Inhaltsübersicht

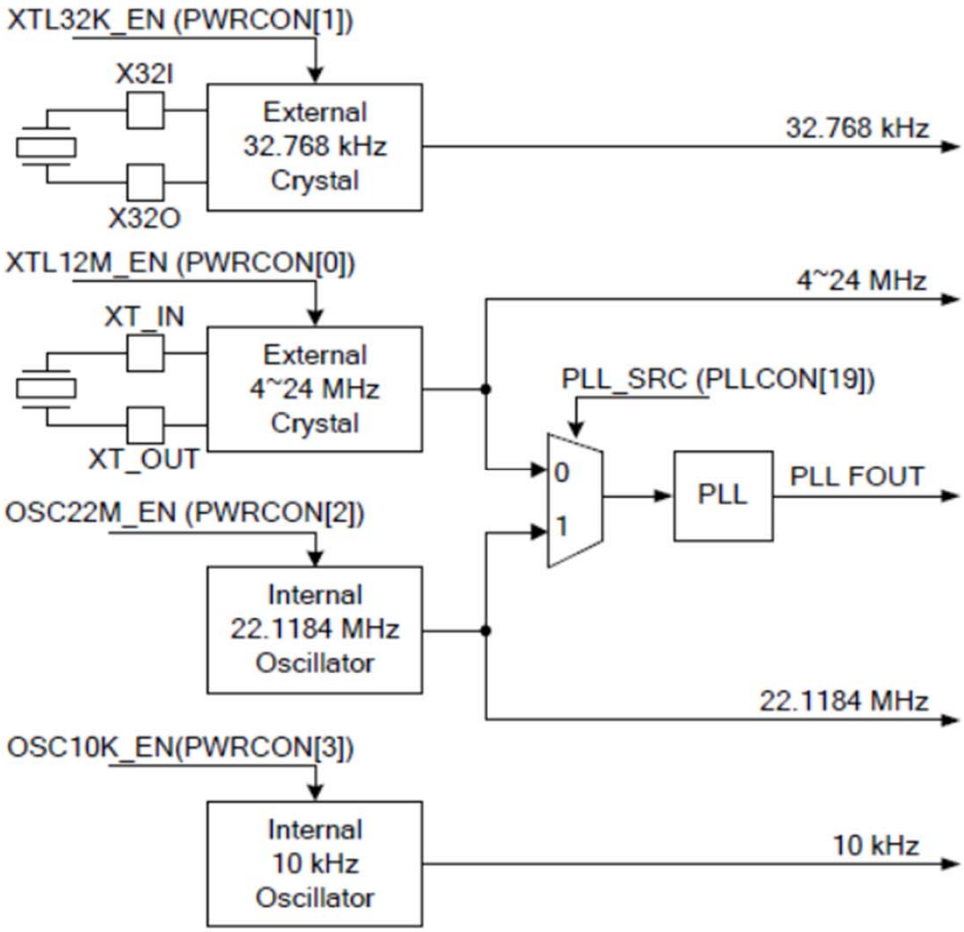
Kapitelübersicht

I. Taktsystem und Timer
II. UART
III. SPI und LCD
IV. A/D-Wandler

Freischalten von Takten

- Der Chip verfügt über verschiedene Taktquellen
  - Externer 12 MHz Oszillator
  - Externer 32 kHz Oszillator (für Power-Down-Modi)
  - Interner Oszillator (22,1148 MHz) und PLL
  - System startet zunächst mit internem 22 MHz Takt.
- Die Taktquellen müssen vor Nutzung zunächst freigeschaltet werden. Wir nutzen den 12 MHz Takt.
- Hierzu kann die Funktion **`DrvSystem_ClkInit(void)`** benutzt werden:
  - Schaltet auf 12 MHz, sofern vorhanden.
  - Anderenfalls wird interner 22 MHz Takt ausgewählt.

Taktquellen



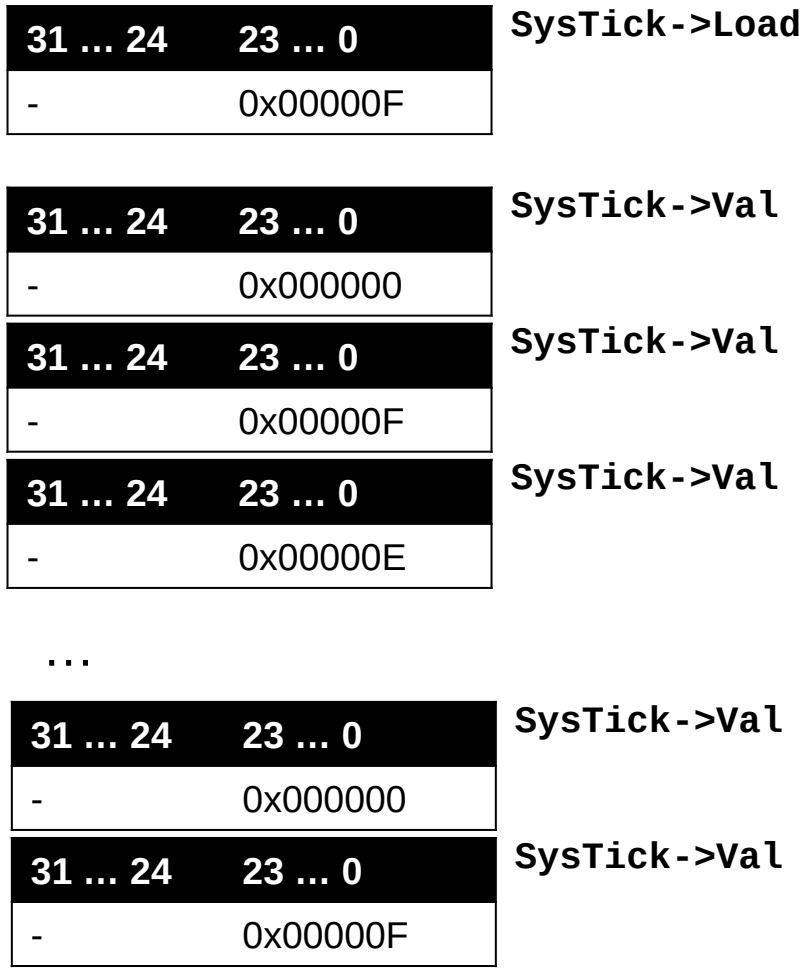
Quelle: Technical Reference Manual NUC130

Der SysTick Timer

- Der Cortex-M0 verfügt über einen „Timer“, der als „SysTick“-Timer bezeichnet wird.
- Der SysTick ist ein 24-Bit Abwärtszähler und kann beim Nulldurchgang einen Interrupt erzeugen.
- Vorgesehen ist der SysTick insbesondere zur Unterstützung von Betriebssystemen.
- Wenn kein Betriebssystem benutzt wird, kann der Timer auch für andere Aufgaben verwendet werden, z.B. für Zeitverzögerungen.

Arbeitsweise des SysTick-Timers

- Man lädt das „Reload“-Register **(SysTick->LOAD)** mit dem Startwert.
- Das Zählerregister **(SysTick->VAL)** dekrementiert und beim Nulldurchgang wird der Reload-Wert wieder geladen und ein „Flag“ gesetzt (= Interrupt).
- Zu Beginn wird daher das Zählerregister ebenfalls auf Null gesetzt.



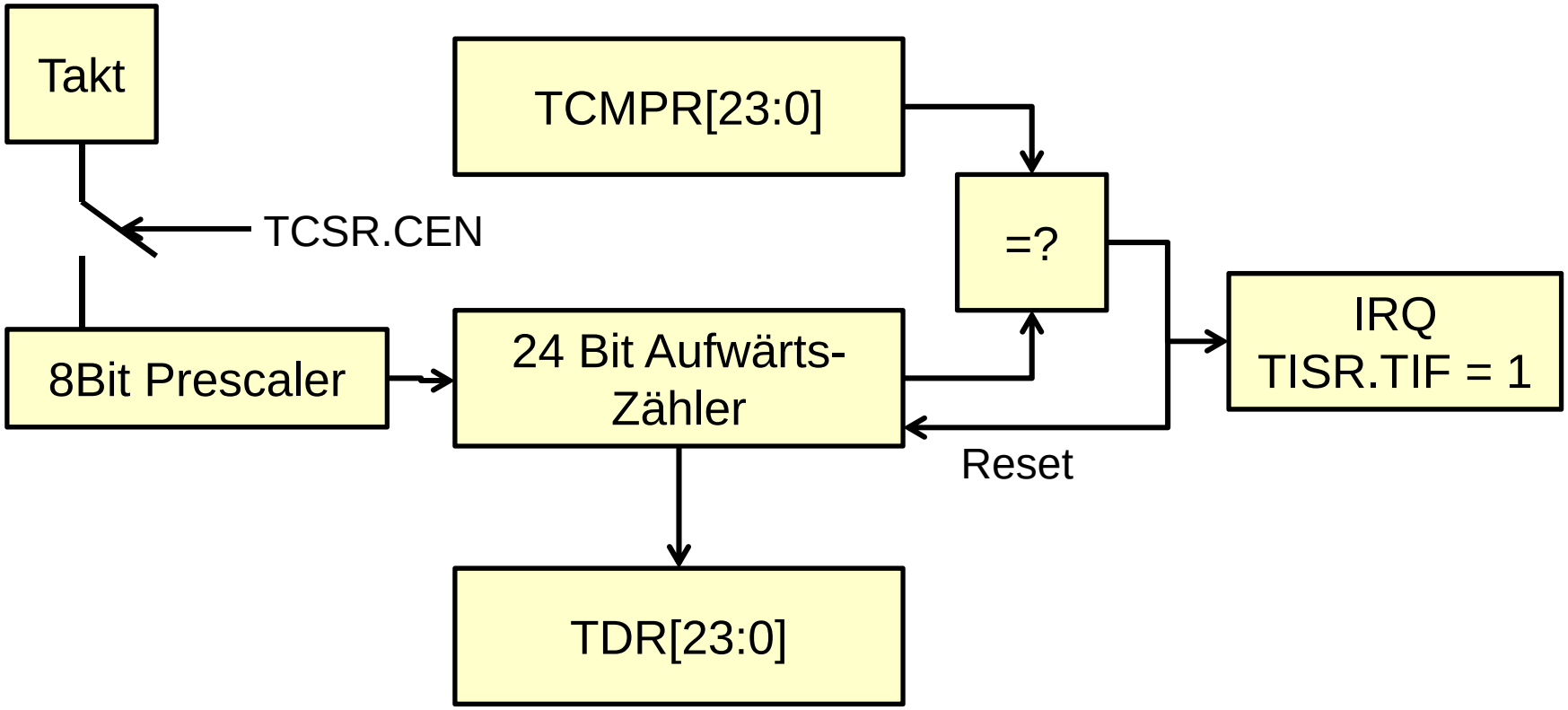
Weitere Timer im NUC130

- Im NUC130 sind vier weitere Timer verfügbar (Timer0 – Timer3).
- Jeder Timer besteht aus einem 24-Bit-Timer (Aufwärtszähler!) und einem 8-Bit-Vorteiler (pre-scale counter)
- Jeder Timer kann Interrupts auslösen
- Verschiedene Modi für Anwendungen wie
  - Zeitverzögerungen
  - Ereignisse an externen Pins zählen
  - …

