1**, d.h. der Master sendet einen abschließenden Reset-Impuls, auf den der Slave als Quittung mit einem Presence-Impuls (**9**) antwortet.

Danach kann ein komplett neuer 1-Wire-Zyklus beginnen.

Wir beenden unsere Page-Write-Funktion also mit:

```

    }                // Ende der for-Schleife

    // Fertig mit dem Senden einer Page:
    // Master sendet abschließenden Reset-Impuls
    ow_pres_check(22);

}                // Ende der gesamten Funktion

```

Tritt hierbei noch ein Fehler auf, so wird dieser mit der Textmeldung Nummer 22 gemeldet.

Zusammen gefasst sieht unsere Funktion nun wie folgt aus:

```

/**/ Schreiben von 8 Byte-Daten in eine DS2450er-Page; sinnvoll nur für Page 1 und 2 ***/
/**/ Inhalt des 8-Byte-Arrays ds2450_p[] wird in die angegebene Page geschrieben. ***/
/**/ LSByte der Page aus der Array-Stelle 0, usw. ***/
/**/ p_nr = 1 oder 2 ***/

void ds2450_write_page(unsigned char p_nr)
{
    unsigned char i,zw;

    // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
    ow_pres_check(20);

    /**/ *****/
    /**/ Werte in übergebene Page einschreiben ***/
    /**/ *****/

    // Adressierung des jeweils als aktiv ausgewählten Slaves
    ow_match();

    // Schreib-Befehl senden: "WRITE Memory" = 0x55
    ow_wr_byte(0x55);

    // Page-Adresse übermitteln
    ow_wr_byte(p_nr*8);                // TA1 = niederwertig = 08h, 10h
    ow_wr_byte(0x00);                // TA2 = höherwertig = 00 (immer fest)

    // Die 8 Bytes aus dem Vektor ds2450_p[] für die Page aussenden
    for (i=0; i<8; i++)
    {
        // Byte senden
        ow_wr_byte(ds2450_p[i]);

        // 16-Bit-CRC vom DS2450er zurück lesen
        crc1=ow_rd_byte();                // CRC-1
        crc2=ow_rd_byte();                // CRC-2

        // Kopie dd gesendeten Datums vom DS2450er zurücklesen und vergl.
        zw=ow_rd_byte();
        if (zw != ds2450_p[i])                // Fehler
        {
            ow_fem(21);                // Fehlermeldung ausgeben
        }
    }
}

```

```

        // CRC-Überprüfung; entfällt zur Zeit !!

    }

    // Fertig mit dem Senden einer Page:
    // Master sendet abschließenden Reset-Impuls
    ow_pres_check(22);
}

```

Wie bereits zuvor erwähnt, beschreiben wir in unserer API immer eine DS2450er Seite komplett.

Es gibt dazu jedoch eine **Ausnahme**, und zwar bei der (Grund)Initialisierung des DS2450er: in Kapitel 7.1 haben wir erwähnt, dass dazu u.a. in die Speicherstelle mit der Adresse 1ch auf der Page 3 (s. Abb.7.1.10) der Wert 40h geschrieben werden muß.

Wir müssen also gezielt etwas in eine einzelne Speicherstelle einschreiben.

Das ist aber nun keine Problem mehr, denn wir können diesen Programmteil ganz einfach wie folgt aus der vorherigen Funktion extrahieren:

```

/*****
/*** Noch notwendigen Wert in Page 3 einschreiben unter Adr: 1ch ****/
/*****/

// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(25);

// Adressierung des jeweils als aktiv ausgewählten Slaves
ow_match();

// Schreib-Befehl senden: "WRITE Memory" = 0x55
ow_wr_byte(0x55);

// Register-Start-Adresse übermitteln: TA1 und TA2
ow_wr_byte(0x1c);          // TA1 = niederwertig = 1c
ow_wr_byte(0x00);          // TA2 = höherwertig = 00

// 1. Datenbyte senden für Reg-Adr: 1ch
ow_wr_byte(0x40);          // Datum = 40h

// 16-Bit-CRC und Datum vom DS2450er zurück lesen
crc1=ow_rd_byte();          // CRC-1
crc2=ow_rd_byte();          // CRC-2

// Kopie des gesendeten Datums vom DS2450er zurücklesen und vergleichen
zw=ow_rd_byte();
if (zw != 0x40)              // Fehler
{
    ow_fem(26);              // Fehlermeldung ausgeben
}

// CRC-Überprüfung; entfällt zur Zeit !!

// Fertig mit dem Senden dieses Bytes
// Master sendet abschließenden Reset-Impuls
ow_pres_check(29);

```

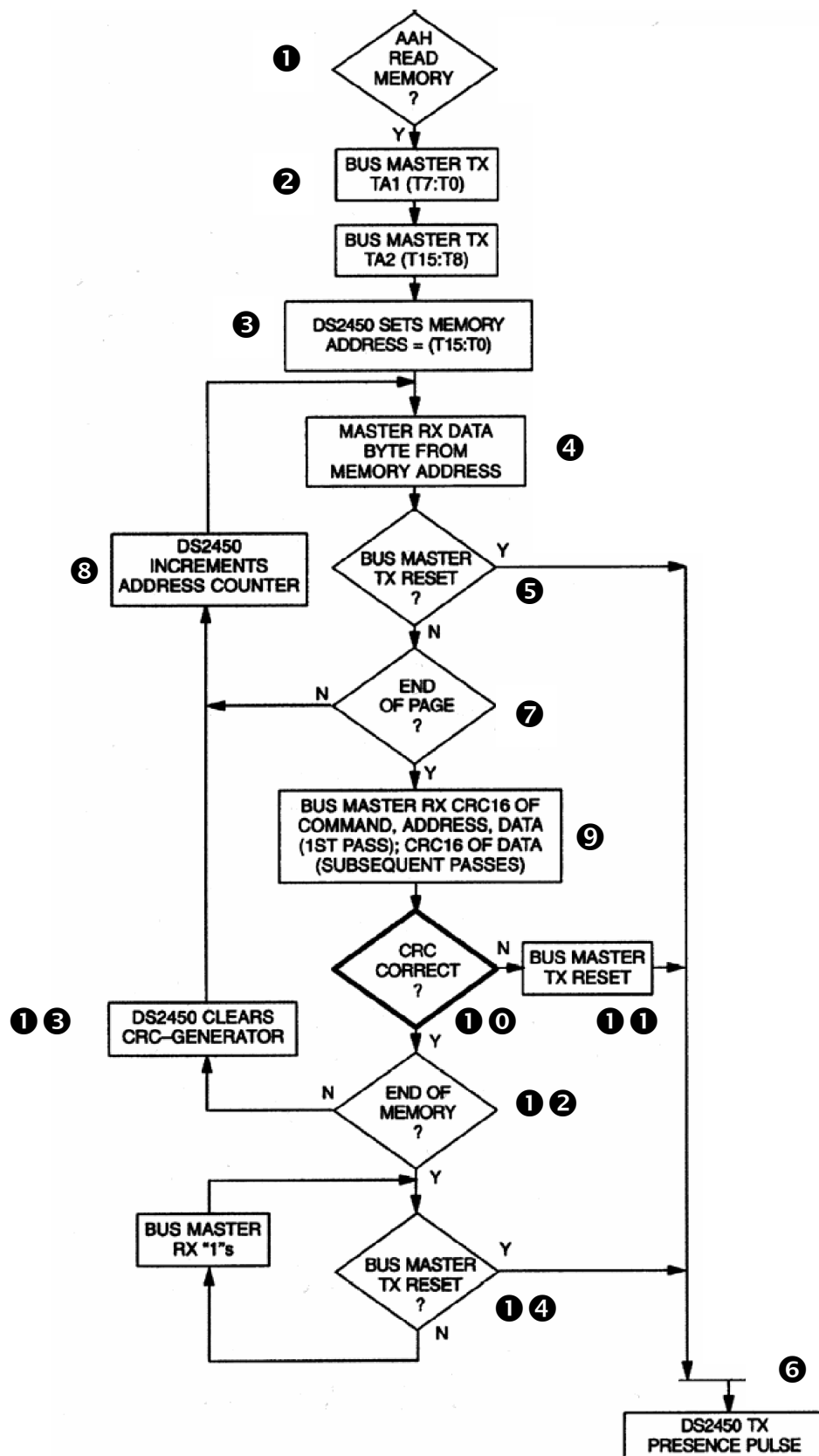
Wir werden dieses Programm-Segment so später in unsere Initialisierungs-Funktion für den DS2450er einbauen.

Kommunikationszyklus 2: Das Auslesen einer DS2450er-Page

Das Auslesen einer Seite ist nur sinnvoll für die internen Pages 0, 1 und 2, bzw. für die Bytes mit den Adressen 00h .. 17h, s. Kap.7.1:

- Page 0: Wandlungsdaten der vier Kanäle
- Page 1: Kontroll-/Status-Daten der vier Kanäle
- Page 2: Eingestellte Grenzwertdaten für die vier Kanäle

Im Datenblatt des DS2450er findet man zum Auslesen einer Page das folgende Flussdiagramm, **Abb.7.2.2:**



(TX ≡ sendet, RX ≡ empfängt)

Abb.7.2.2: Flussdiagramm zum Auslesen aus einer Page

Das Verständnis dieses Ablaufs ist nach den vorherigen Erläuterungen zum Write-Zyklus kein größeres Problem mehr.

Die DS2450er-API-Funktion, die wir zum Auslesen einer Seite benötigen, heißt `'ds2450_read_page(...)'` und wird nachfolgend parallel zu unseren Ausführungen entwickelt. Anzumerken sei hier noch: die vom DS2450er ausgelesenen Daten einer Page werden im globalen Array `'ds2450_p[.]'` abgespeichert und können von dort aus dann entsprechend im main-Programm weiter verarbeitet werden.

Bei der Speicherung gilt, dass das niederwertigste Byte einer Page im Array-Element 0 abgelegt wird.

Zu Anfang der Funktion werden wieder die „einleitenden“ 1-Wire-Befehle zur Adressierung des DS2450ers abgewickelt und dann geht's los mit der eigentlichen Auslese-Routine:

①:

Das Befehlswort zum Auslesen einer Page lautet 55h.

②:

Danach sendet der Master die Adresse der Speicherstelle, die nun zuerst gelesen werden soll (≡ Startadresse): analog wie beim Write-Zyklus werden die beiden Bytes TA1 und TA2 an den DS2450er übertragen.

③:

Der DS2450er übernimmt intern diese Adresse, als Adresse der ersten Speicherstelle, die jetzt ausgelesen werden kann.

