

Grundlagen systemnahes Programmieren

SS 2019

Kap. 6

© HTM, KRF, SWR

bernd.schwarz@haw-hamburg.de

```

void TIM8_UP_TIM13_IRQHandler() {
    static uint8_t i=0, j=0;
    GPIOI->ODR |= (1<<4);
    /// GPIOH->ODR |= (1<<11);
    TIM8 -> SR = ~(1<<0); // Update IRQ löschen
    GPIOI->ODR ^= (1<<7);
    //DAC -> DHR12R1 = Guitar[i] << 6;
    DAC -> DHR12R1 = (sinus_4000[i] + 32768)>>4;
    i++;
    if (i == (num2)) i=0;
    amplitude = ((int32_t)sinus_440[j] + 32768)/256 -1;
    TIM8 ->CCR3 = (uint16_t)amplitude;
}

```



```

if (j >= (num1)) j=0;
GPIOI->ODR ^= (1<<4);
/// GPIOH->ODR ^= (1<<11);
// GPIOI->ODR ^= (1<<6);
//

```



```

// Takt fuer Port I u. H einschalten:
//RCC->AHB1ENR |= RCC_AHB1Periph_GPIOH;
// in CE_lib.c
//RCC->AHB1ENR |= RCC_AHB1Periph_GPIOI;
//RCC->APB2ENR |= RCC_APB2ENR_TIM8EN;
// Takt für Timer 8 einschalten
RCC->APB1ENR |= RCC_APB1ENR_TIM8EN;
// Takt für DAC einschalten
TIM8 -> CR1 = 0; // Timer disable
TIM8 -> CR2 = 0; // Timer disable
TIM8->PSC = 15; // Prescaler,
Timertakt: 11,2 MHz / 16 = 700 ns;
TIM8->ARR = 255-1;
// Auto reload register, T_pwm = 22,727usec;
44kHz; 16800-1
TIM8->CCR3 = 256/2; // Erster
Schaltzeitpunkt // 16800/2
TIM8->CCMR2 = TIM_CCMR2_OC3M_2 |
TIM_CCMR2_OC3M_1; // PWM Mode 1
TIM8->CCER = TIM_CCER_CC3NE;
// Complementary out put enable
TIM8->BDTR = TIM_BDTR_MOE;
// Main output enable
num = sizeof(Guitar)/sizeof(Guitar[0]);

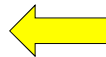
```

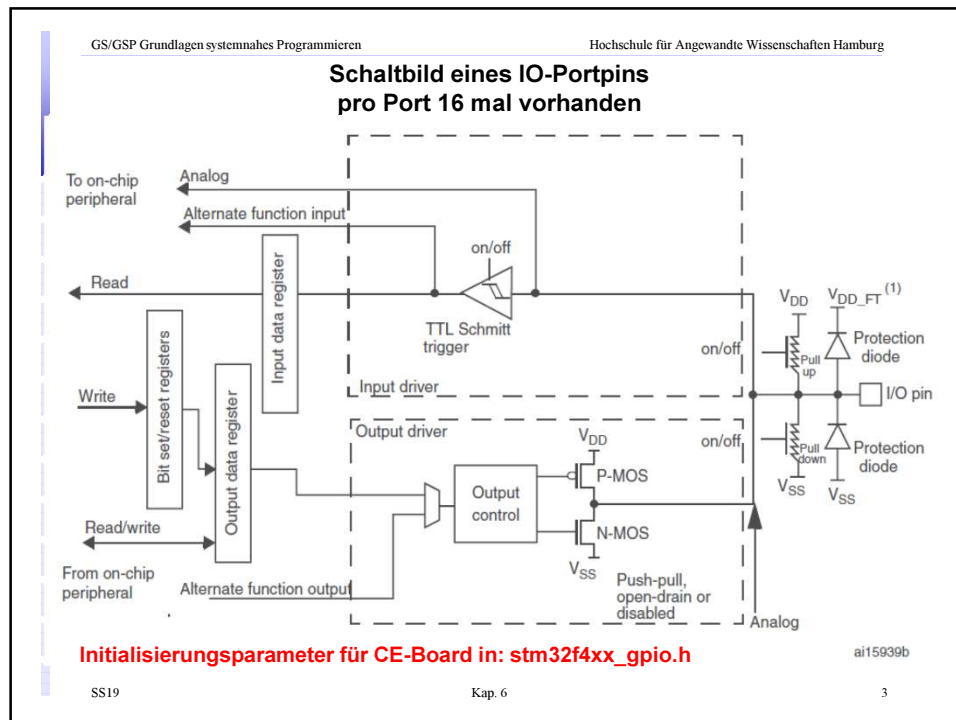


Kapitel 6: I/O Programmierung Teil 2

Gliederung (vgl. Kap. 4)