Timer 0 im „One-Shot“-Modus

- Geeignet für Zeitverzögerungen
- Wenn CEN = 1 gesetzt wird, dann startet der Timer.
- Der Prescaler teilt den Takt um den entsprechenden Faktor. Der Timer inkrementiert das Register TDR entsprechend.
- Ist TDR= TCMPR dann wird das TIF-Flag gesetzt.
- Der Timer wird rückgesetzt (TDR = 0) und automatisch deaktiviert (CEN = 0)

Timer 0 Blockdiagramm



Timer 0 Registerübersicht

R: read only, **W**: write only, **R/W**: both read and write

Register	Offset	R/W	Description	Reset Value
TMR_BA01 = 0x4001_0000 TMR_BA23 = 0x4011_0000				
TCSR0	TMR_BA01+0x00	R/W	Timer0 Control and Status Register	0x0000_0005
TCMPR0	TMR_BA01+0x04	R/W	Timer0 Compare Register	0x0000_0000
TISR0	TMR_BA01+0x08	R/W	Timer0 Interrupt Status Register	0x0000_0000
TDR0	TMR_BA01+0x0C	R	Timer0 Data Register	0x0000_0000

- TCSR.CEN (Bit 30): Start (=1), Stop (= 0), Bit wird im „One-Shot“-Modus automatisch auf 0 rückgesetzt
- TCSR.MODE (Bits 28:27): Modus (00 = One-Shot)
- TCSR.CRST (Bit 26): Timer Reset (=1), TDR = 0
- TCSR.PRESCALE (Bits 7:0): Wert für den Prescaler
- TISR.TIF (Bit 0): Wird gesetzt wenn TDR = TCMPR und kann Interrupt auslösen. Muss durch Schreiben einer 1 rückgesetzt werden.

Quelle: Technical Reference Manual NUC130

Berechnung der Zeitverzögerung

- Zeitdauer bis zum Timer-Überlauf:

$$T = \frac{1}{f_{clock}} \cdot (Presc + 1) \cdot TCMPR$$

- Beispiel: f_{clock} = 12 MHz, Presc = 11

$$T = \frac{1}{12MHz} \cdot 12 \cdot TCMPR = 1\mu s \cdot TCMPR$$

Bibliotheksfunktionen für „One-Shot“-Modus

- **`void DrvTimer0_Init(void)`**
  - Reset des Timers
  - Prescaler-Wert für eine Mikrosekunde
  - One-Shot-Modus
- **`void DrvSystem_Delay(uint32_t us)`**
  - TCMPR = us, Anzahl der Mikrosekunden bis Überlauf
  - Timer starten
  - Warten bis zum Überlauf
  - Rücksetzen des TIF-Flags

Beispiel Lauflicht

```
#include "BoardConfig.h"
#include "Driver_M_Dongle.h"
#include "init.h"

int main (void){
   uint32_t i;

   DrvSystem_ClkInit();   //Setup clk system
   Board_Init();          //Initialize peripherals
   DrvTimer0_Init();      //Initialize Timer 0

   while(1) {
      for(i=LED0; i<LED7+1; i++){
         DrvGPIO_ClearBit(E_GPE, i);//Clear Bit, LED i on
         DrvSystem_Delay(1000000);  //wait 1 sec
         DrvGPIO_SetBit(E_GPE, i);  //Set Bit, LED i off
         DrvSystem_Delay(1000000);  //wait 1 sec
      }
   }
}
```

Unterschied zur Funktion DrvSystem_Wait_us

- Die Funktion **DrvSystem_Wait_us** benutzt nicht den Timer.
- Genauigkeit hängt von der Implementierung im Maschinencode ab, daher ist die Timer-Lösung besser.

```
void DrvSystem_Wait_us(uint32_t ui32Delay)
{
    ui32Delay *= gCyclesPerUs;

    while(ui32Delay != 0)
    {
        ui32Delay --;
    }
}
```

Kapitelübersicht

I. Taktsystem und System-Timer

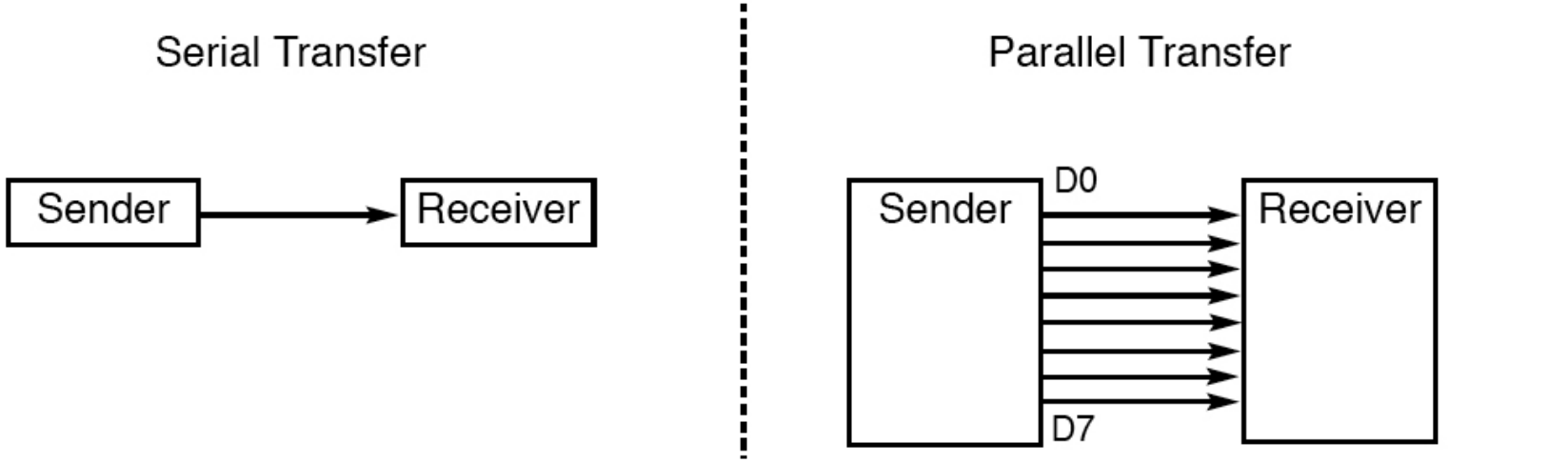
II. UART

III. SPI und LCD

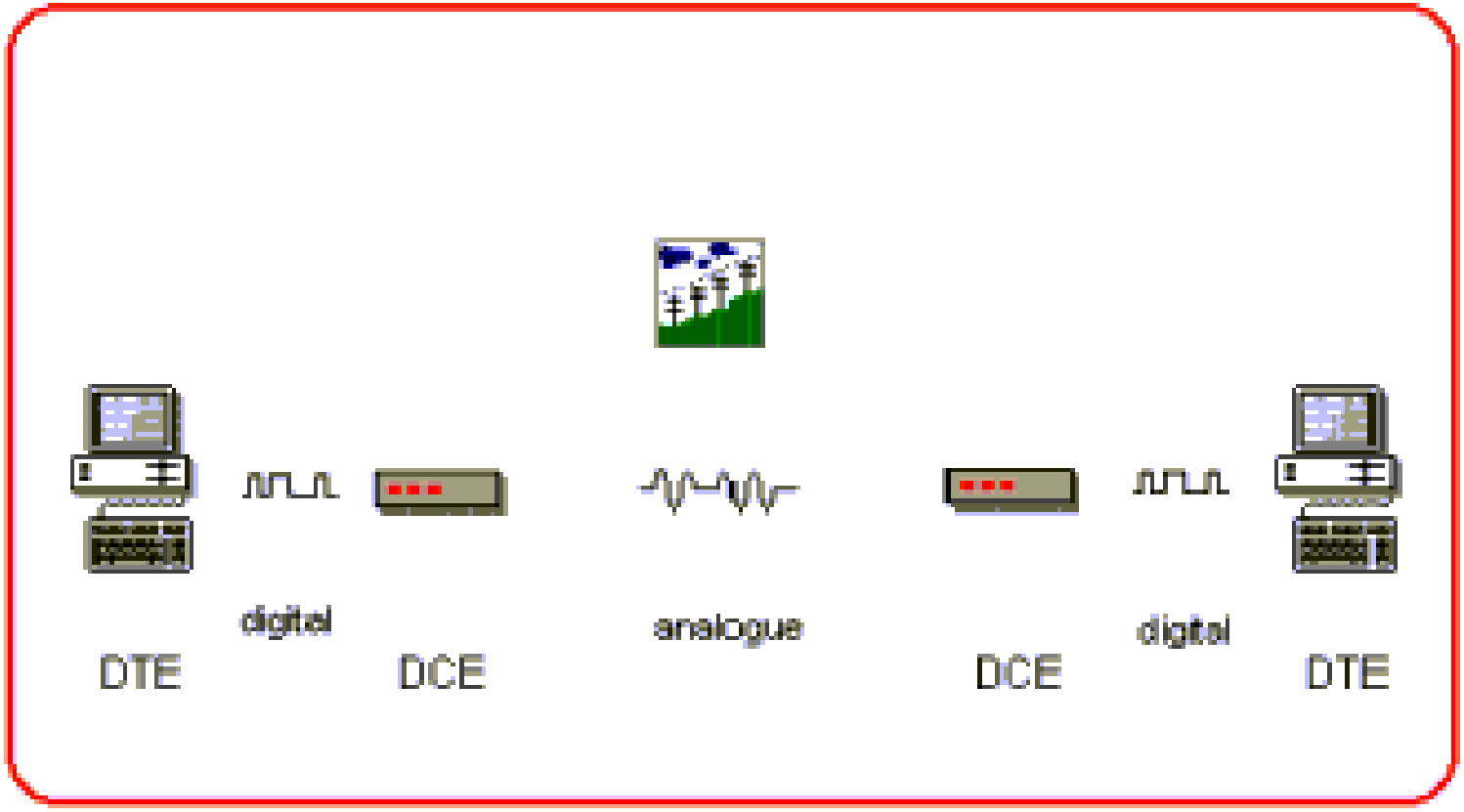
IV. A/D-Wandler

Serieller vs. paralleler Transfer von Daten

- Die parallele Übertragung von Daten ist schneller als die serielle (bitweise) Übertragung.
- Die serielle Übertragung ist einfacher und kostengünstiger zu realisieren.
- Parallele Übertragung findet daher häufig bei kurzen Distanzen statt (z.B. Drucker, Festplatte), für lange Verbindungen werden serielle Verfahren benutzt (z.B. Internet, Modem über Telefonleitung)