Unsere neue DS2450er-API-Funktion beginnt daher wie folgt:

```

/** Auslesen von 8 Byte-Daten aus einer DS2450er-Page, für Pages 0 - 2 möglich.    ***/
/** Ausgelesene Bytes werden in 8-Byte-Array ds2450_p[.] abgespeichert und zwar: ***/
/** LSByte der Page in Array-Stelle 0, usw.                                     ***/
/** p_nr = 0, 1, 2                                                            ***/

void ds2450_read_page(unsigned char p_nr)
{
    unsigned char i;

    // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
    ow_pres_check(23);

    /***/
    /** Read Memory ***/
    /***/

    // Adressierung des jeweils als aktiv ausgewählten Slaves
    ow_match();

```

```
// Speicher lesen-Befehl senden: "Read Memory" = 0xaa
ow_wr_byte(0xaa);

// Register-Start-Adresse übermitteln: TA1 und TA2
ow_wr_byte(p_nr*8);           // TA1 = niederwertig = 00h, 08h, 10h
ow_wr_byte(0x00);           // TA2 = höherwertig = immer 00
```

④:

Nun werden die 8 Datenbytes der jeweiligen Page nacheinander vom Master aus dem DS2450er ausgelesen und im Vektor 'ds2450_p[]' abgespeichert. Die entsprechende interne Adressierung der Speicherstellen übernimmt dabei der DS2450er selber. Der Master muß also nur einmalig, am Anfang des Kommunikationszyklusses, die Speicher-Start-Adresse übermitteln. Wir beginnen in unserer Funktion mit der for-Schleife, lesen das erste Datenbyte aus dem DS2450er aus und speichern es in 'ds2450_p[0]' ab:

```
// Auslesen von 8 Byte hintereinander aus DS2450er
// = eine Page auslesen !
for (i=0; i<8; i++)
{
    ds2450_p[i] = ow_rd_byte();
}
```

⑤:

Über einen vorzeitig gesendeten Master-Rest-Impuls, auf den der DS2450er mit einem Presence-Impuls (⑥) antwortet, kann der Auslesvorgang jeder Zeit beliebig abgebrochen und danach ein neuer 1-Wire-Kommunikationszyklus gestartet werden. Das machen wir hier allerdings nicht. Es geht also weiter bei ...

⑦:

Der Master bzw. der DS2450er selber überprüfen nun, ob bereits das Ende einer Seite erreicht worden ist. Ist das noch nicht der Fall so fährt man fort bei ...

⑧:

Der DS2450er erhöht intern automatisch den Adresszähler und adressiert somit die nächste Speicherstelle auf der Page zum Auslesen.

Wir gelangen somit zurück nach ④ und der Master liest daher nacheinander alle 8 Bytes einer Page aus.

Ist nun das Ende einer Seite erreicht so kommen wir über ⑦ nach ...

⑨:

Erst jetzt stellt der DS2450er die beiden Datensicherungs-CRC-Bytes zur Verfügung, die gebildet werden aus (zuvor wird der CRC-Generator gelöscht):

- dem Command-Byte aah,
- den vom Master übertragenen Start-Adress-Bytes TA1 und TA2 und
- den bis hier hin gesendeten 8 Nutzdatenbytes.

Diese CRC-Summe wird also immer erst dann gesendet, wenn das Ende einer Seite erreicht worden ist.

Um diese CRC-Sicherung erhalten und auswerten zu können, muß man also **immer** alle Bytes bis zum Ende einer Page auslesen, wobei man allerdings die Auslese-Start-Adresse durchaus in an eine beliebige Stelle innerhalb einer Seite setzen kann.

Das ist für uns aber kein Problem, da wir ja sowieso immer eine ganze Seite, von Anfang an, auf einmal komplett auslesen.

Wir müssen daher jetzt in unserer Funktion die beiden CRC-Bytes aus dem DS2450er auslesen und auswerten (aus wir zur Zeit allerdings nicht machen):

```

}                // Ende der for-Schleife: eine Page komplett eingelesen

// Da hier End of page:
// 16-Bit-CRC vom DS2450er zurück lesen
crc1=ow_rd_byte();           // CRC-1
crc2=ow_rd_byte();           // CRC-2

// CRC-Überprüfung; entfällt zur Zeit !!

```

⑩:

Ergibt die CRC-Überprüfung einen Fehler, so sendet der Master einen Reset-Impuls (**①①**) auf den der DS2450er mit einem Presence-Impuls (**⑥**) antwortet und damit ist dann dieser Kommunikationszyklus beendet.

Was im Flussdiagramm nicht steht, aber im begleitenden Text im Datenblatt: nach dem Auslesen der beiden CRC-Bytes löscht der DS2450er den CRC-Generator und beginnt mit der Bildung der neuen CRC-Check-Summe.

Diese neue CRC-Sicherung wird dann aber nur über die neu ausgelesenen Daten gebildet.

Und nicht mehr über das Command-Byte und auch nicht mehr über die beiden Start-Adress-Bytes, denn diese Informationen wurden ja schon zuvor mit der allerersten CRC-Summe abgesichert.

Ergibt die CRC-Überprüfung auf der Master-Seite keinen Fehler, so geht es weiter bei ...

⑪:

Ist nun das Ende des gesamten internen DS2450er-Speichers **noch nicht** erreicht (also die Speicherstelle mit der Adresse 1fh am Ende der Page 3 wurde noch nicht ausgelesen), so löscht der DS2450er den CRC-Generator **❶❸** und der DS2450er macht weiter bei **❸**.

Ist allerdings das Ende des gesamten internen Speichers des DS2450ers erreicht, d.h. wurde die letzte Speicherstelle mit der Adresse 1fh auf der Page 3 ausgelesen, so geht der Verlauf weiter bei **❶❹**.

An diesen Punkt gelangen wir in unseren Applikationen aber nie, da wir ja nie die Page 3 auslesen. Denn auf der Page 3 sind keine relevanten Informationen für die DS2450er-Anwendung enthalten. Wir lesen vielmehr immer nur die Inhalte der Pages 0, 1 oder 2 aus. Damit kommen wir mit unserem Ablauf in dieser API-Funktion nur zu den Punkten **❷**, **❹**, **❶❶**, **❶❷** bzw. **❶❸** und **❶❹**.

Deshalb befinden wir uns nach dem Auslesen einer ganzen Seite (und nach einer fehlerfreien CRC-Überprüfung) jetzt beim Punkt **❶❸**, von dem aus es weiter nach **❸** geht, d.h. der DS2450er adressiert die erste Speicherstelle der nächsten Page, die wir aber gar nicht mehr auslesen wollen.

Wir müssen also jetzt den Kommunikationszyklus mit dem DS2450er korrekt beenden, d.h. abbrechen. Und das machen wir mit einem Master-Reset-Impuls an der Stelle **❺** (das Auslesen des nächsten Bytes an der Stelle **❹** ignorieren wir dabei, wir brechen den Zyklus einfach mit Reset ab).

Der DS2450er antwortet darauf hin mit seinem Presence-Impuls (**❻**) und das Auslesen einer Page ist so einwandfrei beendet.

Wir schließen also unsere API-Funktion ab mit:

```
// Abschließender Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(24);

} // Ende der Funktion
```

Die acht Datenbytes der Page stehen jetzt im Vektor 'ds2450_p[.]' zur Verfügung und können im weiteren Verlauf von main ausgewertet werden.

Zusammen gefasst sieht die zweite Funktion der DS2450er-API jetzt so aus:

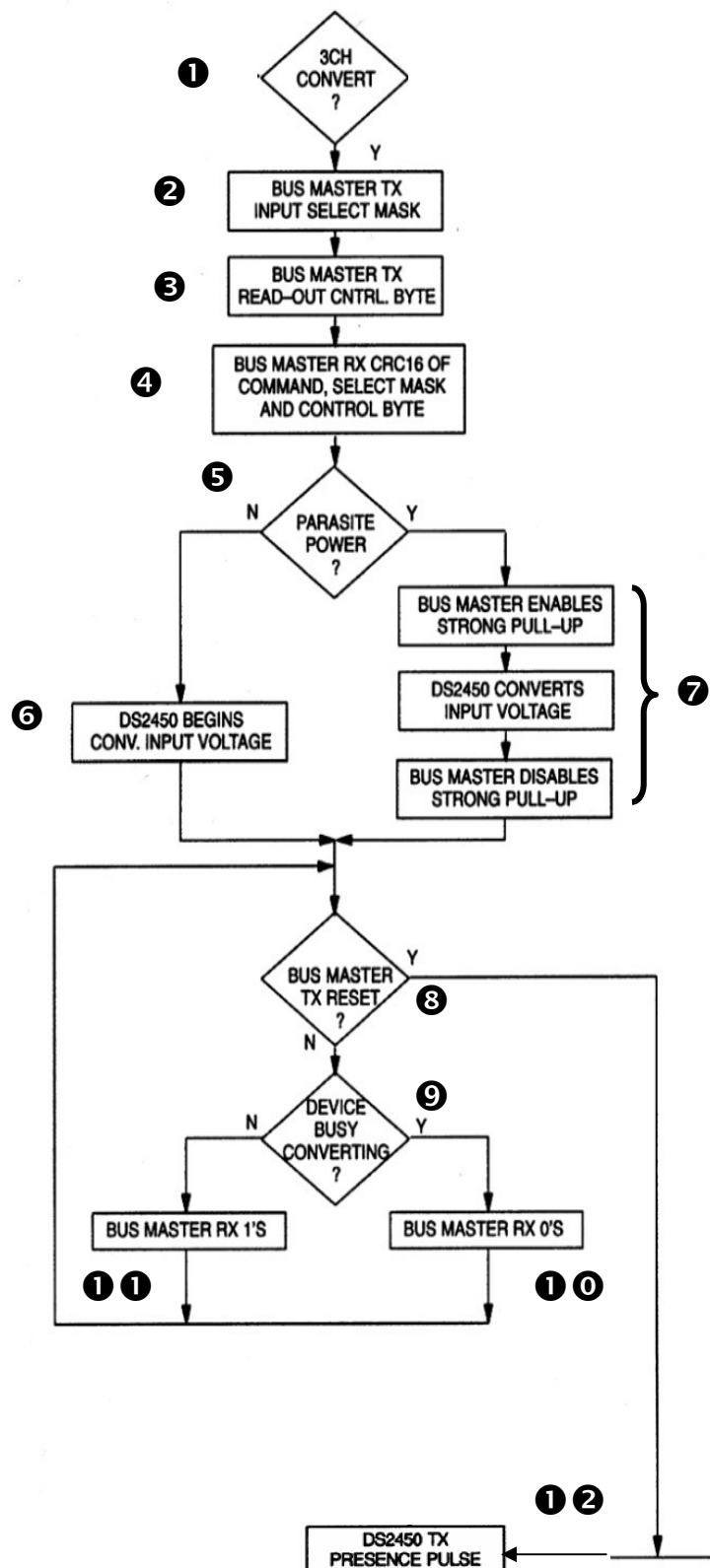
```
/** Auslesen von 8 Byte-Daten aus einer DS2450er-Page, für Pages 0 - 2 möglich. */
/** Ausgelesene Bytes werden in 8-Byte-Array ds2450_p[.] abgespeichert und zwar: */
/** LSByte der Page in Array-Stelle 0, usw. */
/** p_nr = 0, 1, 2 */
void ds2450_read_page(unsigned char p_nr)
{
    unsigned char i;

    // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
    ow_pres_check(23);
```

```
/******  
/*** Read Memory  ***/  
/******  
  
// Adressierung des jeweils als aktiv ausgewählten Slaves  
ow_match();  
  
// Speicher lesen-Befehl senden: "Read Memory" = 0xaa  
ow_wr_byte(0xaa);  
  
// Register-Start-Adresse übermitteln: TA1 und TA2  
ow_wr_byte(p_nr*8);          // TA1 = niederwertig = 00h, 08h, 10h  
ow_wr_byte(0x00);          // TA2 = höherwertig = immer 00  
  
// Auslesen von 8 Byte hintereinander aus DS2450er  
// = eine Page auslesen !  
for (i=0; i<8; i++)  
{  
    ds2450_p[i] = ow_rd_byte();  
}  
  
// Da hier End of page:  
// 16-Bit-CRC vom DS2450er zurück lesen  
crc1=ow_rd_byte();          // CRC-1  
crc2=ow_rd_byte();          // CRC-2  
  
// Abschließender Master-Reset-Impuls u. Abfrage: Slave presence  
ow_pres_check(24);  
  
// CRC-Überprüfung; entfällt zur Zeit !!  
  
}
```

Kommunikationszyklus 3: Start der A/D-Wandlung

Den Start und den Ablauf einer A/D-Wandlung zeigt das Flussdiagramm in der **Abb.7.2.3**:



(TX ≡ sendet, RX ≡ empfängt)

Abb.7.2.3: Flussdiagramm zum Start und zum Ablauf der A/D-Wandlung

Auch dieser letzte Kommunikationszyklus beinhaltet vom Verständnis her keinerlei große Hürden mehr.