- Einführung
- Direct Digital Control
- General Purpose Input/Output am Beispiel de STM32F417ZG
- Anschluss an Sensorik-Schnittstellen
- Serielle Datenübertragung
- Prellen mechanischer Schalter





GS/GSP Grundlagen systemnahes Programmieren Hochschule für Angewandte Wissenschaften Hamburg

Schaltbild eines IO-Portpins pro Port 16 mal vorhanden

Output configuration

When the I/O port is programmed as output:

- The output buffer is enabled:
 - Open drain mode:** A "0" in the Output register activates the N-MOS whereas a "1" in the Output register leaves the port in Hi-Z (the P-MOS is never activated)
 - Push-pull mode:** A "0" in the Output register activates the N-MOS whereas a "1" in the Output register activates the P-MOS
- The Schmitt trigger input is activated
- The weak pull-up and pull-down resistors are activated or not depending on the value in the GPIOx_PUPDR register
- The data present on the I/O pin are sampled into the input data register every AHB1 clock cycle
- A read access to the input data register gets the I/O state
- A read access to the output data register gets the last written value

Referencemanual RM0090 Chapter 8.

SS19 Kap. 6 4

GS/GSP Grundlagen systemnahes Programmieren Hochschule für Angewandte Wissenschaften Hamburg

IO-Portpins

GPIO-Ausgang

Betriebsarten:

- **Push-pull.**
- **Open-drain.**
- **Disabled.**

Ausgangswiderstand

- **Pull-up**
- **Pull-down**
- **None**

Ausgangsgeschwindigkeit
2 MHz, 25 MHz, 50 MHz und 100 MHz

```
// PG7 als Ausgang:
GPIOG->MODER = (GPIOG->MODER & ~(3 << (7 * 2))) | (GPIO_Mode_OUT << (7 * 2));

// Speed: 50 MHz
GPIOG->OSPEEDR = (GPIOG->OSPEEDR & ~(3 << (7 * 2))) | (GPIO_Speed_50MHz << (7 * 2));

// Driver: push-pull
GPIOG->OTYPER = (GPIOG->OTYPER & ~(1 << (7))) | (GPIO_OType_PP << (7));

// No pull-up or pull-down
GPIOG->PUPDR = (GPIOG->PUPDR & ~(3 << (7 * 2))) | (GPIO_PuPd_NOPULL << (7 * 2));
```

SS19 Kap. 6 5

GS/GSP Grundlagen systemnahes Programmieren Hochschule für Angewandte Wissenschaften Hamburg

GPIO-Parameter

stm32f4xx_gpio.h

```
typedef enum
{
    GPIO_Low_Speed      = 0x00, /*!< Low speed */
    GPIO_Medium_Speed   = 0x01, /*!< Medium speed */
    GPIO_Fast_Speed     = 0x02, /*!< Fast speed */
    GPIO_High_Speed     = 0x03, /*!< High speed */
}GPIOSpeed_TypeDef;
```

SS19 Kap. 6 6

GPIO-Parameter

stm32f4xx_gpio.h

```
typedef enum
{
    GPIO_Mode_IN    = 0x00, /*!< GPIO Input Mode */
    GPIO_Mode_OUT   = 0x01, /*!< GPIO Output Mode */
    GPIO_Mode_AF    = 0x02, /*!< GPIO Alternate function Mode */
    GPIO_Mode_AN    = 0x03 /*!< GPIO Analog Mode */
}GPIO_Mode_TypeDef;
```

GPIO-Parameter

stm32f4xx_gpio.h

```
typedef enum
{
    GPIO_OType_PP = 0x00,
    GPIO_OType_OD = 0x01
}GPIO_OType_TypeDef;
```

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

GPIO-Parameter

```
stm32f4xx_gpio.h

typedef enum
{
    GPIO_PuPd_NOPULL = 0x00,
    GPIO_PuPd_UP      = 0x01,
    GPIO_PuPd_DOWN    = 0x02
}GPIOPuPd_TypeDef;
```

SS19

Kap. 6

9

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

IO-Portpins

GPIO-Ausgang: OTYPER

Betriebsarten:

- Push-pull.
- Open-drain.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

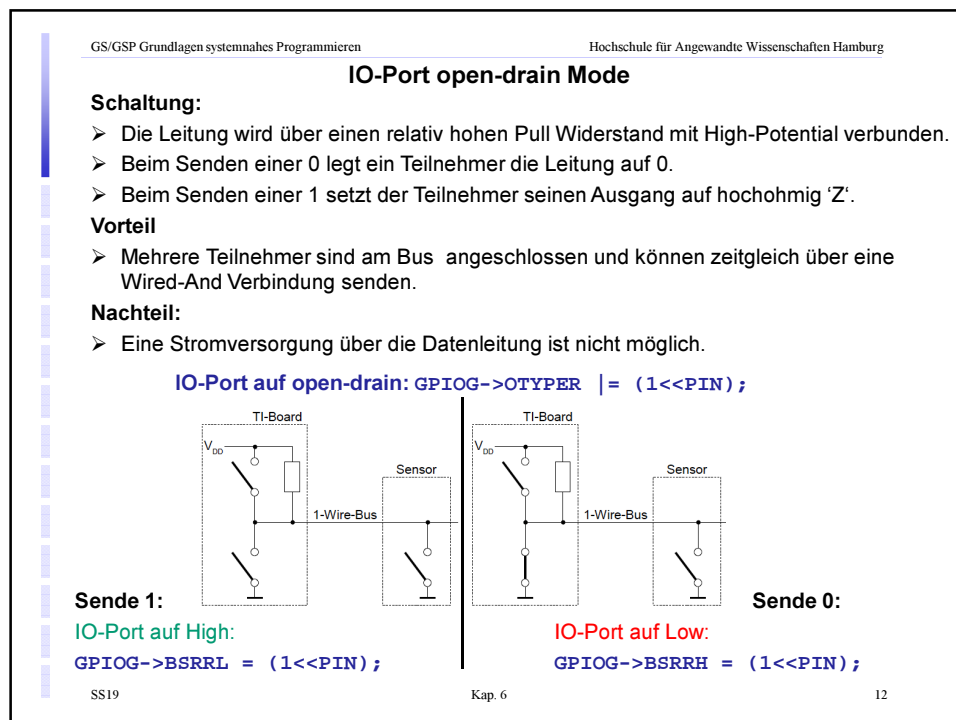
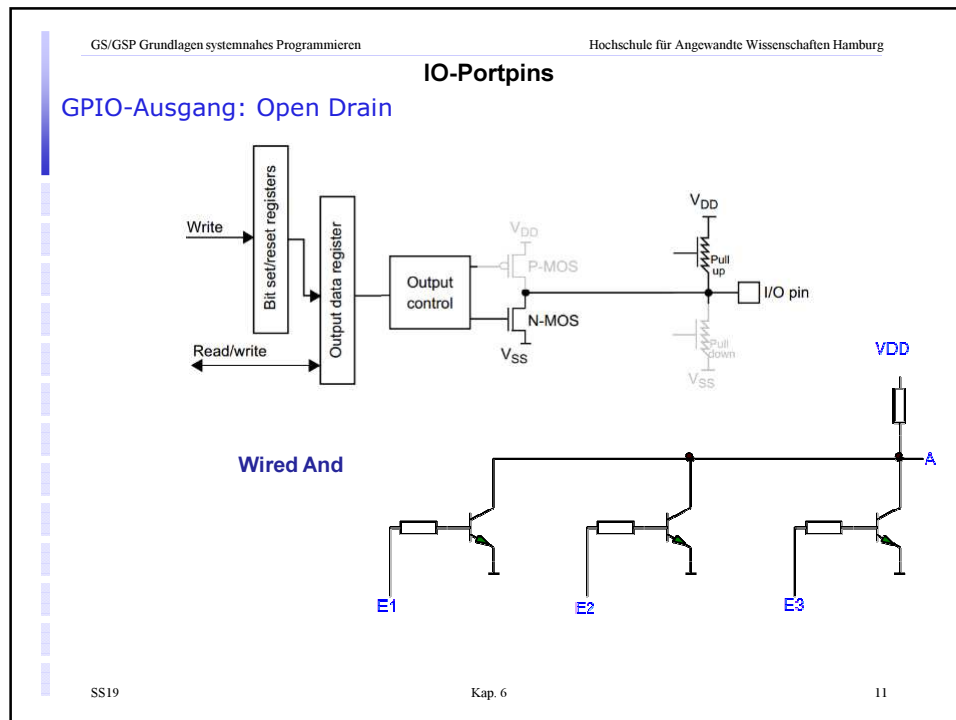
Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

SS19

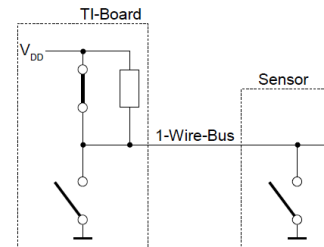
Kap. 6

10



IO-Port push-pull Mode**Schaltung:**

- Die Ausgangsleitung wird explizit getrieben.
- Beim Senden einer 0 zieht ein Teilnehmer die Leitung auf 0.
- Beim Senden einer 1 wird die Leitung mit VDD getrieben.