Beispiel RS-232



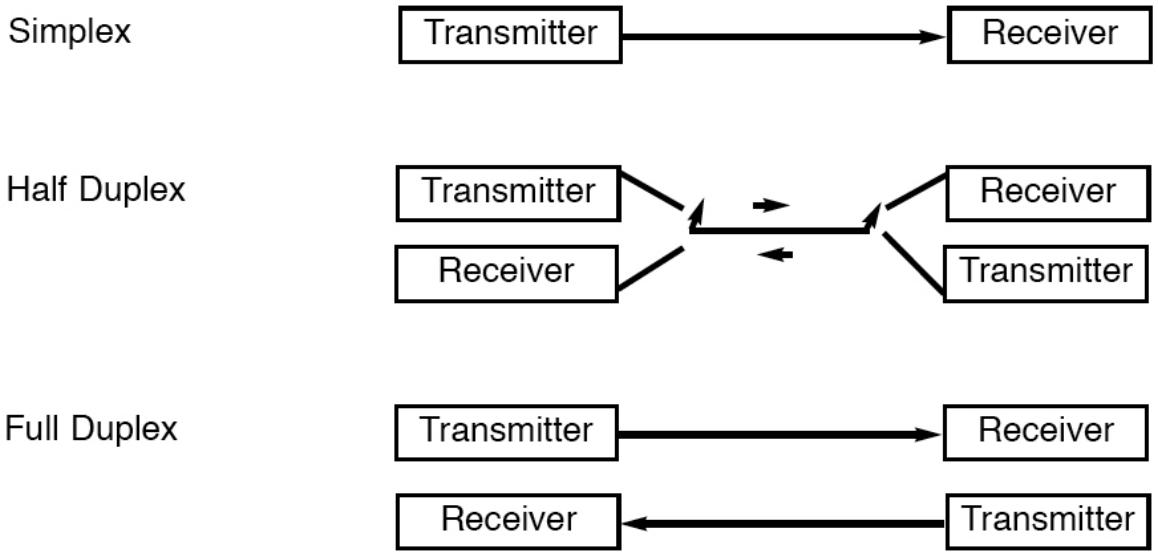
DTE: Data Terminal Equipment, DCE: Data Communication Equipment (Modem)

Serielle Übertragung von Daten

- Zu sendende parallele Daten (Bytes) im Prozessor müssen „serialisiert“ werden
  - „Parallel-In Serial-Out“ Schieberegister
- Empfangene, serielle Daten müssen parallelisiert werden
  - „Serial-In Parallel-Out“ Schieberegister
- Diese Umsetzung wird durch einen UART (Universal Asynchronous Receiver Transmitter) für das RS-232-Protokoll realisiert.

Halb- und Voll-Duplex-Übertragung

- Simplex: Kommunikation nur in eine Richtung
- Halb- und Voll-Duplex:
  - Kommunikation in beide Richtungen
  - Halb-Duplex: Kommunikation findet pro Zeitschritt nur in eine Richtung statt
  - Voll-Duplex:
    Kommunikation findet jederzeit in beide Richtungen statt.



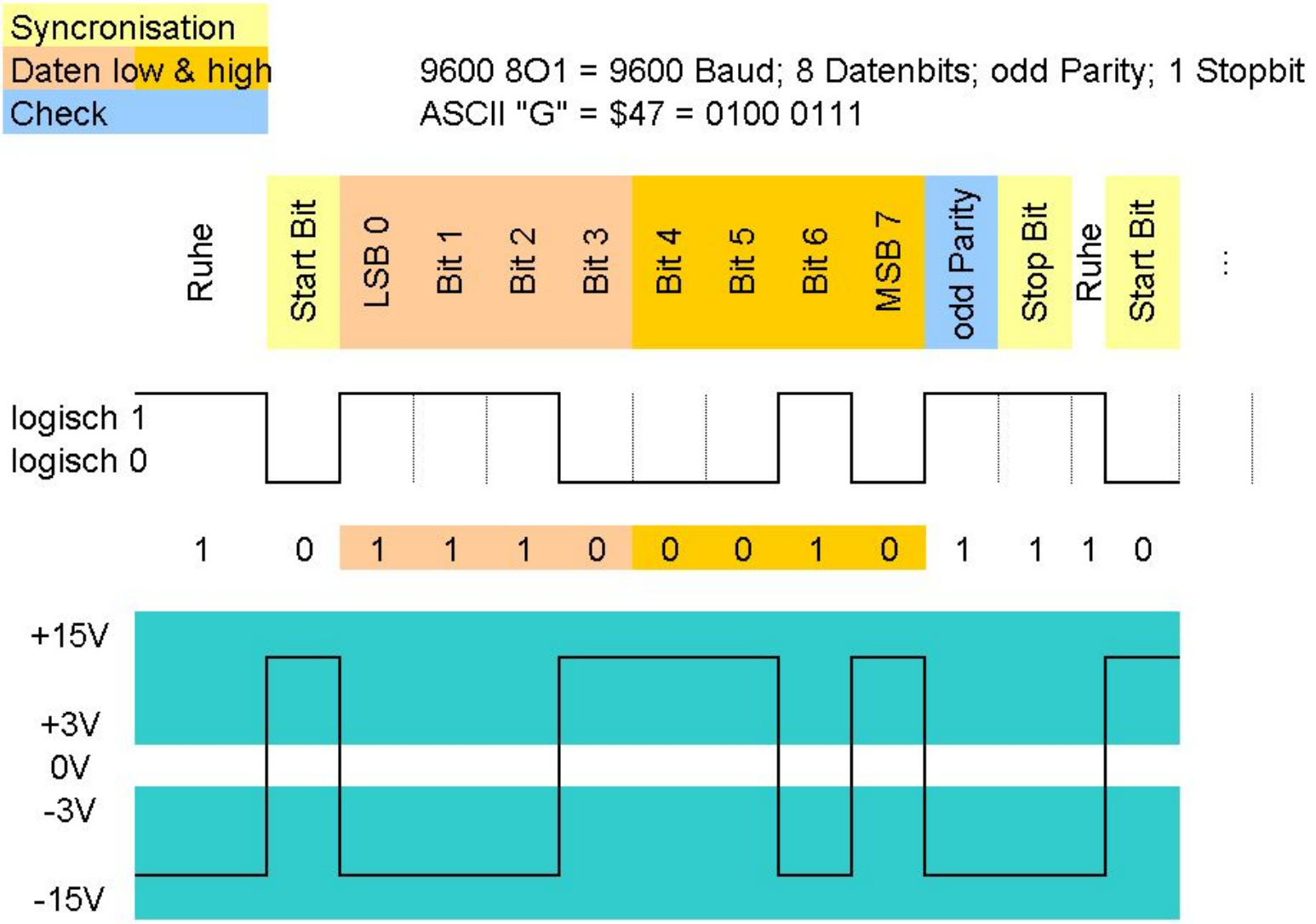
RS-232-Protokoll

- Protokoll: Menge von Regeln auf die sich Sender und Empfänger einigen, um Daten zu übertragen.
- RS-232 (EIA-232): Asynchrones, serielles Protokoll
- Jedes Wort wird durch ein „Startbit“ begonnen und durch ein oder zwei „Stopbits“ beendet (Frame oder Rahmen).
- Es werden „Worte“ übertragen, die 5-9 Bit umfassen. Häufig 8 Bit (Byte) mit optionalem 9. Bit zur Fehlererkennung (Paritätsbit: O = Odd, ungerade; E = Even, gerade; N = No parity).
  Häufig: „8N1“ $\rightarrow$ 8 Datenbits, kein Paritätsbit, 1 Stopbit
- Paritätsbit: Dieses ergänzt die Datenbits zur entsprechenden Parität. Haben wir z.B. „8O1“ (8 Datenbits, Parity Odd, 1 Stopbit) und das Daten-Byte besteht aus vier 1en und vier 0en, so wird das Parity-Bit auf 1 gesetzt, damit über alle 9 Bits eine ungerade Parität entsteht.

RS-232-Protokoll (2)

- Es wird kein Takt mitgeliefert ($\rightarrow$ synchrone Verfahren, siehe SPI), daher synchronisieren sich Sender und Empfänger mit Hilfe des Startbits für jedes Wort von neuem.
- Die Taktraten (Baudrate) von Sender und Empfänger müssen daher bis auf wenige Prozent (< 5%) übereinstimmen.
- Die Nutzdaten werden unverändert, also ohne zusätzliche Synchronisierungsinformation, übertragen. (NRZ-Codierung: Non-Return-to-Zero).
- Daten werden Voll-Duplex übertragen, daher drei Leitungen: RxD, TxD, Masse.

RS-232-Protokoll (2)



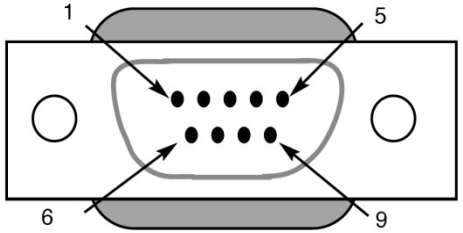
RS-232-Protokoll (3)

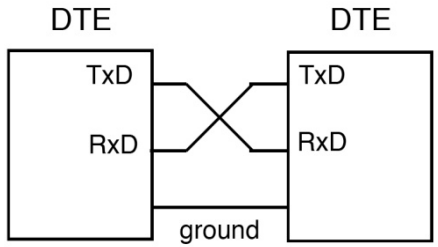
- RS-232 ist eine Spannungsschnittstelle.
- Daten werden durch wechselnde Spannungspegel übertragen.
- Negative Logik:
  - Logische 1 („Mark“): -12 V ... –3V
  - Logische 0 („Space“): +3 V ... +12 V
  - verbotener Bereich: -3 V ... +3 V
- Da der µC nur Pegel zwischen 0 und 5 V liefert, müssen die Pegel in einem zusätzlichen IC gewandelt werden (z.B. Maxim MAX202).

Verbindung von Geräten

- Häufig werden Geräte (z.B. µC ⇔ PC) über 9-polige Sub-D-Stecker und Buchsen verbunden.
- Wenn nur Endgeräte (z.B. µC ⇔ PC) verbunden werden, müssen RxD und TxD gekreuzt werden („Nullmodemkabel“).