Die hier jetzt entwickelte DS2450er-API-Funktion, mit der wir alle vier Kanäle aktivieren, die anliegenden Spannungen wandeln lassen und dann die Wandlungswerte auslesen, heißt `ds2450_convert_all()`.

Anzumerken sei hier noch: die vom DS2450er ausgelesenen Ergebnis-Daten (der Page 0) werden im globale Array `ds2450_p[.]` abgespeichert und können von dort aus dann entsprechend im main-Programm weiter verarbeitet werden.

Bei der Speicherung gilt, dass das niederwertigste Byte vom Kanal A im Array-Element 0 und das höherwertigste Byte vom Kanal D im Array-Element 7 abgelegt wird.

Zu Anfang der Funktion werden wieder die „einleitenden“ 1-Wire-Befehle zur Adressierung des DS2450ers abgewickelt und dann geht's los mit der eigentlichen Kern-Routine:

❶:

Das Befehlswort zum Start einer Wandlung lautet 3ch.

❷:

Nun muss der Master angeben, welche der vier Kanäle gewandelt werden sollen.

Dazu dient das Steuerbyte **Input Select Mask**, Abb.7.2.4:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
"don't care"				D	C	B	A

Abb.7.2.4: Die Auswahl des zu wandelnden Kanals

Soll ein Kanal gewandelt werden, so muss das entsprechende Kanal-Bit auf log. '1' gesetzt werden.

Alle so aktivierten Kanäle werden dann vom DS2450er nacheinander gewandelt, wobei mit dem Kanal A begonnen wird. Die nicht ausgewählten Kanäle werden dabei übersprungen.

Da wir in unserer Applikation auf jeden Fall alle Kanäle wandeln wollen, hat die **Input Select Mask** hier den Wert 0fh.

❸:

Das nun folgende **Read Out Control Byte** ist in unserer Anwendung nicht von Bedeutung und wird daher fest auf 00h gesetzt (nähere Informationen hierzu finden sich im Datenblatt des DS2450ers).

Damit sind alle Anweisungen an den DS2450er zur Kanal-Einstellung und zum Start der Wandlung abgeschlossen und ...

④:

Der DS2450er bildet nun intern, wie gewohnt, die 16-Bit-CRC-Check-Summe aus:

- Command Byte,
- Input Select Mask und
- Read Out Control Byte

und stellt diese dem Bus-Master zur Verfügung, d.h. als nächstes muss der Master erst einmal diese beiden Sicherungsbytes auslesen.

Somit ergibt sich für den Start unserer neuen DS2450er-API-Funktion:

```
/** Start der Wandlung und Auslesen für alle 4 Kanäle eines einzelnen
DS2450ers am Bus */
```

```
void ds2450_convert_all(void)
{
    /***/
    /** Start Convert - Anweisung senden */
    /***/
    // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
    ow_pres_check(27);

    // Adressierung des jeweils als aktiv ausgewählten Slaves
    ow_match();

    // Start-Wandlung-Befehl senden: "Convert" = 0x3c
    ow_wr_byte(0x3c);

    // Senden der input-select-Maske: alle Kanäle wandeln
    ow_wr_byte(0x0f);

    // Senden read out control byte: hier fest auf 00h
    ow_wr_byte(0x00);

    // 16-Bit-CRC vom DS2450er zurück lesen
    crc1=ow_rd_byte();           // CRC-1
    crc2=ow_rd_byte();           // CRC-2

    // CRC-Überprüfung; entfällt zur Zeit !!
}
```

⑤:

An dieser Stelle muss nun unterschieden werden, ob der DS2450er parasitär oder lokal versorgt wird: beim parasitären Zweig (⑦) muss ja der 'Strong-Pull-Up-Widerstand' zur Wandlung ein- und wieder ausgeschaltet werden.

Der DS2450er wartet also nach dem Aussenden des letzten Bits der CRC-Check-Summe (Punkt ④) ca. 10 µs bis er mit der Wandlung der Kanäle beginnt, mit Kanal A zuerst (wenn dieser freigeschaltet worden ist).

Diese Zeit steht dem Master zur Verfügung, um bei parasitärer Speisung den auf jeden Fall benötigten 'Strong Pull-Up'-Widerstand rechtzeitig einzuschalten (s. Zweig Nr. ⑦).

Wir speisen in unserer Applikation (auf dem PT-1-Wire ExBo) den DS2450er aber lokal (der DS2450er wartet trotzdem die 10 µs) und daher geht es für uns weiter bei ...

⑥:

Der DS2450er beginnt jetzt mit den Wandlungen der Eingangsspannungen und wir gelangen zu ...

⑧:

Da wir zur Zeit noch keinen Reset aussenden, überspringen wir diesen Punkt und geht es sofort weiter mit ...

⑨:

Wir warten nun im Programmablauf solange, bis der DS2450er alle zuvor aktivierten Kanäle mit der gewünschten Referenzspannung und mit der ausgewählten Auflösung gewandelt hat. Dieses „Warten“ ist eigentlich recht einfach durchzuführen: wir lesen dazu den DS2450er permanent aus und solange er noch wandelt, solange er also noch beschäftigt ist, sendet der DS2450er lauter log. '0' en zurück, ① ①, wir empfangen also immer den Wert 00h.

Hat der DS2450er dagegen all seine Wandlungen beendet und liegen die Endergebnisse definitiv und stabil vor, so sendet er jetzt lauter log. '1' en zurück, ① ①. Wir empfangen daher nun fffh.

Wir fügen nun also ganz einfach in unsere Funktion ein:

```
// Warten, bis Wandlung aller vier Kanäle fertig ist
while(ow_rd_byte() != 0xff);
```

(Alternativ kann man hier auch eine Wartezeit im Master-Programm einbauen, die der Wandlungszeit entspricht. Zur Berechnung dieser benötigten Zeit: s. unsere nachfolgende Betrachtungen).

Damit ist die Wandlung abgeschlossen und wir könnten die Ergebnisse aus der DS2450er-Page 0 auslesen, wenn jetzt nicht **noch eine weitere Besonderheit** beim DS2450er zu beachten wäre:

Wir befinden uns ja an der Stelle ① ① im Flussdiagramm, es geht also für den DS2450er weiter bei ⑧, dann nach ⑨, dann nach ① ①, dann wieder nach ⑧, usw., usw..

Mit anderen Worten: der DS2450er befindet sich in einer kritischen Endlosschleife,

- er reagiert auf keine Master-Anweisungen mehr
- und vor allen Dingen: der Master liest immer nur log. '1'en aus dem DS2450er aus (❶❶).

Um aus diesem „Teufelskreis“ wieder herauszukommen, gibt es nur einen einzigen Ausweg: der Master muss auch diesen gesamten Kommunikationszyklus definiert mit einem Reset-Impuls abschließen: in dem Falle erfolgt dann der Aussprung bei Punkt ❸ hin nach ❶❷ und von dort geht es weiter zur Antwort des DS2450er in Form eines Presence-Impulses. Damit ist dieser Zyklus einwandfrei beendet und ein neuer Ablauf kann beginnen. Wir ergänzen also in unserer Funktion:

```
// Abschließender Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(28);
```

Nun bleibt nur noch übrig, die gewandelten Messdaten aus der DS2450er-Page 0 auszulesen und dazu können wir jetzt sehr gut unsere bereits fertige API-Funktion zum allgemeinen Auslesen von Page-Daten benutzen:

```
/******
/** Wandlungsergebnisse aus Page 0 auslesen u. in ds2450_p[.] abspeichern */
*****
ds2450_read_page(0);
```

Nach dem Aufruf dieser Funktion liegen die Ergebnisdaten von allen vier Kanälen im globalen Vektor 'ds2450_p[.]' vor und können vom main-Program weiter verarbeitet werden.

Fassen wir abschließend alle einzelnen 'C'-Segmente zusammen, so erhält man als Ganzes:

```
/** Start der Wandlung und Auslesen für alle 4 Kanäle eines einzelnen
DS2450ers am Bus */
```

```
void ds2450_convert_all(void)
{
    /******
    /** Start Convert - Anweisung senden */
    *****
    // Start mit Master-Reset-Impuls u. Abfrage: Slave presence
    ow_pres_check(27);

    // Adressierung des jeweils als aktiv ausgewählten Slaves
    ow_match();

    // Start-Wandlung-Befehl senden: "Convert" = 0x3c
    ow_wr_byte(0x3c);

    // Senden der input-select-Maske: alle Kanäle wandeln
    ow_wr_byte(0x0f);
```



```

// Senden read out control byte: hier fest auf 00h
ow_wr_byte(0x00);

// 16-Bit-CRC vom DS2450er zurück lesen
crc1=ow_rd_byte();           // CRC-1
crc2=ow_rd_byte();           // CRC-2

// CRC-Überprüfung; entfällt zur Zeit !!

// Warten bis Wandlung aller vier Kanäle fertig ist
while(ow_rd_byte() != 0xff);

// Abschließender Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(28);

/****
*** Wandlungsergebnisse aus Page 0 auslesen u. in ds2450_p[,] abspeichern ***
****/
ds2450_read_page(0);
}

```

Einen letzten Punkt haben wir bisher noch gar nicht angesprochen, und das ist die Dauer der Wandlungszeit, d.h. wie lange benötigt der DS2450er eigentlich für eine Wandlung oder ganz einfach ausgedrückt:

Wie schnell ist der DS2450er eigentlich, wie groß ist die 'Conversion Time' ?

Hierzu findet sich im Datenblatt des DS2450er eine recht einfache Berechnungsformel: der DS2450er benötigt für die Wandlung eines jeden einzelnen Bits 60 .. 80 µs.

Wir rechnen sicherheitshalber mit dem 'Worst Case'-Wert von 80 µs/Bit.

Dazu kommt noch einmalig eine Offset-Zeit von 160 µs bei der Ausführung von jedem einzelnen Convert Command, d.h. diese 160 µs fallen einmalig an egal ob mit einem Convert Command nur ein Kanal oder ob alle vier Kanäle gewandelt werden.

Damit ergibt sich:

$$\text{Conversion Time} = (\text{Anzahl der insges. gewandelten Bits}) * 80 \mu\text{s} + 160 \mu\text{s}$$

Beispiel:

Wandlung aller vier Kanäle mit 10 Bit Auslösung:

$$\text{Conversion Time} = (4 * 10) * 80 \mu\text{s} + 160 \mu\text{s} = 3,36 \text{ ms}$$

Damit sind wir am Ende des Kapitels angelangt und haben die drei wesentlichsten Grund-API-Funktionen zum Betrieb des DS2450ers erstellt.