**Vorteil**

- Die Leitung kann im begrenzten Maß als Spannungsquelle dienen.

Nachteil:

- Kurzschluss, wenn mehrere Teilnehmer die Leitung mit unterschiedlichen Ausgangswerten treiben.

IO-Port auf push-pull:

```
GPIOG->OTYPER &= ~(1<<PIN);
```

IO-Port auf High:

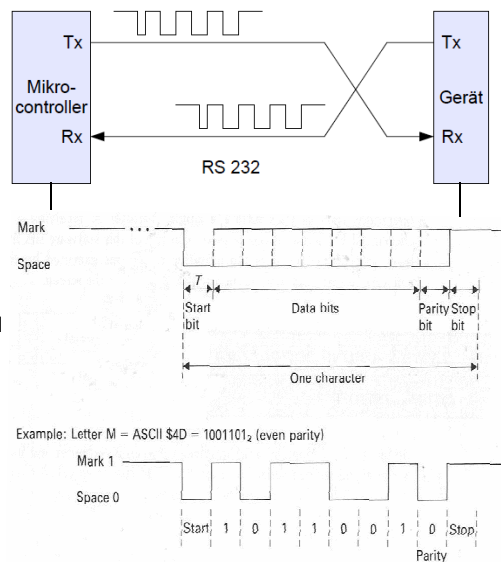
```
GPIOG->BSRRL = (1<<PIN);
```

Kapitel 6: I/O Programmierung Teil 2**Gliederung**

- Einführung
- Direct Digital Control
- General Purpose Input/Output am Beispiel des STM32F417ZG
- Anschluss an die Hardware
- Serielle Datenübertragung ←
- Prellen mechanischer Schalte

Serielle Datenübertragung

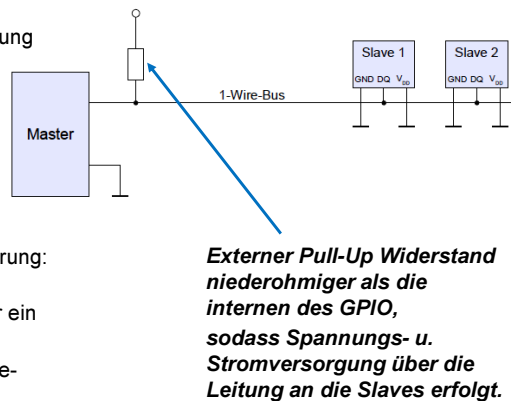
- Serielle Übertragung einzelner Bits
- Üblich: 2 Leitungen
 - Sendeleitung
 - Empfangsleitung
- Gleichzeitige Übertragung in beiden Richtungen im Full Duplex Modus.
- Übertragung beginnt mit **Startbit**.
Danach folgen im festen Zeitabstand die n Bits des Datenwortes mit **LSB First, Paritätsbit und Stopp-Bit**.
- Zeitabstand ist durch Baudrate, die Anzahl Symbole pro Sekunde, festgelegt.



SS19

1-Wire Bus

- Eine Signalleitung für Datenübertragung und Spannungsversorgung.
- Datenaustausch erfolgt nur in eine Richtung zur Zeit: **Half Duplex**.
- Gesendete Daten werden von allen Geräten empfangen: **Broadcast-Übertragung**.
- Übertragung vom Master an einen bestimmten Slave erfordert Adressierung: **Slaves mit eindeutigen Adressen**.
- Zu jedem Zeitpunkt darf maximal nur ein Gerät Daten senden.
- Zugriffsprotokoll gemäß Master-Slave-Prinzip:
 - Master bestimmt, wer als nächstes Daten senden darf.
 - Ein Slave sendet nur Daten, wenn er vom Master aufgefordert wird.



SS19

Kap. 6

16

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

1-Wire Bus: Wired-And Prinzip

Im Ruhezustand ist der Bus high

Master	Slave	Bus
		1
		0
		0
		0

Voraussetzung:
open-drain Mode

SS19

Kap. 6

17

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

1-Wire Bus: Lesen & Schreiben

Schreibe „1“
Bus auf Low setzen
6 µs warten
Bus freigeben
64 µs warten

Schreibe „0“
Bus auf Low setzen
60 µs warten
Bus freigeben
10 µs warten

Lese Bit
Bus auf Low setzen
6 µs warten
Bus freigeben
9 µs warten
Bit vom Bus lesen
55 µs warten