Pin	Beschreibung
1	DCD: Data Carrier Detect
2	RxD: Received Data
3	TxD: Transmitted Data
4	DTR: Data Terminal Ready
5	GND: Masse
6	DSR: Data Set Ready
7	RTS: Request To Send
8	CTS: Clear To Send
9	RI: Ring Indicator





Flußsteuerung

- Das RS-232-Protokoll bietet optional die Möglichkeit der Datenflußsteuerung:
  - Geräte signalisieren sich über zusätzliche Signale ihren Status (z.B. CTS, RTS)
  - Dies wird als „Hardware-Handshake“ bezeichnet
- In der Mikrocontrollertechnik wird der „Hardware-Handshake“ nicht verwendet:
  - Nur drei Leitungen notwendig: RxD, TxD, GND
  - Optional ist ein „Software-Handshake“ durch Senden von speziellen Zeichen möglich (Xon/Xoff)

Baudrate und Bitrate

- In der Übertragungstechnik ist es möglich, in einem „Symbol“ mehrere binäre Bits zu übertragen (→ Modulation, z.B. DSL). Daher kann mit einer bestimmten „Symbolrate“ oder „Baudrate“ eine vielfach höhere „Bitrate“ oder „Datenrate“ erzielt werden.
- Da bei RS232 binäre Symbole übertragen werden, ist die Baudrate gleich der Bitrate und damit die Anzahl der übertragenen Bits pro Sekunde (bit/s).
- Einige gebräuchliche Bitraten/Baudraten für RS232:

Bitrate in bit/s	Bitdauer
2400	417 µs
4800	208 µs
9600	104 µs
19200	52 µs
38400	26 µs

vgl. USB 3.0:
bis zu 4 Gbit/s

Leitungslängen

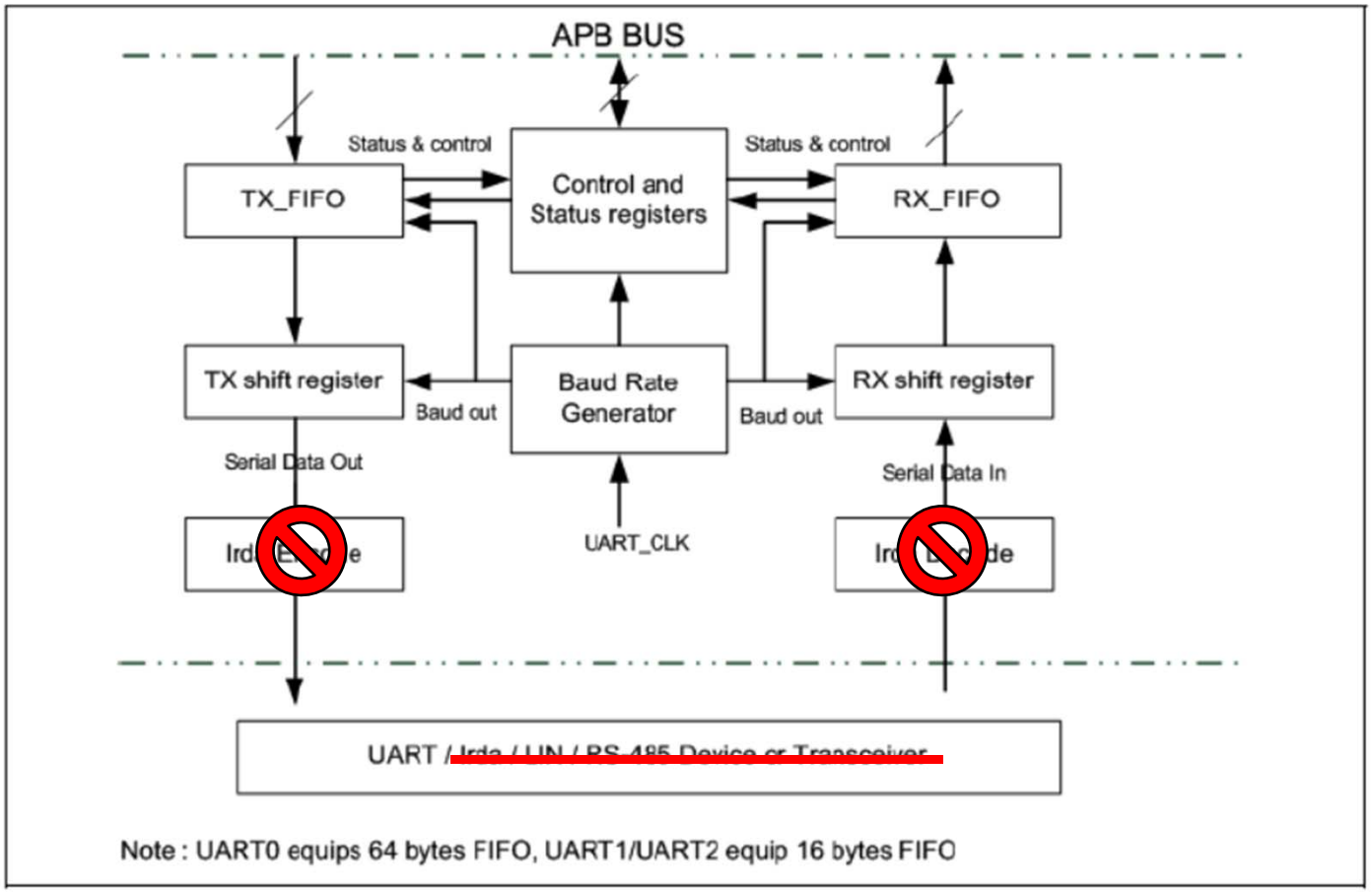
- Bei hohen Bitraten und langen Leitungen können Reflexionen am Leitungsende entstehen, die das Signal stören.
- Störungen können auch von anderen Geräten induktiv und kapazitiv einkoppeln.
- Leitungslängen sind daher begrenzt und die maximale Leitungslänge abhängig von der Bitrate und der Art des Kabels (Kapazität).
- Andere Standards (z.B. RS-485, USB) kommen durch verschiedene Maßnahmen zu größeren Leitungslängen bzw. höheren Bitraten.

Bitrate in bit/s	Leitungslänge (c.a.)
2400	900 m
4800	300 m
9600	152 m
19200	15 m

UARTs im NUC 130

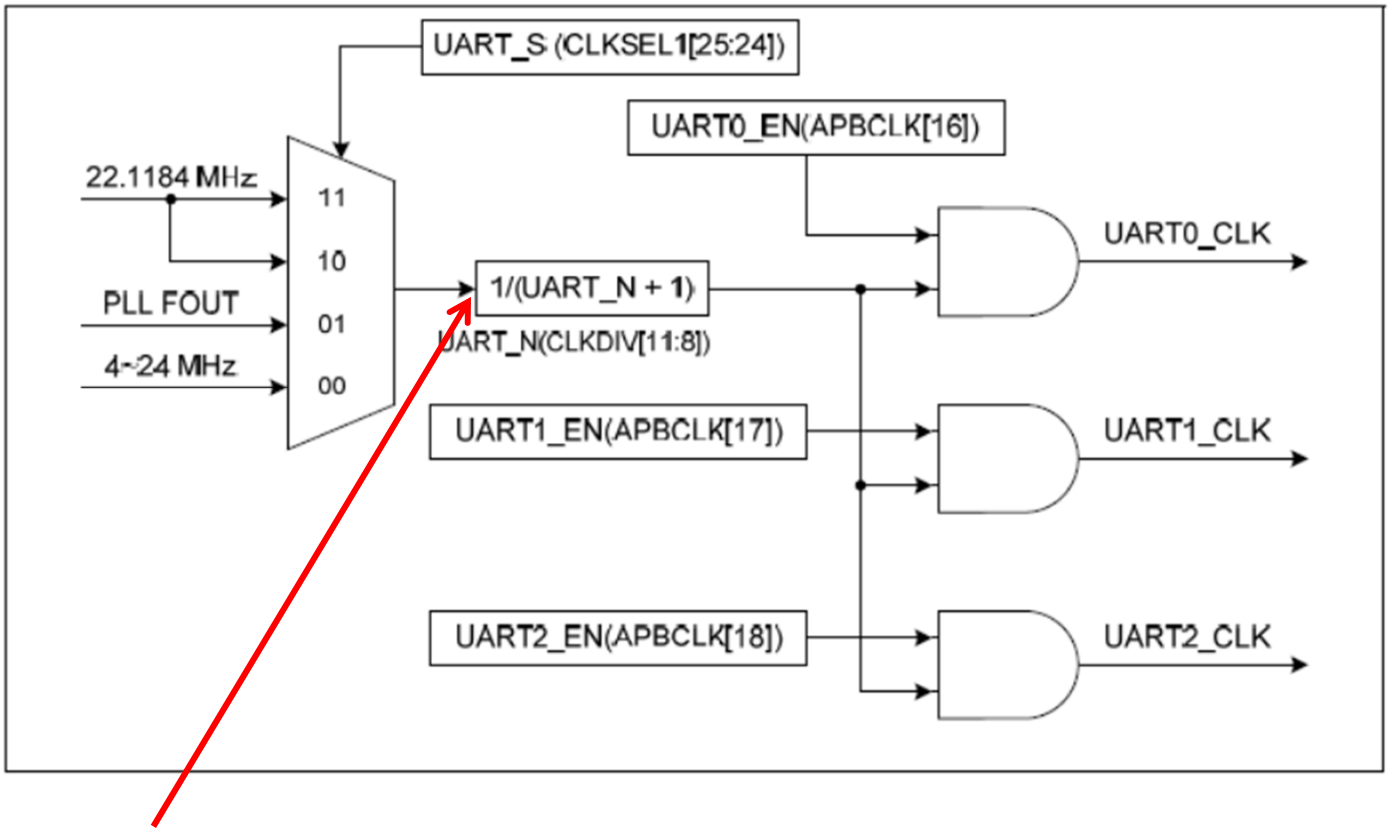
- Drei UARTs (UART0 – UART2)
  - Neben RS232 werden auch andere serielle Protokolle wie IrDA (Infrarot), LIN (Local Interconnect Network) und RS485 unterstützt
  - Nach Reset des Chips ist RS232 eingestellt
- Die UARTs können verschiedene Arten von Interrupts auslösen
- Die UARTs verfügen über Sende- und Empfangspuffer (FIFO: First-In First-Out)
  - UART0: 64 Byte
  - UART1, UART2: 16 Byte
- Unterstützung durch Treiberfunktionen in „Driver_M_Dongle.c“

Blockdiagramm UART



Quelle: Technical Reference Manual NUC130

NUC130 UART Clocks



Vorteiler für UART-Clocks (Clock Divider Register), = 0 nach Reset

Quelle: Technical Reference Manual NUC130

Auswahl des UART-Taktes

- Auswahl der Taktquelle für alle UARTs:
  - Z.B. externer 12 MHz Takt:
    ```
    SYSCLK->CLKSEL1.UART_S = 0;
    ```
- Vorteiler für alle UARTs im „Clock Divider Register“, nach Reset kein Vorteiler
- Einschalten des Taktes an UART (hier UART2):
  ```
  SYSCLK->APBCLK.UART2_EN = 1;
  ```

Pin-Zuordnung und Reset

- Wir benutzen den UART2, der auf dem Board am Sub-D-Stecker angeschlossen ist.
- Die Pins für TX und RX des UART2 sind gleichzeitig die GPIOD-Pins 14 und 15 („Multi-Function Pins“) und müssen dem UART2 zugeordnet werden (Register GPD_MFP):
  - `SYS->GPDMFP.UART2_RX = 1;`
  - `SYS->GPDMFP.UART2_TX = 1;`
- Der UART2 muss einen Reset erhalten, indem das Bit 18 im „Peripheral Reset Control Register 2“ (IPRSTC2) auf 1 und anschließen wieder auf 0 gesetzt wird.
  - `SYS->IPRSTC2.UART2_RST = 1;`
  - `SYS->IPRSTC2.UART2_RST = 0;`

Einstellen der Baudrate

- Die Baudrate BR berechnet sich nach der Gleichung
  BR = UART_CLK / (M * (BRD + 2))
- M ist ein programmierbarer Vorteiler, wenn UART2->BAUD.DIV_X_EN = 0 gesetzt wird, ist M = 16
- BRD ist der „Baud Rate Divider“
- Wenn M = 16 und UART_CLK = 12 MHz ist, gilt also für gewünschte Baudrate BR:
  BRD = 12 MHz / (16 * BR) – 2
- Beispiel: BR = 9600 Baud -> BRD = 76,125,
  da wir nur ganze Zahlen einstellen können ist BRD = 76
  (wie groß ist der Fehler?)