Es fehlt eigentlich nur noch eine wichtige Funktion, nämlich diejenige zur Anfangsinitialisierung des Bausteins, d.h. die Funktion mit der die Referenzspannung, die Auflösung, etc. für jede Kanal eingestellt wird.

Diesen Programmteil werden wir im Kapitel 7.4 noch erstellen, zusätzlich mit den bereits angesprochenen neuen Funktionen zur verbesserten Adressierung eines 1-Wire-Slaves. Zunächst aber noch einmal einen kleinen, genaueren Blick auf die Erzeugung der 16-Bit-CRC-Check-Summe beim DS2450er.

7.4 Die DS2450er-API

Zum praktischen Betrieb des DS2450er haben wir bereits die drei wichtigen Funktionen

- Einschreiben von Daten in eine DS2450er-Page: **ds2450_write_page(...)**
- Auslesen von Daten aus einer DS2450er-Page: **ds2450_read_page(...)**
- Start der Wandlung und Auslesen der Werte: **ds2450_convert_all()**

entwickelt, so dass im Kern eigentlich nur noch eine Funktion fehlt: die zur Anfangs- bzw. Grundinitialisierung des Baustein.

In Kapitel 7.1 wurde beschrieben, was alles für die Initialisierung notwendig ist (Festlegungen auf der Page 1 und der Page 2). Im Wesentlichen sind dieses:

- Auswahl der Referenzspannung für jeden Kanal
- Auswahl der Auflösung für jeden Kanal
- Einstellung der oberen und unteren Alarmgrenzwerte für jeden Kanal.

Da wir in unserer ersten Applikation nicht mit einer Grenzwertüberwachung arbeiten, konzentrieren wir uns bei den Einstellungen nur auf die Festlegung der Referenzspannung und auf die Festlegung der Auflösung für jeden der vier Kanäle.

Und hierbei wenden wir eine kleine praxisorientierte Vereinfachung an: da eine menügeführte Eingabe aller Variationsmöglichkeiten recht aufwendig zu programmieren und in der Praxis, beim Programmstart, doch recht umständlich in der Handhabung ist (wenn jedes Mal nach einem Reset des Programms erst einmal der komplette Einstellungsprogrammteil für alle Kanäle durchlaufen und abgearbeitet werden muss), arbeiten wir hier mit so genannten **Initialisierungsvektoren** in denen wir die wichtigsten Parameter festlegen und die dann einfach beim Aufruf der Initialisierungsfunktion verarbeitet werden und die Einstellungsinformationen für den DS2450er liefern.

Ein einfaches **Beispiel** soll das verdeutlichen:

Ein solcher Initialisierungsvektor besteht in unserem Falle aus 8 unsigned char –Werten, er ist also ein unsigned char-Array mit 8 Elementen. In diese Felder tragen wir nun die gewünschten Werte für die Kanal-Auflösungen und die Kanal-Referenzspannungen ein, wobei gelten soll, **Tab.7.4.1:**

Element des Init.-Vektors	Inhalt
0	Auflösung Kanal A
1	Auflösung Kanal B
2	Auflösung Kanal C
3	Auflösung Kanal D
4	Referenzspannung Kanal A
5	Referenzspannung Kanal B
6	Referenzspannung Kanal C
7	Referenzspannung Kanal D

Tab.7.4.1: Der Inhalt des Initialisierungsvektors

Der Wertebereich für die Auflösungen geht von 0 – 16, und es bedeuten:

0 \equiv 16 Bit-, 1 \equiv 1 Bit-, 15 \equiv 15 Bit-Auflösung des Kanals.

Bei den Festlegungen der Referenzspannungen gilt: 1 \equiv 5,12 V und 0 \equiv 2,56 V.

Man kann daher, z.B. am Anfang des Programms, als globale, vorbelegte Variable definieren:

```
code unsigned char ds2450_init_v1[8] = {8,8,10,10,1,1,1,1};
```

und legt damit fest:

- der Initialisierungsvektor heißt hier 'ds2450_init_v1',
- Kanal A und B werden mit 8 Bit Auflösung gewandelt,
- Kanal C und D werden mit 10 Bit Auflösung gewandelt,
- die Referenzspannung für alle Kanäle ist 5,12 V.

Durch den Zusatz 'code' vor der Datentypfestlegung wird erreicht, dass diese Variable im Programmspeicher des Mikrocontrollers abgelegt wird und das hat (bei uns) folgende Gründe: zum einen müssen diese Werte nicht mehr verändert werden, können also fest abgelegt werden und zum anderen hat unser verwendeter Mikrocontroller zwar einen 64 kByte großen Programmspeicher ON-Chip aber „nur“ 2 kByte Datenspeicher ON-Chip.

Im Programmspeicher ist daher reichlich viel Platz für die Abspeicherung solcher Initialisierungsdaten und der knappe Datenspeicherplatz bleibt frei für die Anwendungsvariablen.

Nun lassen sich verschiedene solche Initialisierungsvektoren „anlegen“, die dann bei Bedarf an die Initialisierungsfunktion übergeben werden und somit für die gewünschte, korrekte Einstellung des DS2450er sorgen.

In unserer Applikation haben wir z.B. als globale Variablen am Anfang des Programms festgelegt:

Globale Variablen, II

```

/**/ Unterschiedliche Initialisierungs-Vektoren für den DS2450er          /**/
/**/ Ablage der Werte im Programmspeicher-Bereich !                      /**/
/**/ Aufbau: Aufl. Kanal A, Aufl. Kanal B, Aufl. Kanal C, Aufl. Kanal D    /**/
/**/      Ref.Spg. Kanal A, Ref.Spg. Kanal B, Ref.Spg. Kanal C, Ref.Spg. Kanal D  /**/
code unsigned char ds2450_init_v0[8] = {8,8,8,8,1,1,1,1};
code unsigned char ds2450_init_v1[8] = {8,8,10,10,1,1,1,1};
code unsigned char ds2450_init_v2[8] = {10,10,10,10,1,1,1,1};
code unsigned char ds2450_init_v3[8] = {2,3,4,5,1,1,1,1};
code unsigned char ds2450_init_v4[8] = {16,16,16,16,1,1,1,1};
code unsigned char ds2450_init_v5[8] = {8,8,8,8,0,0,0,0};

// Auflösungsweite (2er-Potenzen)
code unsigned long pot2[17] =
{1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536};

// Variablen zum Abspeichern der Kanal-Auflösung
unsigned char res_A,res_B,res_C,res_D;

// Variablen zum Abspeichern der Kanal-Referenzspg.
float refspg_A,refspg_B,refspg_C,refspg_D;

```

Zunächst einmal wurden hier sechs verschiedene Initialisierungsvektoren ‘..._v0’ bis ‘..._v5’ angelegt, die bei Bedarf für unterschiedliche DS2450er-Einstellungen sorgen. Mit anderen Worten: je nach den gewünschten Einstellungen für den DS2450er wird genau einer dieser Vektoren an die Initialisierungsfunktion übergeben und dort ausgewertet. Der Anwender kann sich selbstverständlich weitere, eigene, individuelle Vektoren für andere Einstellungen „zusammenbauen“ und verwenden.

Als nächstes haben die (festen) 2er-Potenz-Umrechnungswerte für die verschiedenen Auflösungen im Vektor ‘pot2[.]’ hinterlegt: bei der Berechnung der gemessenen Eingangsspannungen gilt ja für den Wert des LSBs bei der A/D-Wandlung:

$$\text{Wert des LSBs} = U_{\text{ref}} / 2^n$$

wobei n die jeweilige Auflösung ist.

Und anstatt diese 2^n -Werte jeweils im Programm immer neu berechnen zu lassen (das kostet Programmspeicher und vor allen Dingen Zeit !), werden diese Konstanten ebenfalls fest im Programmspeicher hinterlegt.

Danach folgen vier Variablen zur Abspeicherung der gewählten Auflösungen ‘res_...’, damit man mit diesen Werte im Programm auch rechnen kann und vier weitere Variablen ‘refspg_...’, in denen die gewählten Referenzspannungen für jeden Kanal festgehalten werden.

Jetzt fehlt nur noch die eigentliche Initialisierungsfunktion, die die Werte aus dem jeweils übergebenen Initialisierungsvektor in den DS2450er überträgt und die notwendigen Berechnungsvariablen auf die entsprechenden Werte setzt.

Unsere Lösung zu dieser letzten DS2450er-API-Funktion sieht wie folgt aus:

```

/**/ Initialisierung des DS2450ers durch Übergabe u. Auswertung des entspr. Init-Vektors ***/
/**/ Der gewünschte Init-Vektor wird an die Fkt. übergeben ***/

void ds2450_init(unsigned char *v)
{
    unsigned char zw;

    /*****/
    /**/ Festlegungen für Page 1 in Sendevektor einschreiben ***/
    /*****/
    /**/ Auflösungen ***/
    // Kanal A - Adr: 08h
    res_A = v[0]; // Abspeichern der Kanal-Auflösung
    if (v[0] == 16) ds2450_p[0] = 0;
        else ds2450_p[0] = v[0];

    // Kanal B - Adr: 0ah
    res_B = v[1];
    if (v[1] == 16) ds2450_p[2] = 0;
        else ds2450_p[2] = v[1];

    // Kanal C - Adr: 0ch
    res_C = v[2];
    if (v[2] == 16) ds2450_p[4] = 0;
        else ds2450_p[4] = v[2];

    // Kanal D - Adr: 0eh
    res_D = v[3];
    if (v[3] == 16) ds2450_p[6] = 0;
        else ds2450_p[6] = v[3];

    /**/ Referenzspannungen ***/
    // Kanal A - Adr: 09h
    ds2450_p[1] = v[4];
    refspg_A = (v[4]*2.56) + 2.56; // Kanal-Ref.spg. merken

    // Kanal B - Adr: 0bh
    refspg_B = (v[5]*2.56) + 2.56;
    ds2450_p[3] = v[5];

    // Kanal C - Adr: 0dh
    refspg_C = (v[6]*2.56) + 2.56;
    ds2450_p[5] = v[6];

    // Kanal D - Adr: 0fh
    refspg_D = (v[7]*2.56) + 2.56;
    ds2450_p[7] = v[7];

    /*****/
    /**/ Einschreiben der Init-Daten (= Inhalt ds2450_p[.]) in Page 1 ***/
    /*****/
    ds2450_write_page(1);
}

```

```

/*****
/ *** Noch notwendigen Wert in Page 3 einschreiben unter Adr: 1ch ***
/ *****/

// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(25);

// Adressierung des jeweils als aktiv ausgewählten Slaves
ow_match();

// Schreib-Befehl senden: "WRITE Memory" = 0x55
ow_wr_byte(0x55);

// Register-Start-Adresse übermitteln: TA1 und TA2
ow_wr_byte(0x1c);          // TA1 = niederwertig = 1c
ow_wr_byte(0x00);          // TA2 = höherwertig = 00

// 1. Datenbyte senden für Reg-Adr: 1ch
ow_wr_byte(0x40);          // Datum = 40h

// 16-Bit-CRC vom DS2450er zurück lesen
crc1=ow_rd_byte();          // CRC-1
crc2=ow_rd_byte();          // CRC-2

// Kopie des gesendeten Datums vom DS2450er zurücklesen und vergleichen
zw=ow_rd_byte();
if (zw != 0x40)             // Fehler
{
    ow_fem(26);             // Fehlermeldung ausgeben
}

// CRC-Überprüfung; entfällt zur Zeit !!

// Fertig mit dem Senden dieses Bytes
// Master sendet abschließenden Reset-Impuls
ow_pres_check(29);
}

/*****/

```

An die Funktion wird der gewünschte Initialisierungsvektor übergeben und somit sieht der Initialisierungs-Aufruf von der main-Funktion dann z.B. so aus (wenn der Initialisierungsvektor 'ds2450_init_v4' verwendet werden soll):

```
ds2450_init(ds2450_init_v4);
```

Das ist schon alles, was der Anwender machen muß, um den DS2450er richtig einzustellen. Den Rest erledigt die Initialisierungsfunktion:

Alle notwendigen Initialisierungswerte werden aus dem Initialisierungsvektor entnommen und in die globale Variable 'ds2450_p[.]' eingeschrieben, die danach mit ihren 8 Byte exakt dem Aufbau der DS2450er-Page 1 entspricht, in der ja alle Initialisierungsdaten für den DS2450er eingeschrieben werden müssen (s. Kap.7.1, Beschreibung zur Page 1).