Reset
Bus auf Low setzen
480 µs warten
Bus freigeben
70 µs warten
Buszustand abfragen
410 µs warten

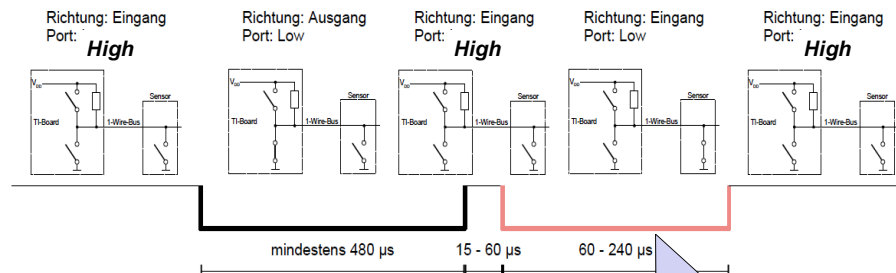
SS19

Kap. 6

18

Initialisierung mit Reset-Ablauf

➤ Bus im open-drain Modus betreiben.



Master erkennt am Low-Pegel (Presence-Pulse):

- **Mindestens ein Slave hat den Reset erkannt.**

Master erkennt nicht:

- **die Anzahl der Slaves, die den Reset ausführen.**

SS19

Transaktions-Sequenz für den Temperatursensor am 1-Wire Bus

1. Initialisierung mit Reset-Zyklus

2. ROM-Kommando ggf. mit Datentransfer z.B. Adressierung

3. Funktionskommando ggf. mit Datentransfer z.B. Temperatur lesen;

Weitere Sequenzen beginnen jeweils mit 1.

SS19

Kap. 6

20

Kommando-Sequenzen für den Temperatursensor am 1-Wire Bus

Nur ein Sensor am Bus

2. Skip ROM CCh
Adressierung ohne ROM-Code
3. Convert T 44h
2. Skip ROM
3. Read Scratchpad BEh
Byte 0 und Byte 1 bis 8 Byte lesen
2. Read ROM 33h
64-Bit ROM Code lesen
2. Match ROM 55h
Sensor mit ROM Code adressieren
3. Convert T

Mehrere Sensoren am Bus

2. Search ROM F0h
Alle ROM-Codes sukzessive identifizieren
2. Match ROM
3. Convert T
2. Match ROM
3. Read Scratchpad

Auslesen des ROMs des Temperatursensors

- Bus im open-drain Modus betreiben.

Reset
Sende Read ROM command (0x33)
Read Family Code (1 Byte)
Read Serial Number (6 Byte)
Read CRC (1 Byte)

64-Bit 'Registration' ROM Number					
MSB				LSB	
8-Bit CRC		48-Bit Serial Number		8-Bit Family Code	
MSB	LSB	MSB	LSB	MSB	LSB

- Funktioniert, wenn nur ein Sensor am Bus angeschlossen ist!
Ansonsten: Search ROM Command verwenden.

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

Durchführung einer Temperaturmessung

Wähle Slave aus			Führe Messung durch			Wähle Slave aus			Lese Ergebnis		
Reset	Sende Match ROM command (0x55)	Sende ROM code (8 Bytes)	Sende Convert T command (0x44)	Versorge Sensor aus niederohmiger Spannungsquelle	Warte auf Beendigung der Messung (mindestens 750 msec)	Reset	Sende Match ROM command (0x55)	Sende ROM code (8 Bytes)	Sende Read Scratchpad command (0xBE)	Lese 9 Bytes (vollständiges Scratchpad inklusive Checksumme)	
open-drain mode			push-pull mode			open-drain mode					
Reset	Sende Skip ROM command (0xCC)					Reset	Sende Skip ROM command (0xCC)				

Richtung: Ausgang Port: High

Alternative, wenn nur ein Sensor am Bus angeschlossen ist

SS19

Kap. 6

23

GS/GSP Grundlagen systemnahes Programmieren

Hochschule für Angewandte Wissenschaften Hamburg

Bestimmung der Temperatursensoren am Bus

➤ Beispiel eines Ablaufs zum Low-Level Protokoll

Starte Algorithmus		Führe Algorithmus aus									
Reset	Sende Search ROM command (0xF0)	Empfange Bit 0	Empfange Komplement Bit 0	Sende Bit 0	Empfange Bit 1	Empfange Komplement Bit 1	Sende Bit 1	...	Empfange Bit 63	Empfange Komplement Bit 63	Sende Bit 63

SS19

Kap. 6

24

GS/GSP Grundlagen systemnahes Programmieren
Hochschule für Angewandte Wissenschaften Hamburg