Wichtige Register für einfache UART-Ausgabe

- Datenregister „UART2->DATA“ (32 Bit):
  - Schreiben auf das Register bringt das LS-Byte des Registers in den Sende-FIFO
  - Lesen vom Register liest ein Byte aus dem Empfangs-FIFO (also nur LS-Byte relevant)
- „Line Control Register“ „UART2->u32LCR“:
  - Einstellen des Protokolls
  - u32LCR = 3: 8N1-Protokoll (weitere Möglichkeiten siehe Datenblatt Register UA_LCR)
- Baudraten-Register „UART2->BAUD“:
  - Einstellen der Baudrate

Treiberfunktionen

- **`void DrvUART2_Init(uint16_t u16Baudrate, uint8_t uiTrigLevelBytes)`**
  - Reset des UART2
  - Takt und Baudrate (BRD) einstellen
  - Pinzuordnung
  - Format 8N1 einstellen
  - Anzahl der Bytes im Empfangspuffer bis Interrupt generiert wird (**`uiTrigLevelBytes`**)

Treiberfunktionen (2)

- **`uint32_t DrvUART2_Write(uint8_t* pu8Data, uint32_t u32Bytes)`**
  - Sende eine Anzahl (**`u32Bytes`**) von Bytes über den UART2
  - Übergeben wird ein Zeiger (**`pu8Data`**) auf Bytes (z.B. String)
- Makros für Schreiben und Lesen:
  - **`M_UART2_DATA_WRITE(u8Data)`**
  - **`M_UART2_DATA_READ`**
  - Alternativ: Verwendung des Registers UART2->DATA

Beispiel: Lauflicht mit UART-Ausgabe

```
#include "BoardConfig.h"
#include "init.h"
#include <string.h>

int main (void){
   uint32_t i;
   char greet[] = "Hello World!\r\n";

   DrvSystem_ClkInit();     //Setup clk system
   Init_Board();            //Initialize peripherals
   DrvTimer0_Init();        //Initialize Timer 0
   DrvUART2_Init(9600, 1);  //Initialize UART

   i = strlen(greet);
   DrvUART2_Write((uint8_t *)greet, i);
   while(1) {
      while(UART2->FSR.TE_FLAG == 0){}  //Wait until TX empty
      for(i=LED0; i<LED7+1; i++){
         UART2->DATA = i + 0x30 - LED0; //Send character
         DrvGPIO_ClearBit(E_GPE, i);    //Clear Bit, LED i on
         DrvSystem_Delay(1000000);      //wait 1 sec
         UART2->DATA = '_';             //Send character
         DrvGPIO_SetBit(E_GPE, i);      //Set Bit, LED i off
         DrvSystem_Delay(1000000);      //wait 1 sec
      }
      UART2->DATA = 0x0D;  //Carriage Return
      UART2->DATA = 0x0A;  //Line Feed
   }
}
```

ASCII-Tabelle

ASCII-Zeichentabelle, hexadezimale Nummerierung

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Quelle: Wikipedia

Ausgabe des Programms

- PC an Stecker über Nullmodem-Kabel anschließen.
- Terminalprogramm auf PC starten (z.B. Tera Term)
- Protokoll auf 8N1 einstellen, Baudrate korrekt einstellen.
- Kein RS232 mehr am PC? USB-RS232-Adapter!

```
Hello World!
0_1_2_3_4_5_6_7_
0_1_2_3_4_5_6_7_
0_1_2_3_4_5_6_7_
0_1_2_3_4_5_6_7_
…
```

Pufferüberlauf

- Im Programm wird durch die Verzögerung in der Schleife jede Sekunde ein Zeichen/Byte gesendet.
- Zeitdauer für das Senden eines Bytes bei 9600 Baud:
  - Dauer 1 Bit = 1/9600 s = 104 µs
  - 1 Byte = 11 Bit = 1,14 ms
  - Die Zeitabstände zwischen zwei Zeichen dürfen nicht kürzer als diese Zeit sein, da sonst der Sende-Puffer überläuft und die Zeichen nicht mehr korrekt gesendet werden
- Daher Abfrage des „Transmitter FIFO Full Flag“ aus dem „FIFO Status Register“ (FSR, Abfrage in Treiberfunktion vorhanden):

```
while(UART2->FSR.TX_FULL == 1){}
```

Kapitelübersicht

I. Taktsystem und System-Timer

II. UART

III. SPI und LCD

IV. A/D-Wandler

SPI und LCD

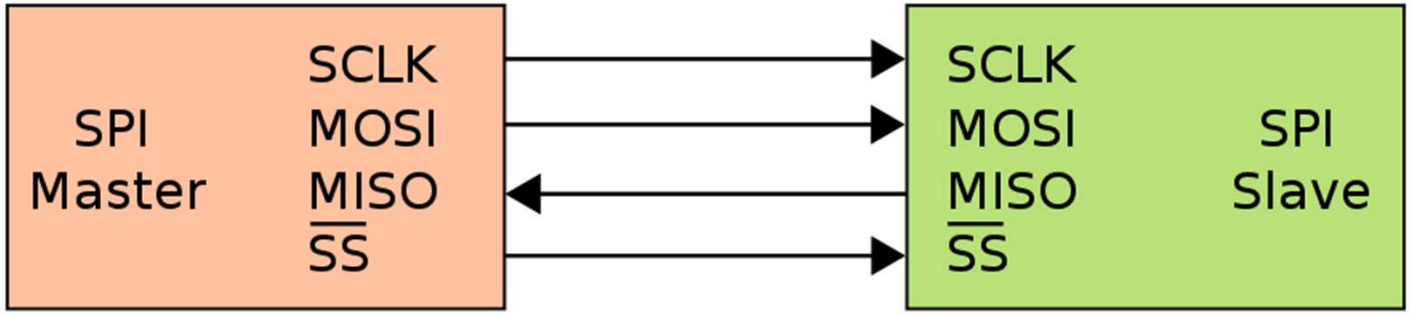
- Das Laborboard verfügt über ein LC-Display (LC: Liquid Crystal, dt.: Flüssigkristallanzeige)
- Die Daten für das LCD werden vom NUC 130 über SPI (Serial Peripheral Interface) übertragen.
- Wir besprechen zunächst das SPI und dann die Ansteuerung des LCDs über SPI.

Serial Peripheral Interface

- SPI ist ein serielles, synchrones Bussystem, mit welchem Komponenten eines Mikrocontrollersystems verbunden werden können.
  - A/D-Wandler
  - EEPROM und Flash-Speicher
  - Sensoren
  - LCD
  - …
- Wurde ursprünglich von Motorola entwickelt.
- Daten können vollduplex übertragen werden.
- Eine Komponente ist der so genannte „Master“, an den ein oder mehrere Komponenten als „Slave“ angeschlossen werden können.

Datenübertragung im SPI

- Für die Übertragung der Daten werden vier Leitungen zwischen Master und Slaves benutzt:
  - SCK/SCLK: Taktleitung, wird vom Master getrieben
  - MISO: Master-In, Slave-Out, Daten fließen vom Slave zum Master
  - MOSI: Master-Out, Slave-In, Daten fließen vom Master zum Slave
  - SS_n: Auswahl des Slaves, Slave nur aktiv (senden/empfangen), wenn diese Leitung aktiviert wird (low-aktiv).

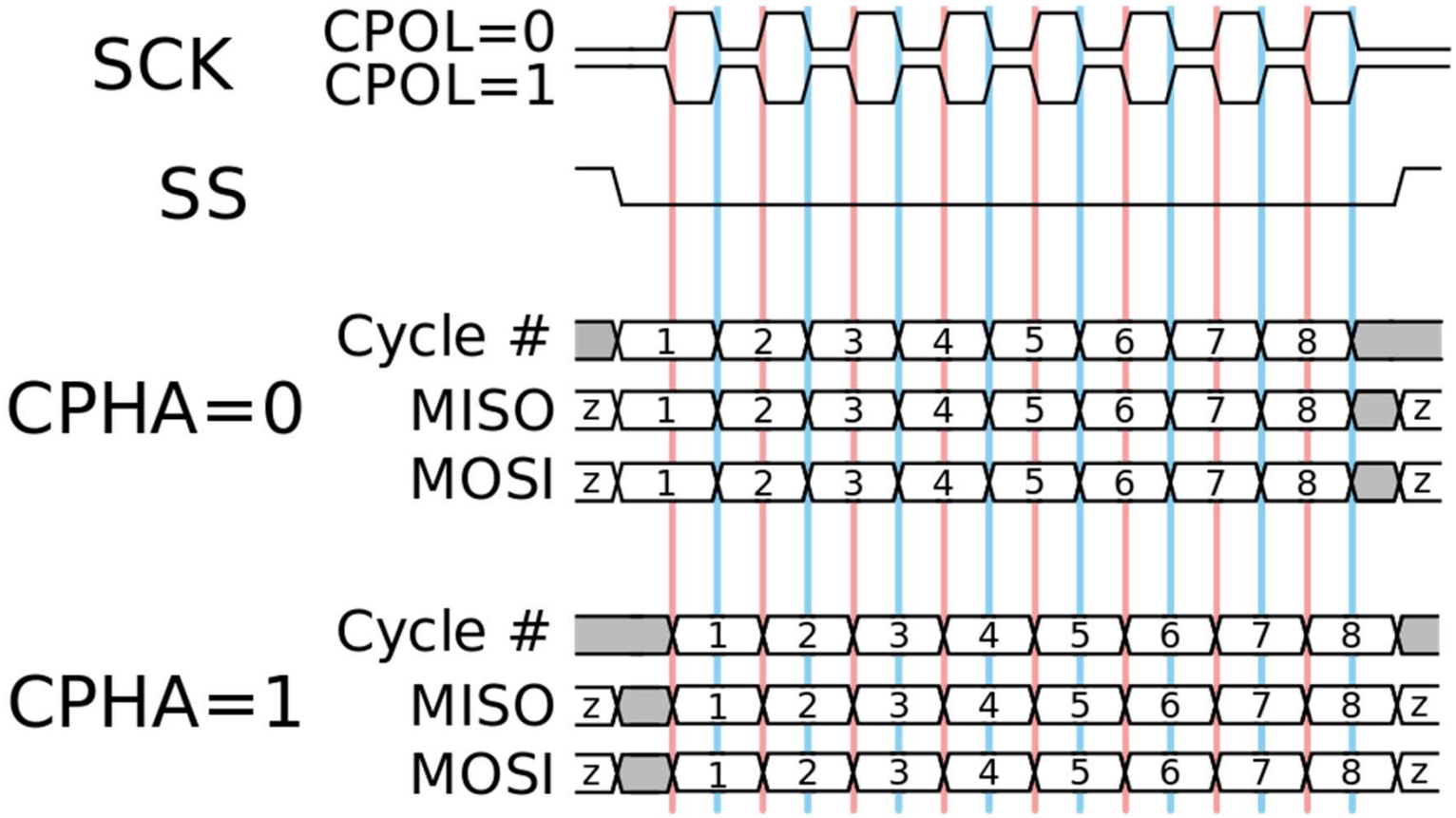


Quelle: Wikipedia

SPI Protokoll (1)

- Das Protokoll wurde von Motorola nicht exakt festgelegt und es sind verschiedene Modi möglich, die dann nicht kompatibel sind.
- CPOL = Clock Polarität im inaktiven Zustand:
  - im inaktiven Zustand (engl.: idle) werden keine Daten übertragen
  - 0: Clock inaktiver Zustand „low“
  - 1: Clock inaktiver Zustand „high“
- CPHA = Clock Phase
  - 0: Daten werden mit der ersten Taktflanke übernommen, nachdem SS = 0 (aktiv) wurde
  - 1: Daten werden mit der zweiten Taktflanke übernommen, nachdem SS = 0 (aktiv) wurde
- Die Übertragung von Daten wird also vom Master durch SCK und SS gesteuert, üblicherweise werden die Daten byteweise übertragen, wobei auch mehrere Bytes nacheinander übertragen werden können.