Gleichzeitig werden die Variablen für die Abspeicherung der gewählten Auflösungen und der gewählten Referenzspannungen für jeden Kanal gesetzt.

Anschließend erfolgt die Übertragung des so „zusammengebauten“ Vektors 'ds2450_p[.]' in die DS2450er-Page 1: dazu verwenden wir die bereits zuvor entwickelte DS2450er-API-Funktion 'ds2450_write_page(...)' in der Form:

```
ds2450_write_page(1);
```

Abschließend muß noch unbedingt die einzelne Speicherstelle mit der Adresse 1ch auf der Page 3 mit dem Wert 40h beschrieben werden, s. Abb.7.10.

Das geschieht mit dem kleinen Programmsegment, das wir schon in Kap.7.2 vorgestellt haben.

Und fertig ist die universell einsetzbare DS2450er-Initialisierungsfunktion !

Bevor wir nun endgültig unser DS2450er-Gesamt-Demo-Programm erstellen, sollen, wie versprochen, noch einige Verbesserungen an der 1-Wire-Grund-API vorgenommen werden, um die Adressierung von verschiedenen 1-Wire-Bausteinen in einem 1-Wire-Bussystem zu vereinfachen.

Ergänzungen in der API 'ow.c/h'

Drei neue Funktionen sind hier hinzu gekommen:

Funktion: 'void ow_fem(unsigned char fe_nr)

```
/** Ausgabe einer Fehlermeldung zur einer Fehlernummer fe_nr */
void ow_fem(unsigned char fe_nr)
{
    printf("\n\n%3u: ", fe_nr);          // Fehlernummer ausgeben

    // Fehlernummer auswerten
    switch(fe_nr)
    {
        case 20:    printf("Kein Presence beim Schreiben in eine Seite (1) !"); break;
        case 21:    printf("Fehler beim Schreiben - Ruecklesen !"); break;
        case 22:    printf("Kein Presence beim Schreiben in eine Seite (2) !"); break;
        case 23:    printf("Kein Presence beim Lesen aus einer Seite (1) !"); break;
        case 24:    printf("Kein Presence beim Lesen aus einer Seite (2) !"); break;
        case 25:    printf("Kein Presence beim Schreiben in Page 3 (Init-Anf.) !"); break;
        case 26:    printf("Fehler beim Rücklesen von Page 3 (Init) !"); break;
        case 27:    printf("Kein Presence beim Start Convert (Anfang) !"); break;
        case 28:    printf("Kein Presence beim Start Convert (Ende) !"); break;
        case 29:    printf("Kein Presence beim Schreiben in Page 3 (Init-Ende) !"); break;

        default:    printf("Ungueltige Fehlernummer aufgetreten !"); break;
    }

    printf("\n\nSystem angehalten !\n\n\n");

    // Endlosschleife
    while(1);
}
```


Diese Funktion dient dazu, beim Auftreten von 1-Wire-Fehlern im Verlauf des Programms eine Fehlernummer und einen Fehlermeldungstext auszugeben.

An die Funktion wird eine vom Anwender beliebige wählbare Fehlernummer (Wertebereich 0 - 255) übergeben, die beim Auftreten des Fehlers die Fehlerstelle bzw. die Fehlerursache charakterisiert.

Dieser Fehlernummer muß dann in der obigen Funktion ein passender Fehlermeldungstext zugeordnet sein, der dann zusammen mit der Fehlernummer ausgegeben wird (Auswertung der Fehlernummer in der switch ... case-Abfrage).

Danach bleibt das Programm in einer Endlosschleife stehen, d.h. der Mikrocontroller muß, nachdem der Fehler beseitigt worden ist, komplett zurück gesetzt werden, und das Programm startet neu.

Beispiel:

Der 1-Wire-Master sendet zu Anfang eines Kommunikationszyklusses (Schreiben von Daten in die Page 1 eines DS2450ers) einen Reset-Impuls und kein einziger DS2450er bzw. kein einziger 1-Wire-Slave meldet sich mit einem Presence-Impuls zurück.

In diesem Fall könnte der Aufruf der Fehlermeldung wie folgt aussehen:

```
ow_fem(20);
```

Und auf dem Terminalbildschirm erscheint die Meldung:

```
20: Kein Presence beim Schreiben in eine Seite (1) !
```

So lassen sich im Programm an den „kritischen Stellen“ Fehlermeldungen einbauen, die beim ersten Testen und auch beim späteren Betrieb des Systems wertvolle Hilfestellungen leisten können.

Funktion: `void ow_pres_check(unsigned char fe_nr)`

```
/** Master-Reset senden und Slave-Presence abfragen - mit Fehlernummer, wird übergeben */  
void ow_pres_check(unsigned char fe_nr)  
{  
    // Start mit Master-Reset-Impuls u. Abfrage: Slave-Presence  
    if(ow_reset())           // Fehler aufgetreten  
    {  
        ow_fem(fe_nr);       // Fehlermeldung ausgeben  
    }  
}
```

Die gerade beschriebene Möglichkeit zur Ausgabe einer Meldung im Fehlerfalle haben wir hier direkt angewendet: wenn der Master jetzt zu Beginn eines Kommunikationszyklusses einen Reset-Impuls ausgibt und kein einziger 1-Wire-Slave meldet sich zurück, so wird eine entsprechende Fehlernummer und ein Fehlertext ausgegeben, die zusammen die Stelle des Fehlers kennzeichnen.

Mit anderen Worten: ab jetzt **beginnt jeder unserer 1-Wire-Kommunikationszyklen mit:**

```
ow_pres_check(x);          // x = zugehörige Fehlernummer
```

Da man als Anwender ja die Fehlernummer und den Fehlertext beliebig festlegen kann, lässt sich jetzt sehr genau erkennen, wo und wann das Problem aufgetreten ist.

Abschließend betrachten wir uns noch **die wesentlichste Änderung** bei Ablauf der Kommunikation mit verschiedenen 1-Wire-Bausteinen in einem System:

Dazu haben wir das Chip-Adressierungs-Konzept etwas überarbeitet:

- 1) Im **Einzelbetrieb** (eines Bausteins am Bus) erfassen wir den ROM-Identifizier von jedem 1-Wire-Chip, den wir später im System einsetzen wollen.
- 2) Diese ganzen benutzbaren Identifier legen wir in einem Feld (Matrix) aus unsigned char-Daten ab, das die Größe hat:

- Anzahl der Spalten = 8 \equiv da jeder ROM-Identifizier aus 8 Bytes besteht
- Anzahl der Zeilen \equiv mindestens Anzahl der 1-Wire-Chips im System, da jeder ROM-Identifizier komplett in einer Zeile steht.

Dieses Array nennen wir 'ow_sl_mat[.][.]' (\equiv 1-Wire-Slave-Matrix) und es wird als (globale) Variable wie folgt definiert:

```
code unsigned char ow_sl_mat[20][8] = ...
```

Durch den Zusatz 'code' wird dieses Feld fest im Programmspeicher abgelegt, man spart also wertvollen Datenspeicherplatz (da ja die ROM-Identifizier während des Programmablaufs immer nur ausgelesen und nie neu beschrieben werden, kann man sie sehr gut im nur lesbaren Programmspeicher hinterlegen).

Durch die zuvor gemachten Array-Definition haben wir also ein Feld für die ROM-Identifizier für bis zu 20 1-Wire-Slaves angelegt.

- 3) Wir besetzen nun jede Zeile des Arrays mit dem kompletten ROM-Identifizier eines 1-Wire-Slaves (\equiv Anfangsinitialisierung des Arrays).

Damit erhält man dann z.B.:

```
// Matrix für bis zu 20 angeschlossene 1-Wire-Slaves:
// - Jeder Slave hat einen 8 Byte großen ROM-Code (ROM Identifizier).
// - Jede Zeile in der Matrix entspricht dem ROM-Code eines Slaves.
// - Eintragungen in einer Zeile fangen beim CRC-Code an, letztes Byte: Family-
//   Code.
// - Die Slave-Zeilen können in beliebiger Reihenfolge stehen.
// - Die jeweilige Zeilennummer entspricht der Nummer, unter der der Slave
//   im Programm angesprochen wird = Wert der Variablen 'ow_sl_aktiv'

code unsigned char ow_sl_mat[20][8] = {
{0,0,0,0,0,0,0,0},          // Slave 0:
{0xfe,0x00,0x08,0x01,0x9c,0x0e,0x6d,0x10}, // Slave 1: DS18S20er - 1
```

```

{0x0c,0x00,0x08,0x01,0xdb,0x08,0x21,0x10}, // Slave 2: DS18S20er - 2
{0xc6,0x00,0x08,0x01,0x9c,0x3f,0x0b,0x10}, // Slave 3: DS18S20er - 3
{0x54,0x00,0x08,0x01,0x86,0x70,0x0a,0x10}, // Slave 4: DS18S20er - 4
{0,0,0,0,0,0,0,0}, // Slave 5:
{0,0,0,0,0,0,0,0}, // Slave 6:
{0,0,0,0,0,0,0,0}, // Slave 7:
{0,0,0,0,0,0,0,0}, // Slave 8:
{0,0,0,0,0,0,0,0}, // Slave 9:
{0xd9,0x00,0x00,0x00,0x13,0xfa,0xd3,0x20} // Slave 10: DS2450er - 0
};

```

Wenn eine Zeile nicht mit dem ROM-Identifizier eines Slave belegt wird, so trägt man einfach acht Nullen ein.

Es müssen auch nicht alle 20 Zeilen der Matrix auf einmal vorbesetzt werden.

In unserem Fall haben wir von den 20 möglichen Zeilen insgesamt nur 11 vordefiniert, und zwar mit den Erkennungsnummern von (nur) fünf 1-Wire-Slaves: vier Temperatur-Sensoren DS18S20er und einen 4-fach-A/D-Wandler DS2450er.

Und nun kommt der „wesentliche“ Trick bei der Adressierung der 1-Wire-Slave-Stationen:

Zu jedem 1-Wire-Chip gehört ja seine eigene Zeile mit seinem individuellen ROM-Identifizier, mit anderen Worten: über die Nummer seiner Zeile in der Matrix `ow_sl_mat[.][.]` ist jeder eingetragene Baustein Adress-mäßig eindeutig identifizierbar.