Bestimmung der Temperatursensoren am Bus

1. Schritt: Alle Sensoren senden gleichzeitig ihr erstes Bit
Signal auf dem Bus ist die UND-Verknüpfung dieser Bit
2. Schritt: Alle Sensoren senden gleichzeitig das Inverse ihres ersten Bits
Signal auf dem Bus ist wiederum die UND-Verknüpfung dieser Bit
3. Schritt: Analyse
 - Master hat die Bits '0' und '1' empfangen:
Das erste Bit aller Sensoren ist '0'
Master sendet eine '0' als Bestätigung
 - Master hat die Bits '1' und '0' empfangen:
Das erste Bit aller Sensoren ist '1'
Master sendet eine '1' als Bestätigung
 - **Master hat die Bits '0' und '0' empfangen:**
Das erste Bit der Sensoren ist nicht einheitlich.
Master sendet ein Bit zurück zur Auswahl der Sensoren.
Mit denen die Suche fortgesetzt werden soll.
Alle anderen Sensoren werden inaktiv.
 - Master hat die Bits '1' und '1' empfangen:
Fehler ist aufgetreten, Verbindung zu den Sensoren ist unterbrochen.

Wiederhole die Schritte 1 bis 3 mit den noch aktiven Sensoren solange, bis alle 64 Bits ausgewertet sind.

SS19
Kap. 6
25

GS/GSP Grundlagen systemnahes Programmieren
Hochschule für Angewandte Wissenschaften Hamburg

Bestimmung der Temperatursensoren am Bus

1. Schritt: Alle Sensoren senden gleichzeitig **ihr erstes Bit**
Signal auf dem Bus ist die UND-Verknüpfung dieser Bit
2. Schritt: Alle Sensoren senden gleichzeitig das **Komplement ihres ersten Bits**
Signal auf dem Bus ist wiederum die UND-Verknüpfung dieser Bit
3. Schritt: Analyse

Bit (true)	Bit (complement)	Information Known
0	0	There are both 0s and 1s in the current bit position of the participating ROM numbers. This is a discrepancy.
0	1	There are only 0s in the bit of the participating ROM numbers.
1	0	There are only 1s in the bit of the participating ROM numbers.
1	1	No devices participating in search.

Wiederhole die Schritte 1 bis 3 mit den noch aktiven Sensoren solange, bis alle 64 Bits ausgewertet sind.

SS19
Kap. 6
26

GS/GSP Grundlagen systemnahes Programmieren		Hochschule für Angewandte Wissenschaften Hamburg
Verzweigung der Schreib-Bit Auswahl im Fall bit = \negbit = 0		
Table 3. Search Path Direction		
Search Bit Position vs Last Discrepancy	Path Taken	
=	2. Durchgang: next 1 hinten	Take the '1' path
<	2. Durchgang: next 1	Take the same path as last time (from last ROM number found)
>	1. Durchgang: first	Take the '0' path

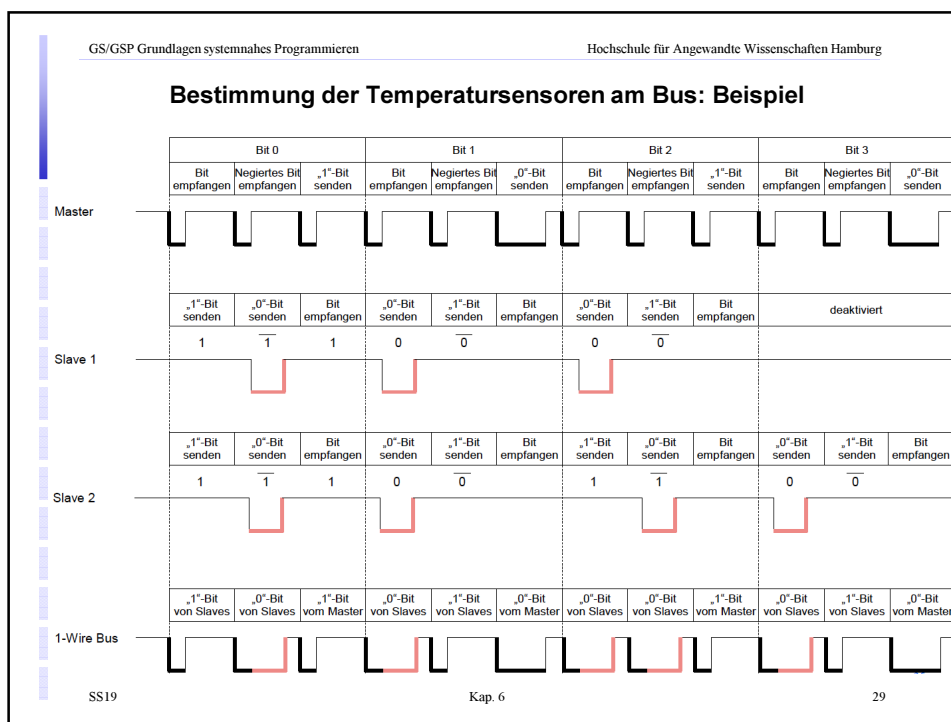
Bestimmung der Temperatursensoren am Bus: Beispiel