SPI Protokoll (2)



Quelle: Wikipedia

SPI Modi

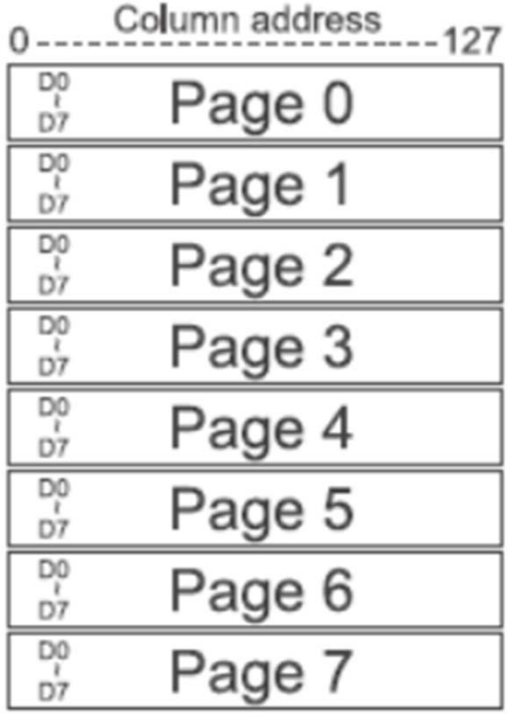
Mode	CPOL	CPHA	Datenübernahme Master und Slave
0	0	0	Mit steigender Flanke
1	0	1	Mit fallender Flanke
2	1	0	Mit fallender Flanke
3	1	1	Mit steigender Flanke

Das LC-Display

- Grafik-LCD mit 128 x 64 Pixel
  - „Transflektiv“ mit einschaltbarem „Backlight“ (RGB)
- Pixel sind monochrom, 1 Bit pro Pixel notwendig
  - 0: Pixel aus, 1: Pixel an (schwarz)
- LCD-Controller verfügt über Bildspeicher mit 128 x 64 Bit
  - Jedes Bit steuert ein Pixel an
- Ausgabe von Text möglich: 21 Zeichen pro Zeile, 8 Zeilen
  - Unterstützung der Textausgabe durch Aufteilung des LCD in „Pages“, eine Page = 8 Pixel-Zeilen
- Ausgabe von Grafiken möglich
- Wir besprechen die Ausgabe von Zeichen

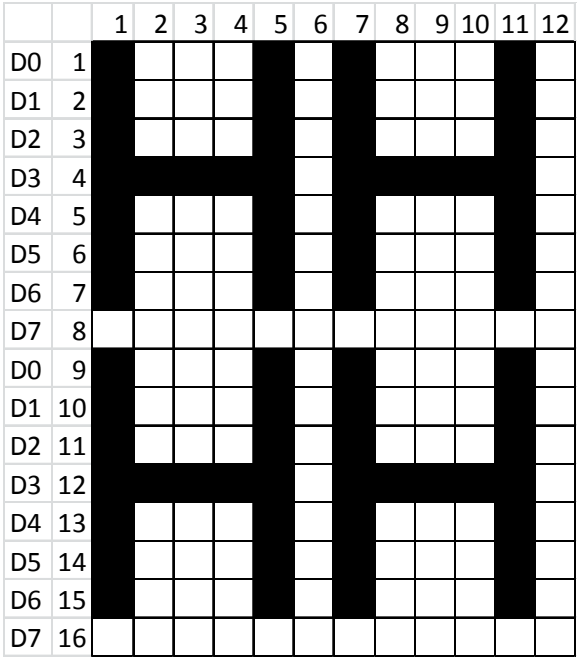
Page-Aufteilung des LCDs

- Per Kommando (SPI) wird eine Page angewählt.
- Per Kommando wird eine Spalte in der Page ausgewählt (D0-D7)
- Nun können ab dieser Spalte Bytes über SPI an den Bildspeicher des LCD-Controllers gesendet werden. Jedes gesendete Byte wird an den folgenden Spalten gespeichert und definiert die Pixel dieser Spalte.



Darstellung von ASCII-Zeichen

- Zeichen werden durch entsprechende Belegung der Pixel erzeugt.
- Jedes Zeichen belegt 5 x 7 Pixel
- Die sechste Spalte und die achte Zeile werden freigelassen, also mit 0 beschrieben.
- Somit wird für jedes Zeichen 5 x 8 Bit benötigt.
- Pro Page sind somit 21 Zeichen darstellbar.

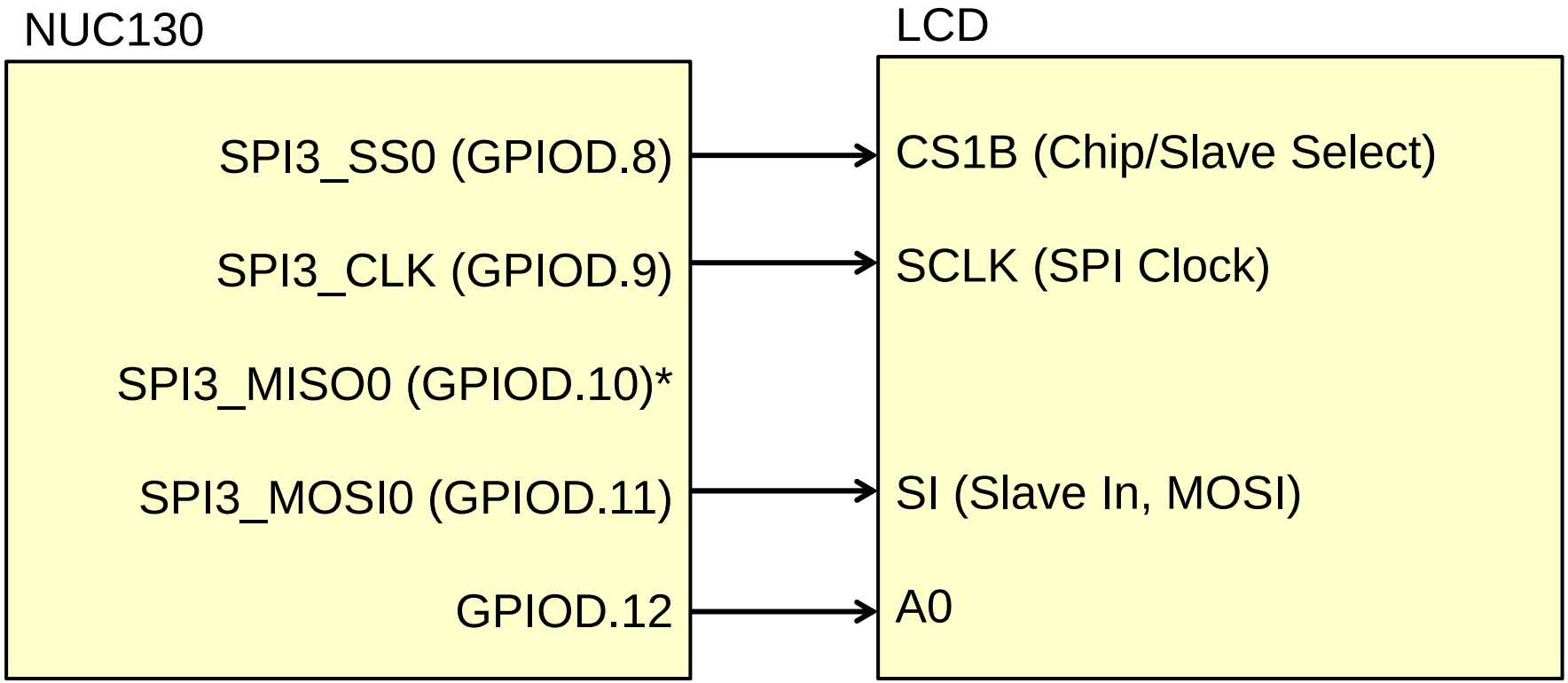



```
const uint8_t font5x7[]={   // ASCII
  0x00,0x00,0x00,0x00,0x00, // 0x00
  …
  0x7F,0x08,0x08,0x08,0x7F, // 0x48:'H‘
  …
}
```

Erzeugen der Zeichendaten

- Einfachste Möglichkeit: Die notwendigen 5 Byte für jedes ASCII-Zeichen werden in einer Tabelle abgelegt und die Startadresse des Zeichens (1. Byte) wird über den ASCII-Wert bestimmt.
  - Startadresse = ASCII-Wert x 5
- Es wird also ein Feld mit 128 x 5 Byte benötigt.
  - Optimierung: Nur die druckbaren Zeichen werden gespeichert.
- Beispiel: ‚H‘ = 0x48 = 72
  - Startadresse ist: A = 72 x 5 = 360
  - Bytes: 0x7F, 0x08, 0x08, 0x08, 0x7F

Verschaltung von NUC130 und LCD



*: Nicht verwendet, nur Simplex-Kommunikation vom Master zum Slave möglich.

Übertragen von Daten per SPI

- Man schreibt ein Byte auf den TX-Puffer des SPI (hier SPI3)
- Durch Setzen des GO-BUSY-Flags werden die Daten übertragen.
- Sind die Daten übertragen, so wird das GO-BUSY-Flag zurückgesetzt. Es muss daher zuvor abgefragt werden, ob der TX-Puffer frei ist.

```
void SPI3_SingleWrite_Data(uint8_t pu8Data)
{
    while(SPI3->CNTRL.GO_BUSY == 1)
    {}

    SPI3->TX[0] = pu8Data;
    SPI3->CNTRL.GO_BUSY = 1;
}
```

Bestandteil von GLCD.c (Labor)

Kommandos und Daten im LCD

- Die über SPI übermittelten Bytes werden vom LCD als Kommando interpretiert, wenn A0 = 0 ist (Makro M_LCD_SET_COMMAND).
- Ist A0 = 1, so werden die Bytes als Daten in den Bildspeicher geschrieben (Makro M_LCD_SET_DATA), ab der gesetzten Position (page/column).
- Das LCD muss zu Beginn durch entsprechende Kommandos initialisiert werden (nicht gezeigt).
- Die Position der Zeichenausgabe muss immer gesetzt werden („Cursor“).