Beispiel:

Über die (Zeilen)Nummer 1 kann der ROM-Identifizier des Slaves 1 (\equiv DS18S20er - 1), über die (Zeilen)Nummer 10 kann der ROM-Identifizier des Slaves 10 (\equiv DS2450er - 0), usw. angesprochen werden.

Über die Angabe der Zeilen-Nummer in der Matrix `ow_sl_mat[.][.]` erhält man also sofort den zur Adressierung des Chips benötigten ROM-Identifizier !

Globale Variablen, III

Im letzten Schritt führen wir nun noch eine globale Variable namens `ow_sl_aktiv` ein, in der wir die Zeilen-Nummer des gerade aktiven 1-Wire-Slaves ablegen, oder anderes formuliert: wenn wir in jetzt unserem (Haupt)Programm mit dem Slave Nummer 3 (\equiv DS18S20er - 3) arbeiten wollen, so erhält die Variable `ow_sl_aktiv` den Wert 3 zugeordnet und gibt damit die Zeilen-Nummer in der Matrix an, aus der dann der zugehörige ROM-Identifizier für diesen Chip entnommen werden kann.

Soll dann im weiteren Verlauf mit einem anderen 1-Wire-Baustein gearbeitet werden, z.B. mit dem DS2450 - 0 (\equiv Slave-Nr. 10), so erhält die Variable `ow_sl_aktiv` den Wert 10 zugeordnet, usw.

Über diesen so festgelegten Wert von `ow_sl_aktiv` wird dann in der letzten noch zu entwickelnden Funktion namens `ow_match()` die ganz konkrete Adressierung des jeweiligen Bausteins durchgeführt:

```

/*** Adressierung eines 1-Wire-Slaves über Skip-ROM oder Match-ROM ***

```

```

void ow_match(void)
{
    unsigned char i;

    // Prüfen der Slave-Adresse
    if(ow_sl_aktiv == 255)                // Nur ein Slave am Bus angeschlossen,
                                          // also mit Skip-ROM arbeiten
    {
        // Skip ROM-Befehl, da nur ein Slave angeschlossen ist
        ow_wr_byte(0xcc);
    }
    else                                  // Mehrere Slaves am Bus
    {
        // Match ROM-Befehl und Slave-ROM-Code aussenden.
        // ROM-ID gem. Eintrag in Slave-Matrix 'ow_sl_mat[..]' in
        // der Header-Datei 'ow_slaves.h'
        // Die globale Var. 'ow_sl_aktiv' enthält die Nr. des 1-Wire-Slaves,
        // mit dem gearbeitet werden soll !

        ow_wr_byte(0x55);                // Match-ROM-Anweisung

        // Entspr. ROM-ID aussenden
        for (i=0; i<8; i++) ow_wr_byte(ow_sl_mat[ow_sl_aktiv][7-i]);
    }

    // Ab hier kommt dann der jeweilige Befehl für den 1-Wire-Slave:
    // hier kommt dann die Fortsetzung im aufrufenden Programm-Teil !
}

```

Nach dem Aufruf dieser Funktion wird im Funktionskörper der Wert der globalen Variablen 'ow_sl_aktiv' ausgewertet, wobei jetzt zwei große Fälle unterschieden werden:

- 1) Hat die Variable den **Wert 255**, so bedeutet das, dass **nur ein einziger 1-Wire-Slave am Bus angeschlossen** ist und dass wir in diesem Fall auf eine individuelle Adressierung über den ROM-Identifizierer verzichten und statt dessen mit dem vereinfachten ROM-Kommando 'Skip ROM' arbeiten können.
In diesem Falle ist die if-Abfrage erfüllt und die Skip-ROM-Anweisung wird ausgeführt.
So geht man z.B. vor, wenn ein neuer Slave-Baustein erstmalig (alleine) am Bus betrieben wird und man den ROM-Identifizierer erst einmal auslesen möchte.
- 2) Sind nun **mehrere 1-Wire-Chips am Bus angeschlossen** und möchte man jetzt ganz gezielt für die nachfolgenden Aktionen einen bestimmten Slave adressieren, so setzt man den Wert der Variablen 'ow_sl_aktiv' auf die Zeilen-Nummer des Bausteins in der Matrix.
Jetzt wird der else-Zweig ausgeführt, d.h. aus der adressierten Zeile in der Matrix 'ow_sl_aktiv[.][.]' wird der passende ROM-Identifizierer ausgelesen und mittels 'Match ROM'-Kommando an den Slave ausgesendet:

```
for (i=0; i<8; i++) ow_wr_byte(ow_sl_mat[ow_sl_aktiv][7-i]);
```

So lässt sich über den Wert von 'ow_sl_aktiv' sehr einfach die gewünschte Zeile in der Matrix und damit der gerade zu adressierende 1-Wire-Chip auswählen.

Zusammen gefasst muss der Anwender **beim Einsatz eines neuen 1-Wire-Bausteines im System** einfach nur folgendes machen:

- Im „Einzel-Betrieb“ wird der ROM-Identifizier des neuen Chips ermittelt.
- Die achte Bytes des ROM-Identifiziers werden in eine neue, freie Zeile der Matrix 'ow_sl_mat[.][.]' eingetragen und man „merkt“ sich einfach die zugehörige Zeilen-Nummer.
- Soll dann später im Hauptprogramm mit diesem Slave gearbeitet werden, so erhält die Variable 'ow_sl_aktiv' diese Zeilen-Nummer zugeordnet.

Die nachfolgenden **Beispiele** sollen das Arbeiten mit diesem Konzept verdeutlichen:

Beispiel 1:

Ein beliebiger Kommunikations-Zyklus mit einem 1-Wire-Slave beginnt ab jetzt also immer wie folgt:

```
// Im main-Teil des Programms: Festlegung des Wertes für 'ow_sl_aktiv'
// für dne gewünschten Slave
ow_sl_aktiv = .....

.....
.....
// Nun: Kommunikation mit dem Slave:
// Start mit Master-Reset-Impuls u. Abfrage: Slave presence
ow_pres_check(.....);      // Mit Übergabe der passenden Fehler-Meldungs-Nr.

// Adressierung des jeweils als aktiv ausgewählten Slaves
ow_match();

// Nun wird der gewünschte Befehl gesendet und der Slave-spez.
// Kommunikationsteil abgewickelt:
ow_wr_byte(.....);        // Befehlsbyte senden
```

Beispiel 2:

Nur ein einziger Slave ist am Bus angeschlossen, Arbeiten mit 'Skip ROM':

```
// Spezielle Skip ROM-Adresse
ow_sl_aktiv = 255;

// Adressierung dieses Slaves:
ow_match();

// Nun können weitere ROM-Befehle bzw. Slave-spezifische Befehle
```

```
// folgen, da der Slave bereits korrekt (allgemein) adressiert worden ist  
ow_wr_byte(.....);          // Befehlsbyte senden
```

Beispiel 3:

Betrieb eines kompletten 1-Wire-Systems mit einer Vielzahl von Slave-Bausteinen, deren ROM-Identifizierer in der Matrix `ow_sl_aktiv[.][.]` abgelegt sind.

Es soll nun mit einem bestimmten Slave gearbeitet werden:

```
// Spezielle Adresse dieses Slaves  
ow_sl_aktiv = .....;  
  
// Individuelle Adressierung dieses Slaves:  
ow_match();  
  
// Nun können weitere ROM-Befehle bzw. Slave-spezifische Befehle  
// folgen, da der Slave bereits korrekt adressiert worden ist  
ow_wr_byte(.....);          // Befehlsbyte senden
```

Zum Abschluss erstellen wir nun noch eine neue `‘C’`-Header-Datei namens `‘ow_slaves.h’`, in der wir nur die globale Variable `‘ow_sl_aktiv’` und das globale Array `‘ow_sl_mat[.][.]’` ablegen.

Mit anderen Worten: nur in dieser Datei muss der Anwender jetzt etwas ändern, nämlich die jeweiligen ROM-Identifizierer der Slaves eintragen.

Danach kann er diese Datei in sein Projekt und in sein Programm einbinden.

Mehr ist nicht zu tun, um mit einem 1-Wire-Bussystem komfortabel zu arbeiten.

Eine entsprechende Realisierung dieses Konzeptes finden Sie in unseren DS2450er-Demo-Programmen in Kap.7.6

Hinweis:

Die bisher entwickelten Routinen für den digitalen Temperatursensor DS18S20 (s. Kap.6.3 und Kap.6.4) arbeiten zur Zeit noch nicht mit diesem neuen Adressierungsverfahren. Sie müssen beim Arbeiten mit den DS18S20ern noch die dort beschriebene „alte Methode“ verwenden oder Sie passen sich die DS18S20er-API-Routinen selber entsprechend auf die neuen Abläufe an.

Beim nächsten Up-Date des Projektes werden wir aber diese Änderungen mit durchführen, so dass die gesamte 1-Wire-Software „aus einem Guss“ besteht.

Damit haben wir nun (endlich) alles zusammen, um den DS2450er in Betrieb zu nehmen. Aber vorher müssen wir ihn natürlich noch Hardware-mäßig an unseren Mikrocontroller an-koppeln.

7.5 Hardware

Beim Hardware-mäßigen Anschluss des DS2450ers an einen Mikrocontroller gibt es nun wieder zwei Möglichkeiten:

Verwendung des PT-1-Wire-Experimental-Boards, Abb.7.5.1

Auf dieser Platine haben wir bereits einen DS2450er mit integriert, bei dem ein Eingang fest mit einem Potentiometer verbunden ist, so dass damit sehr einfach analoge Eingangsspannungen im Bereich von 0 ... +5 V simuliert und erfasst werden können.

Abb.7.5.1: Der DS2450er auf dem PT-1-Wire-ExBo

Die anderen drei Analog-Eingänge des DS2450ers sind einfach auf entsprechende Schraub-Klemmanschlüsse geführt worden, so dass man dort z.B. externe (Sensorausgangs)Spannungen auflegen kann.

Die Versorgung des DS2450er erfolgt hier durch lokale Speisung.

Zwei Punkte sind allerdings zu beachten:

- 1) Es sind keinerlei Schutzbeschaltungen gegenüber zu hohen Eingangsspannungen vorhanden.
Hier muss der Anwender also selber darauf achten, dass keine „zerstörerischen“ Pegel auftreten oder er muss selber noch einen entsprechenden Schutz dafür vorsehen.
- 2) Desgleichen sind keinerlei Filterbaugruppen vorgesehen, die unerwünschte Störungen auf dem Nutzsignal herausfiltern.
Auch hier muss der Anwender bei Bedarf selber handeln.