The diagram shows a binary tree structure representing the determination of temperature sensors on a bus. The root node branches into 0 and 1. Node 0 branches into 00 and 10, which further branch into 000, 100, 010, and 110 respectively. Node 1 branches into 01 and 11, which further branch into 001, 101, 011, and 111 respectively. The path from the root to node 01 is highlighted in red. The path from node 01 to its left child 001 is highlighted in green. The path from node 01 to its right child 101 is highlighted in red. The leaf nodes 001 and 101 are labeled 'Slave 1' and 'Slave 2' respectively.

SS19

Kap. 6

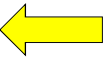
28



GS/GSP Grundlagen systemnahes Programmieren Hochschule für Angewandte Wissenschaften Hamburg

Kapitel 6: I/O Programmierung Teil 2

Gliederung

- Einführung
- Direct Digital Control
- General Purpose Input/Output (am Beispiel von STM32F417ZG)
- Anschluss an die Hardware
- Serielle Datenübertragung
- Prellen mechanischer Schalter 

SS19 Kap. 6 30

Situation

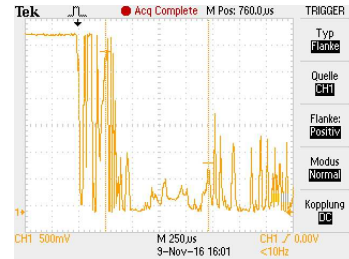
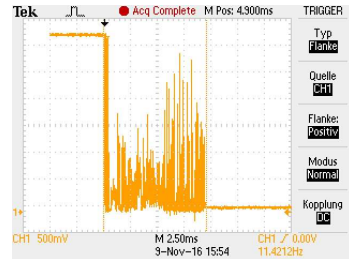
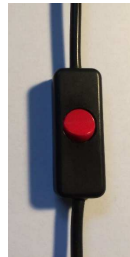
- Beim Schließen oder Öffnen schwingen die Kontakte von mechanischen Schaltern.
 - Sie schließen und öffnen mehrmals schnell hintereinander, bis sie schließlich offen bzw. geschlossen bleiben.
- Dieser Vorgang heißt **Prellen (bounce)**.
- In Abhängigkeit vom Schalter selbst und seinem Alter dauert die Phase des Prellens wenige Mikrosekunden bis einige Millisekunden.
 - In dem Prellintervall schwankt der „Wert des Schalters“ zwischen auf und zu.
- **Achtung:** Nachmessen alleine reicht nicht - ein Blick ins Datenblatt ist notwendig.
 - Bei mehreren Schaltvorgängen prellt der Schalter unterschiedlich lange.
 - Mit der Zeit (der Alterung) wird das Prellverhalten schlechter – dauert der Prellvorgang länger, bis ein schalterspezifisches Maximum erreicht ist.
- Je nach Hardwareaufbau können auch elektronische Schaltungen schwingen.

Lösungsansätze

- Entprellen durch Hardware
- Entprellen durch Software
- Entprellen durch Software wird vorgezogen
 - da dies in der Regel leicht realisierbar ist
 - da Bauteile etc. eingespart werden und somit die Stückkosten fallen.

Ein Blick auf einige Schalter

Ein Lichtschalter der „1 € Klasse“.

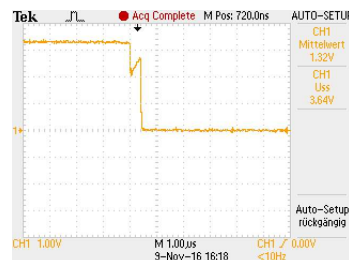
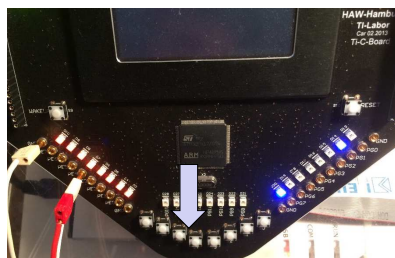


Ergebnisse

- Prellvorgang dauert ca. 10 ms
- An Anfang schwankt der Ausgang der Schalter zwischen V_{ref} und 0V

Ein Blick auf einige Schalter

Ein Taster des TI-Boards

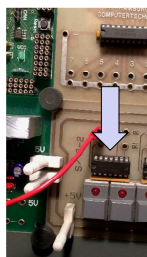


Ergebnisse

- Der Taster prellt „etwas“.
- Eine Messung alleine reicht nicht.
- Wir wissen nicht, wie der Schalter sich in 2 Jahren verhält, wenn er noch 100000 Mal betätigt wurde ...
- Hier sieht man die Notwendigkeit des Blicks ins Datenblatt.