TABLE OF PROGRAMMING COMMANDS

Command	Command Code									Function
	A0	D7	D6	D5	D4	D3	D2	D1	D0	
(1) Display ON/OFF	0	1	0	1	0	1	1	1	0 1	LCD display ON/OFF 0: OFF, 1: ON
(2) Display start line set	0	0	1	Display start address						Sets the display RAM display start line address
(3) Page address set	0	1	0	1	1	Page address				Sets the display RAM page address
(4) Column address set upper bit	0	0	0	0	1	Most significant column address				Sets the most significant 4 bits of the display RAM column address.
Column address set lower bit		0	0	0	0	Least significant column address				Sets the least significant 4 bits of the display RAM column address.
(6) Display data write	1	Write data								Writes to the display RAM

Setzen des „Cursors“

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	D	i	e	s		i	s	t		e	i	n		D	i	s	p	l	a	y	!
1		T	e	s	t	T	e	x	t												
2			T	e	s	t	T	e	x	t											
3				T	e	s	t	T	e	x	t										
4					T	e	s	t	T	e	x	t									
5						T	e	s	t	T	e	x	t								
6							T	e	s	t	T	e	x	t							
7								T	e	s	t	T	e	x	t						

GLCD_SetRow(x): Setzen der Zeile (Page)
GLCD_SetColumn(y): Setzen auf die Spalte
Da jedes Zeichen 6 Spalten benötigt, müssen aufeinander folgende Zeichen einen Abstand von 6 Spalten haben.

Setzen der Zeilenposition des „Cursors“

```
void GLCD_SetRow(uint8_t ui8Row)
{
        M_LCD_SET_COMMAND;

        ui8Row &= 0x0F;

        SPI3_SingleWrite_Data(0xB0 + ui8Row);
}
```

M_LCD_SET_COMMAND: Makro setzt A0 = 0

Argument für SPI3_SingleWrite_Data:
- ui8Row: unteres Nibble extrahieren (ui8Row &= 0x0F), Zeilenposition
- Oberes Nibble auf 0xB setzen (siehe „Command Code“ in Tabelle)

Hinweis: ui8Row darf nur Werte zwischen 0 und 7 annehmen, da nur 8 Pages vorhanden!

Setzen der Spaltenposition des „Cursors“

```
void GLCD_SetColumn(uint8_t ui8Column)
{
        M_LCD_SET_COMMAND;

        SPI3_SingleWrite_Data((ui8Column / 16) + 16); //Oberes Nibble

        SPI3_SingleWrite_Data(ui8Column % 16);        //Unteres Nibble
}
```

Spaltenposition muss in zwei Nibbles übertragen werden (Achtung: nur 128 Spalten vorhanden, d.h. ui8Column < 128):

Oberes Nibble H: Kommando 0x1H
- ui8Column/16: Schieben um 4 Positionen nach rechts, Extraktion des oberen Nibbles
- +16: Kommando-Code 0001

Unteres Nibble L: Kommando 0x0L
- ui8Column % 16: Extraktion des unteren Nibbles, damit Kommando-Code automatisch 0000

Zeichenausgabe

```
void GLCD_PrintChar(uint8_t ui8Char)
{
    uint32_t i;

    M_LCD_SET_DATA;                                 //  Set LCD to data mode

    for(i = 0; i < 5; i++)
    {
         SPI3_SingleWrite_Data(font5x7[i+ui8Char*5]);
    }
    SPI3_SingleWrite_Data(0);

}
```

Hinweis: Beim Schreiben auf den Bildspeicher des Displays (genauer gesagt des Display-Controllers) wird die Spaltenposition im Bildspeicher automatisch inkrementiert.
Nur die Spaltenpositionen 0-127 entsprechen Bildpunkten auf dem Display. Ab Spaltenposition 128 wird also nichts mehr angezeigt und ab Position 131 hört der Display-Controller auf zu inkrementieren. Ein „Überlauf“ in die nächste Page ist nicht möglich.

Textausgabe mit Setzen des Cursors

```
void GLCD_PrintText(uint8_t ui8Row, uint8_t ui8Column, uint8_t *aui8Text)
{
    uint8_t i;

    GLCD_SetRow(ui8Row);
    GLCD_SetColumn(ui8Column*6);

    for(i = 0; aui8Text[i] != 0; i++)
    {
        GLCD_PrintChar(aui8Text[i]);
    }

}
```

Achtung: Eine Zeile kann nur 21 Zeichen darstellen! Ist der übergebene String länger, so werden die restlichen Zeichen nicht dargestellt. Entsprechendes gilt, wenn die Spalten-position > 0 ist.

Beispiel-Programm

```
#include "BoardConfig.h"
#include "Driver_M_Dongle.h"
#include "init.h"
#include "GLCD.h"
#include <string.h>

int main (void){
   uint32_t i;
   uint8_t text[] = "TestText";

   DrvSystem_ClkInit();     //Setup clk system
   Board_Init();            //Initialize peripherals
   DrvTimer0_Init();        //Initialize Timer 0
   GLCD_Init();             //Initialize LCD

   GLCD_PrintText(0,0,"Dies ist ein Display!");

   while(1) {
      for(i=1; i<8; i++){
         GLCD_PrintText(i,i,text);
         DrvSystem_Delay(1000000);  //wait 1 sec
         GLCD_ClearRow(i);
      }
   }
}
```

Kapitelübersicht

I. Taktsystem und System-Timer

II. UART

III. SPI und LCD

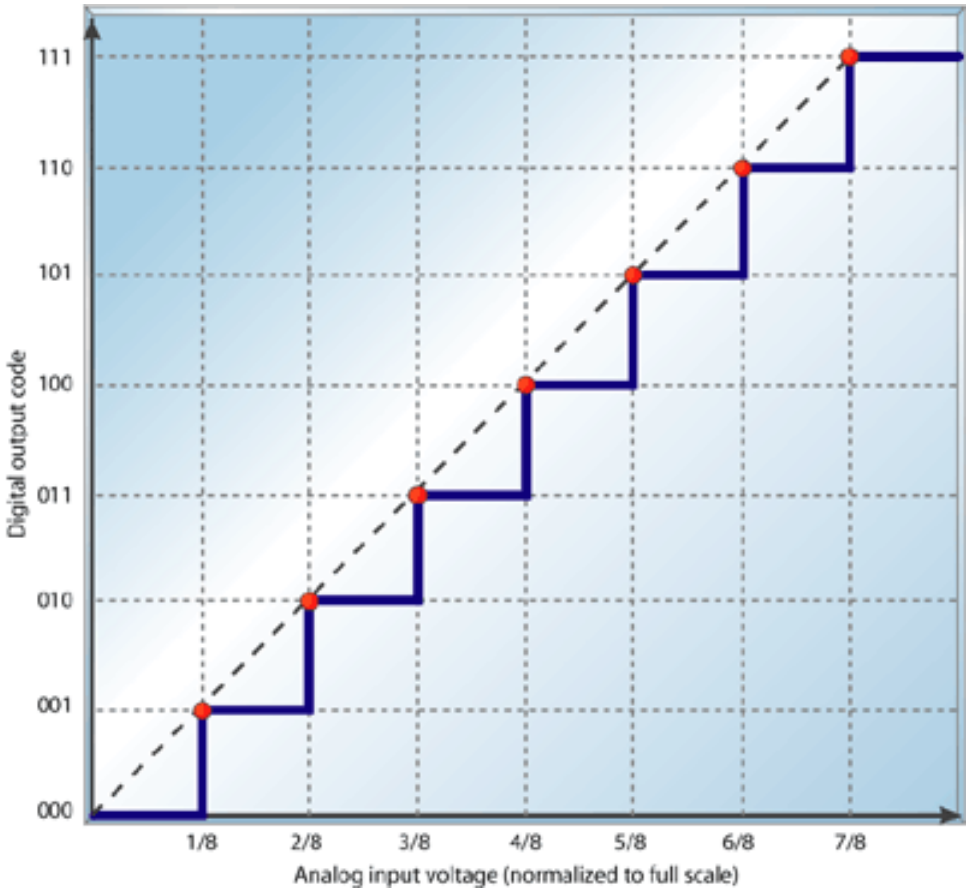
IV. A/D-Wandler

A/D-Wandler

- Ein Analog-Digital-Wandler (engl. Analog-to-Digital-Converter, ADC) wandelt eine analoge Spannung in einen digitalen Code (Dualzahl). Dabei wird gegen eine Referenzspannung Uref verglichen.
- Einem digitalen Wert ist ein Wertebereich des analogen Signals zugeordnet.
- Die Größe dieses Wertebereichs (Stufe) hängt von der **Auflösung** des ADC ab (in Bit) und wird häufig als 1 LSB bezeichnet. Kleinere Spannungsunterschiede werden vom ADC nicht erkannt.
- $1\ \text{LSB} = \text{Uref} / 2^{\text{Auflösung}}$

ADC Transferfunktion

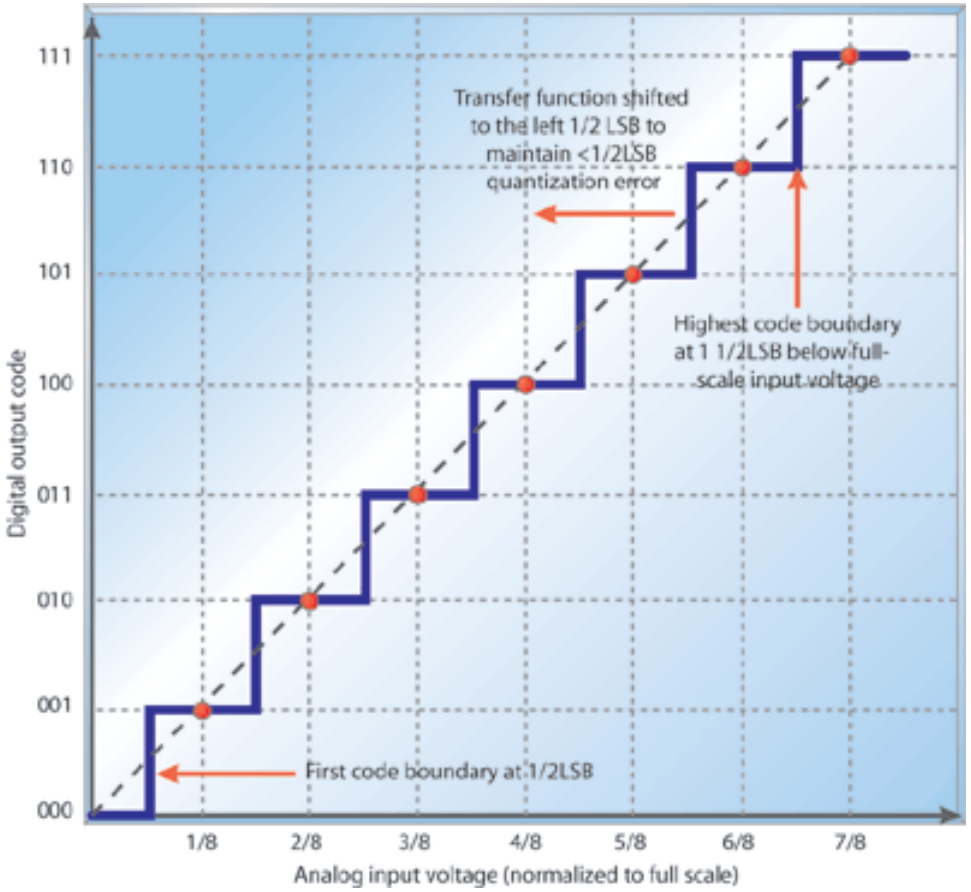
- Beispiel: 3 Bit Auflösung
- Uref = 4 V
- 1 LSB = 0,5 V
- Code 111 entspricht Uref – 1 LSB
  = 7/8 * 4 V = 3,5 V