Entwicklung einer eigenen Schaltung mit dem DS2450er, Abb.7.5.2

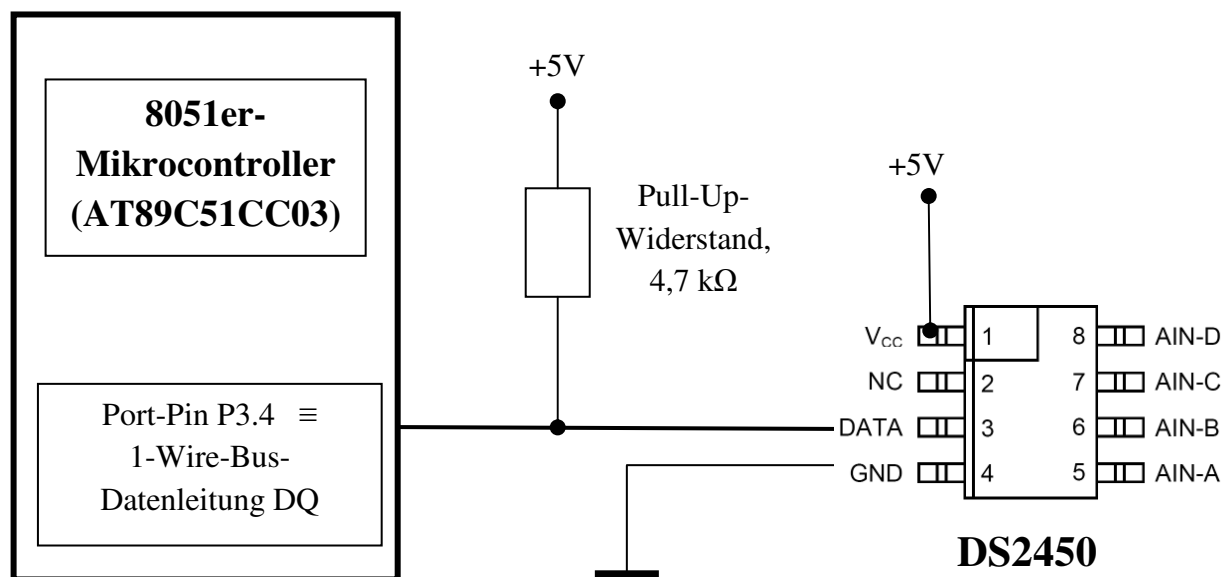


Abb.7.5.2: Der einfache Anschluss des DS2450ers

Da der DS2450er aus der Gruppe der 1-Wire-Slaves stammt, ist sein Anschluss an den Mikrocontroller, wie schon beim DS18S20er gesehen (s. Kap.6.2), natürlich denkbar einfach:.

Die charakteristischen Kennzeichen dieser Minimalbeschaltung sind auch hier:

- Lokale Speisung.
- Keine Schutzbeschaltung gegenüber zu hohen Eingangsspannungen.
- Keine Filterbeschaltung zur Unterdrückung von Störungen auf dem Nutzsignal.

Aber der Entwicklungsvielfalt des Anwenders sind natürlich auch hier keine Grenzen gesetzt.

7.6 Das DS2450er-Demo-Programm

Nun haben wir alles zusammen, um ein kleines DS2450er-Demo-Programm zu entwickeln, das Sie in zwei Versionen in unserer Programmsammlung finden: 'ds2450_demo_1.c' bzw. 'ds2450_demo_2.c':

- Das erste Demo-Programm ist so geschrieben, dass sich alle Funktionen im Programm selber befinden und die DS2450er-API nicht mit eingebunden wurde. So kann der gesamte Programm-Ablauf etwas besser verfolgt werden und man muss nicht immer zwischen dem Hauptprogramm und der API-Funktionssammlung hin und herschalten, wenn man etwas nachvollziehen möchte.
- Das zweite Demo-Programm enthält die gleiche Funktionalität wie das Erste, nur wird hier die DS2450er-API (mit den bekannten Funktionen) mit eingebunden. Das eigentliche Hauptprogramm wird so kompakter und letztendlich wird man in der späteren Praxis seine Programme immer so, unter Verwendung der erstellten APIs, entwickeln.

Nach dem Start von einem der Programme erscheint zunächst das allgemeine Auswahlmenü, **Abb.7.6.1:**

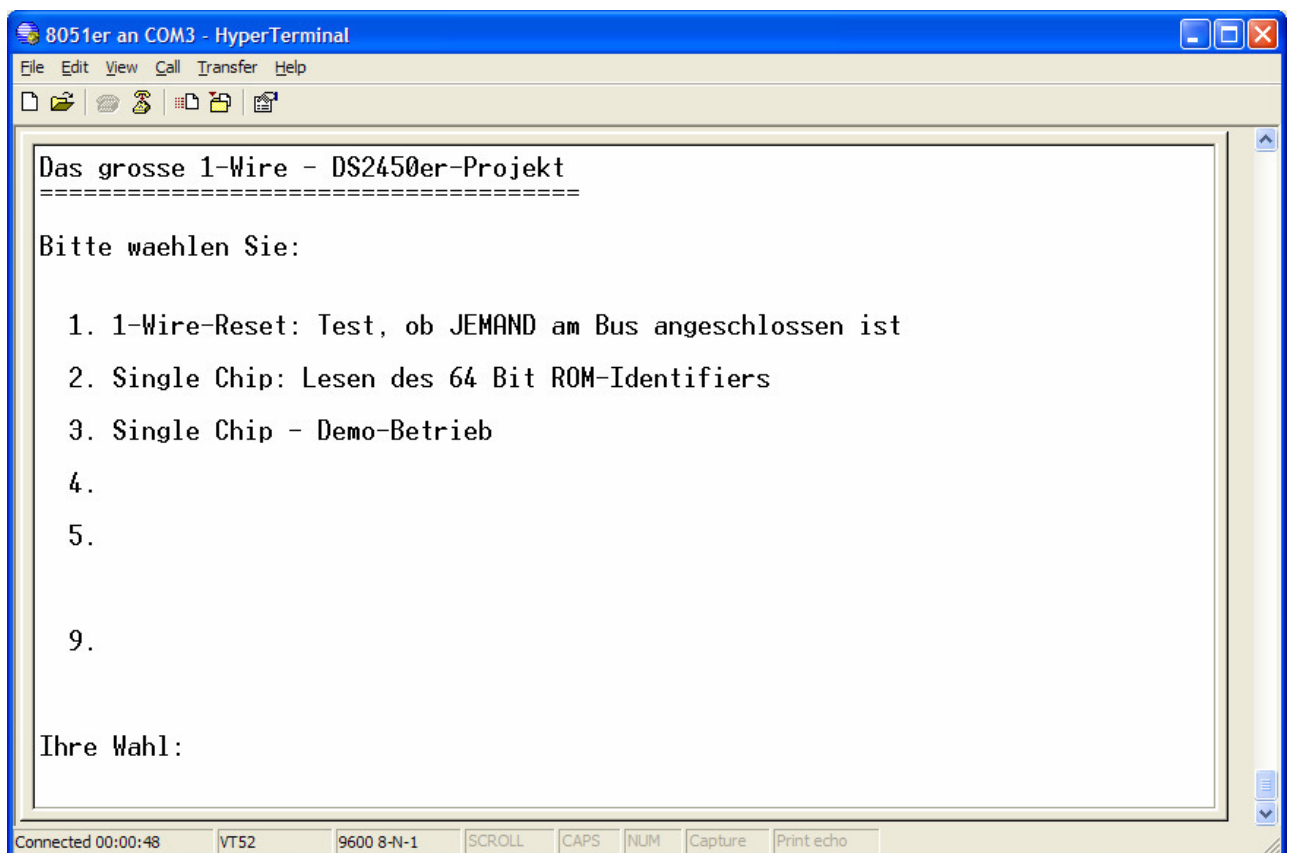


Abb.7.6.1: Das Auswahlmenü

Drei Auswahlpunkte stehen dem Anwender hier zunächst zur Verfügung: die ersten beiden kennen Sie schon vom DS18S20er her und diese dienen zur Feststellung, ob mindestens ein 1-Wire-Slave am Bus angeschlossen ist und welchen ROM-Identifizierer dieser besitzt, **Abb.7.6.2** und **Abb.7.6.3**:

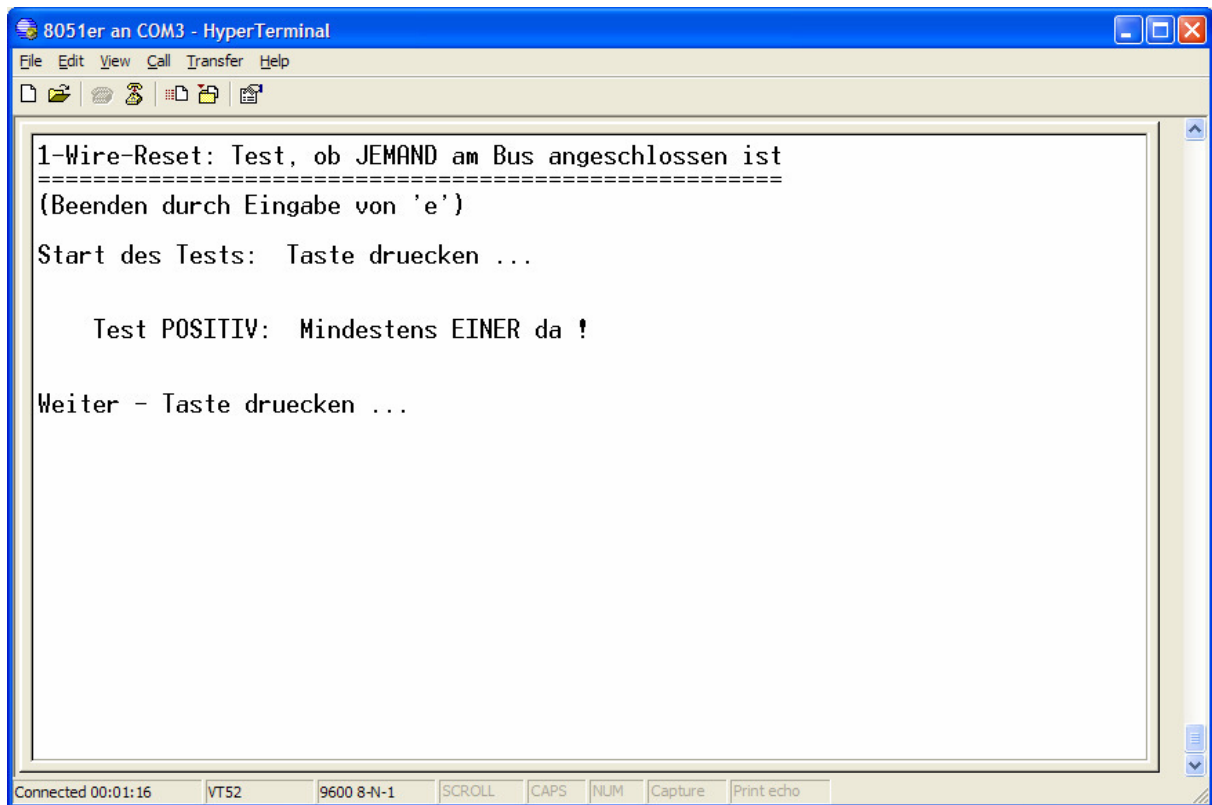
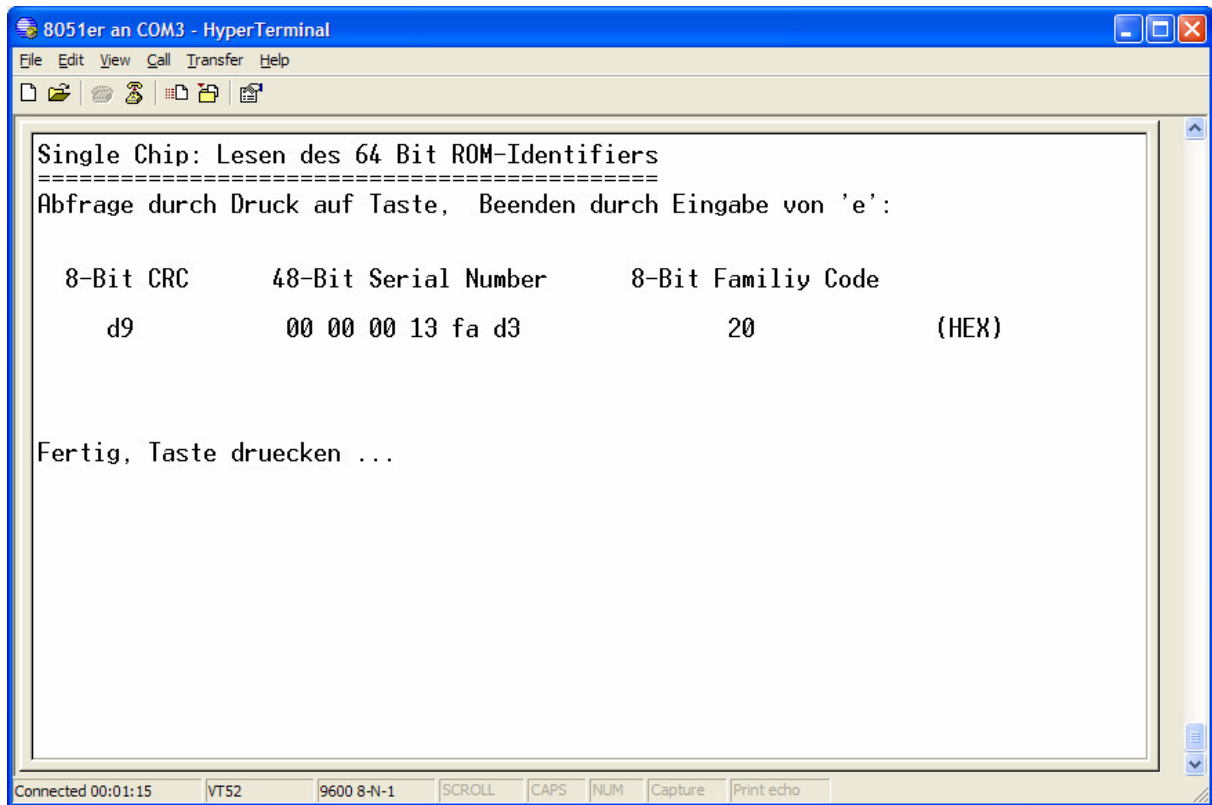
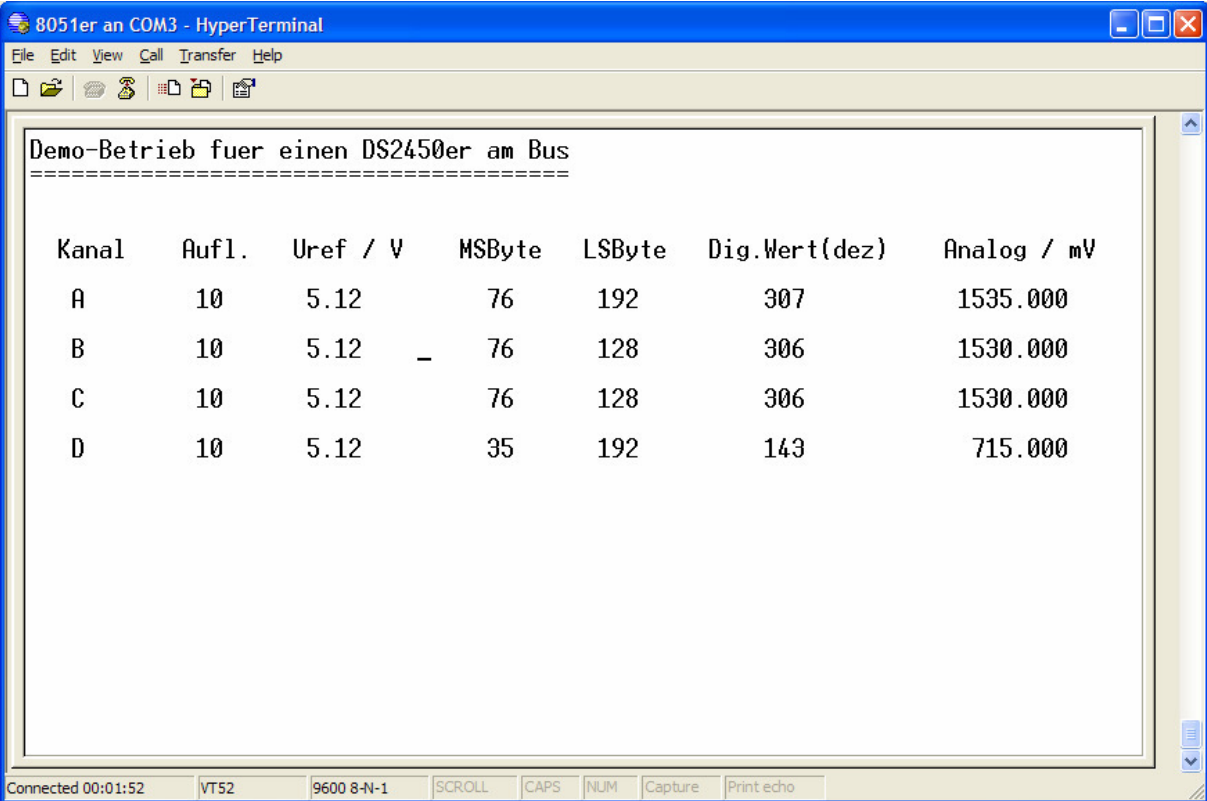


Abb.7.6.2: Die Suche nach einem 1-Wire-Slave



7.6.3: Die Ermittlung des zugehörigen ROM-Identifiers

Der Kern der Demo-Programme liegt aber natürlich beim konkreten Betrieb des DS2450ers und dazu dient der letzte Menü-Auswahlpunkt, **Abb.7.6.4:**



Demo-Betrieb fuer einen DS2450er am Bus

Kanal	Aufl.	Uref / V	MSByte	LSByte	Dig. Wert(dez)	Analog / mV
A	10	5.12	76	192	307	1535.000
B	10	5.12	76	128	306	1530.000
C	10	5.12	76	128	306	1530.000
D	10	5.12	35	192	143	715.000

Abb.7.6.4: Der Betrieb des DS2450ers

In diesem Programmteil werden alle relevanten Daten des aktuellen Betriebs des DS2450ers dargestellt:

- Zuerst für jeden Kanal die eingestellten Initialisierungsdaten: gewählte Auflösung und gewählte Referenzspannung (gemäß den Festlegungen aus dem verwendeten Initialisierungsvektor).
- Danach die aktuell erfassten Messwerte: einmal in der Originalform als zwei einzelne Byte-Werte (MSByte und LSByte) und einmal als zusammengefasster Gesamtdigitalwert im dezimalen Zahlensystem.
- Und letztendlich wird der zurück gerechnete, am Kanal anliegende, analoge Spannungswert dargestellt.

Zwischen den einzelnen Messungen wird eine Wartezeit von ca. 400 ms eingefügt.

Beachten:

Die aktuelle Demo ist für einen (einzigen) DS2450er am Bus geschrieben, da auf dem PT-1-Wire-ExBo auch nur ein solcher Chip vorhanden ist.

Einer Erweiterung auf mehrere Bausteine bzw. der Durchführung komplett eigener Entwicklungen und Applikationen mit dem DS2450er steht nun nichts mehr im Wege und wir beschäftigen uns abschließend mit **dem dritten 1-Wire-Chip im Bunde**, mit der digitalen Port-Erweiterung, mit dem **DS2408**.

10. Die Projekt-Software

In den nächsten Kapiteln werden die einzelnen APIs bzw. deren globale Variablen und Funktionen kurz beschrieben und die Demo-Programme für die einzelnen 1-Wire-Bausteine aufgelistet.

So erhält man eine kompakte Übersicht über die vorhandenen Software-Module für die tägliche Entwicklungsarbeit mit dem 1-Wire-Bus.

10.2.5 ds2450.c/.h

Globale Variabeln im *.h-Teil dieser API

- Beispielhafte Initialisierungsvektoren für den DS2450er.
- Ausgerechnete 2er-Potenz-Werte.
- Variablen zur Abspeicherung der eingestellten Kanal-Auflösungen und Kanal-Referenzspannungen.
- 8 Byte großes Array 'ds2450_p[.]' zum Page-orientierten Datentransfer.
- Variablen zur Abspeicherung der CRC-Checksummen-Werte.

Funktionen im *.c-Teil dieser API

void ds2450_init(unsigned char *v)

Durch Aufruf dieser Funktion wird der DS2450er entsprechend initialisiert, wobei der jeweilig gewünschte Initialisierungsvektor an die Funktion übergeben wird.

void ds2450_write_page(unsigned char p_nr)

Einschreiben von 8 Byte in eine Page des DS2450er.

Übergeben wird die Page-Nr 'p_nr', hier sinnvoll nur '1' oder '2'.

Die Werte, die in die Page geschrieben werden, müssen zuvor in die globale Variable 'ds2450_p[.]' eingetragen worden sein.

void ds2450_read_page(unsigned char p_nr)

Auslesen von 8 Byte aus einer Page des DS2450er.

Übergeben wird die Page-Nr 'p_nr', hier sinnvoll nur '0', '1' oder '2'.

Die Werte, die aus der Page ausgelesen wurden befinden sich in der globalen Variablen 'ds2450_p[.]' und können von dort aus weiter verarbeitet werden.

void ds2450_convert_all(void)

Durch Aufruf dieser Funktion werden alle vier Kanäle gewandelt und die Wandlungsergebnisse in der globalen Variablen 'ds2450_p[.]' zurück gegeben.

Von dort aus können die Wandlungswerte dann weiter verarbeitet werden.

10.3.3 DS2450

Als Demo-Software für den DS2450er stehen zwei Programme zur Verfügung, die die gleichen Aufgaben erfüllen:

- **ds2450_demo_1.c/.hex:** in diesem Programm sind alle DS2450er-Funktionen direkt enthalten und es wird auf den Einsatz der DS2450er-API verzichtet.
So kann man besser nachvollziehen, wie die Funktionen aufgebaut sind und wie sie eingesetzt werden.
- **ds2450_demo_2.c/.h:** Hier wird die DS2450er-API mit eingesetzt und das Programm wird so übersichtlicher und kompakter.

11. Bezugsquellen

Die Beschaffung von 1-Wire-Chips in kleinen „haushaltsüblichen“ Mengen für den engagierten Praktiker gestaltet sich manchmal etwas schwierig, da die großen Distributoren erst bei Stückzahlen von weit über 100 bereit sind, zu liefern.

Ausgiebige Recherchen im Internet führen jedoch zu Händlern, die auch Einzelstücke, wenn auch mit Preisaufschlag, ausliefern.

Nachfolgend haben wir, ohne Gewähr auf Vollständigkeit, einige Lieferanten zusammengestellt, bei denen wir auch unsere Chips bestellt haben:

Moritz Fuchs Elektronik

www.fuchs-shop.com/de/

DS18S20, DS2408 u.a. 1-Wire-Chips

eservice-online

www.eservice-online.de

DS2450, u.a. 1-Wire-Chips

12. 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 μ C/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

13. 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.

Version 1.2

Veröffentlicht am 07.06.2010.

Teile 1, 2 und 3 des Projektes.

Neu hinzugekommen:

- Kapitel 7, 7.1, 7.2, 7.4, 7.5, 7.6.
- Kapitel 10, 10.2.5, 10.3.3.
- Kapitel 11.