Ein Blick auf einige Schalter

Ein Schalter mit HW Entprellung



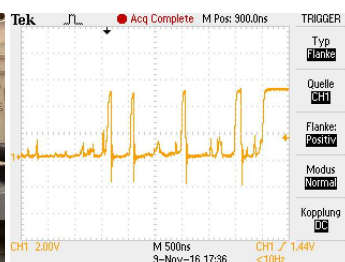
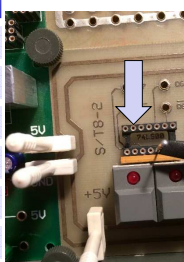
SS19

Kap. 6

35

Ein Blick auf einige Schalter

Der selbe Schalter ohne HW Entprellen (der Schalter ist relativ „alt“)



Ergebnisse

- Hohe Schwankungen bezüglich Zeit und Amplitude
- Prelldauer einige Mikrosekunden

SS19

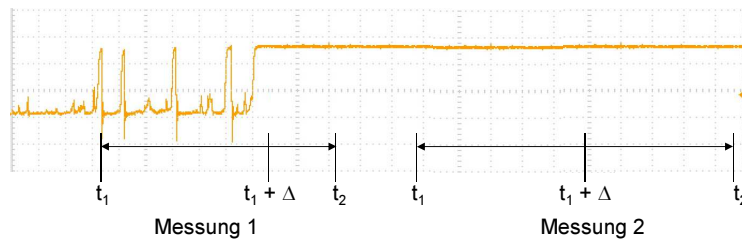
Kap. 6

36

Entprellen durch SW

Genereller Ansatz:

- Lese den Ausgang des Schalters zum Zeitpunkt t_1
- Lese den Ausgang des Schalters zum Zeitpunkt $t_2 > t_1 + \Delta$ (Δ = Zeitintervall des Prellens)
- Im Prinzip muss t_1 nicht an die „erste“ Änderungen des Schalterausgangs gebunden sein - aber so erkennt man die Änderung des Schalters schneller.



- Was passiert, wenn $t_1 + \Delta < \text{Zeit zwischen zwei mal Schalter drücken} < t_2$?

Umsetzung in SW

Einfaches Verfahren: Lese den Schalter an einer beliebigen Stelle im Code

```
#define ANZAHL_LESEVERSUCH 5

...
wertSchalter = undefined;
for (int i = 0 ; i < ANZAHL_LESEVERSUCH; i++) {
    s1 = lese Wert des Ausgangs des Schalters ein;
    Warte mindestens die Zeitspanne, die das Prellen
        maximal dauert; Realisation mit Polling und
        Timer Modul
    if (s1 == lese Wert des Ausgangs des Schalters ein) {
        // Schalter hat den Wert s1
        wertSchalter = s1;
        break;
    }
    // Schalter ist gerade in der Prellphase (oder defekt)
}
```

Umsetzung in SW

Alternatives Verfahren, basierend auf einer ISR

- Der Nachteil des vorgestellten Verfahrens ist, dass die erste Änderung des Schalters nicht erkannt wird.
- Stattdessen wird zu irgendeinem Zeitpunkt der Wert des Schalters überprüft.
- Somit wird eine Änderung des Schalters nicht zum „frühest möglichen Zeitpunkt“ erkannt.
- **Verfahren 1:**
 - Die Änderung eines Eingangs und somit die Änderung des Schalterausgangs wird an eine ISR gebunden (s. Kapitel 8).
 - Die ISR führt die Berechnung des vorgestellten Verfahrens durch. Das findet „sofort“ statt, wenn die Schalteränderung aufgetreten ist.
- **Verfahren 2:**
 - Die Änderung eines Eingangs und somit die Änderung des Schalterausgangs wird an eine ISR gebunden (s. Kapitel 8).
 - Diese ISR speichert den aktuellen Wert des Schalterausgangs.
 - Sie startet einen Timer, so dass eine weitere ISR gestartet wird, wenn die Zeitspanne $\Delta \text{des } \tau_{\text{μερσ}}$ vergangen ist.
 - Die zweite ISR führt dann Vergleich der Schalterwerte durch