Quelle: www.embedded.com

ADC Transferfunktion mit Offset

- Verschiebung um ½ LSB
- Bereich für Code 000 ist ½ LSB
- Bereich für Code 111 ist 1,5 LSB

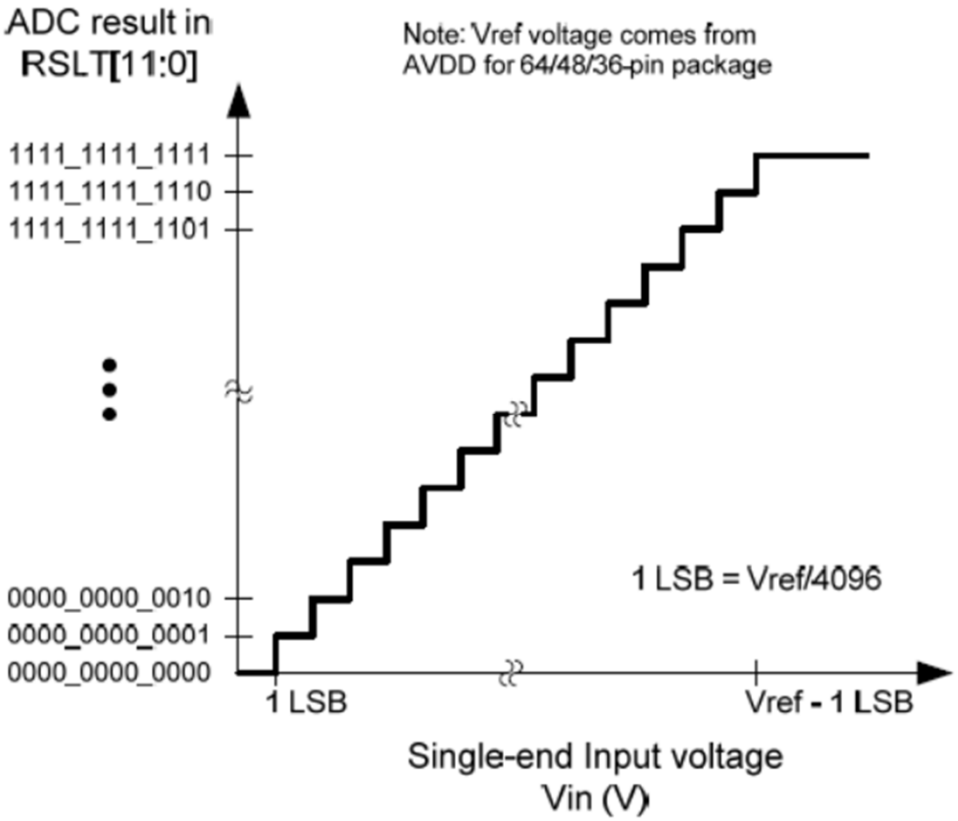


Quelle: www.embedded.com

Welche Spannung entspricht einem Codewort?

- Problem: Unsicherheit von 1 LSB
- Sinnvoll: Codewort 000 entspricht 0 V
- Somit: Codewort 111 entspricht Uref – 1 LSB
- Alle anderen Codeworte werden linear zugeordnet, indem man das Codewort als Integerwert auffasst:
  Spannung = Codewort * 1 LSB
- Im Beispiel:
  Spannungswert = Codewort * 4 V / 8

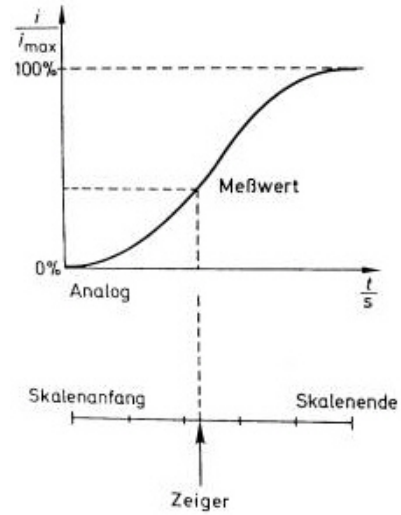
Transferfunktion des ADC im NUC130

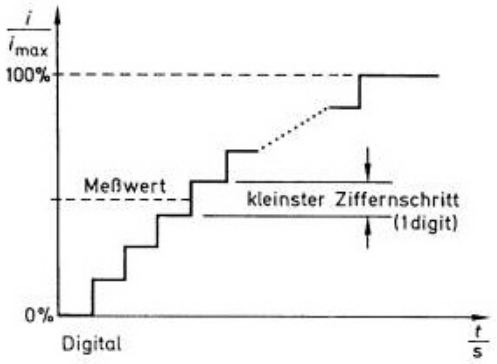


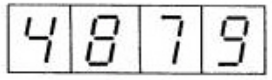
Auflösung: 12 Bit
Uref: 3,3 V
1 LSB = 0,8 mV

Zeitdiskretisierung

- Die Wandlung findet nicht kontinuierlich statt, sondern der ADC entnimmt dem analogen, zeitkontinuierlichen Signal in bestimmten Zeitabständen einen Messwert und wandelt diesen.
- Aus einem **wert- und zeitkontinuierlichen** (analogen) Signal wird also ein **wert- und zeitdiskretes** (digitales) Signal.
- Der kleinste mögliche Zeitabstand t_{min} zwischen zwei Wandlungen ergibt die maximale Abtastfrequenz des ADC $f_{A,max}$. Für das zu wandelnde Signal muss gelten: $f_{signal,max} < ½ \cdot f_{A,max}$. Dies bedeutet, dass Signale, die sich schneller ändern, nicht mehr korrekt erfasst werden können.





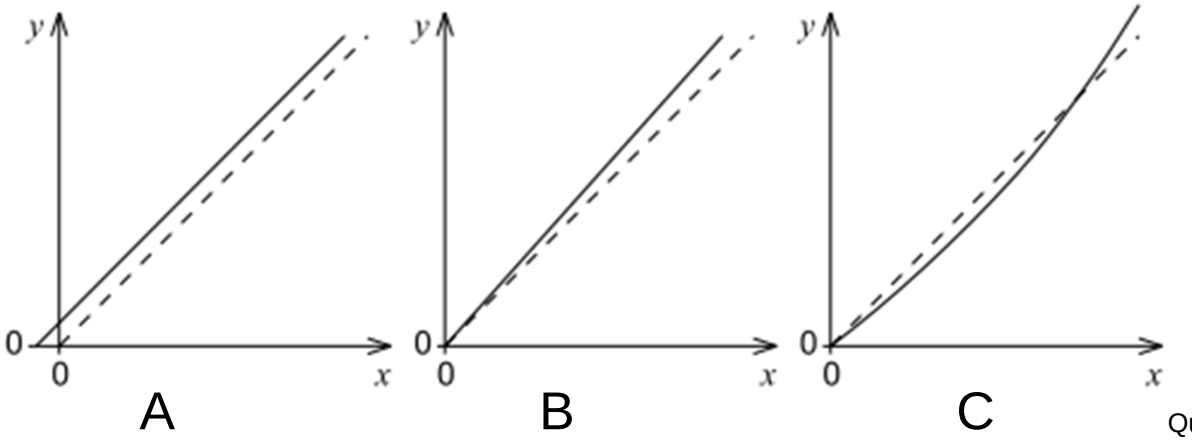


Geschwindigkeit und Genauigkeit

- Es sind eine Vielzahl von ADC-Funktionsprinzipien bekannt, die sich hauptsächlich in Geschwindigkeit ($f_{A,max}$) und Genauigkeit sowie den Kosten unterscheiden, z.B.
  - Flash-Wandler (Direkte Umsetzung)
  - Delta-Sigma-Wandler (Überabtastung)
  - Sukzessive Approximation (Iterative Umsetzung)
- Die Auflösung (1 LSB) stellt die Genauigkeitsgrenze für die Wandlung dar („Quantisierungsfehler“).
  - Änderungen, die kleiner als 1 LSB sind, können vom ADC nicht erfasst werden.
- Die nutzbare Genauigkeit kann durch Kennlinienfehler des ADC und andere Einflüsse verschlechtert werden.

A/D-Wandler: Kennlinienfehler

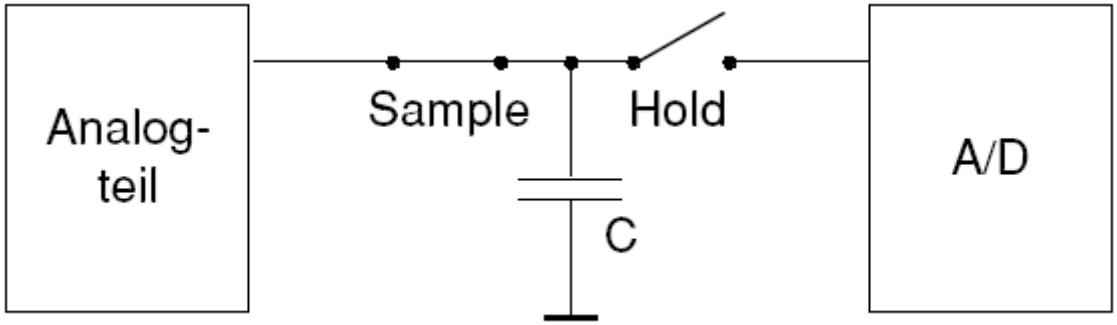
- A/D-Wandler können folgende Fehler in der Kennlinie aufweisen:
  - A) : Nullpunkt-Fehler (offset)
  - B) : Verstärkungsfehler (gain)
  - C) : Nichtlinearitäten
- ADCs werden i.d.R. abgeglichen („kalibriert“), um diese Fehler zu korrigieren.



Quelle: Wikipedia

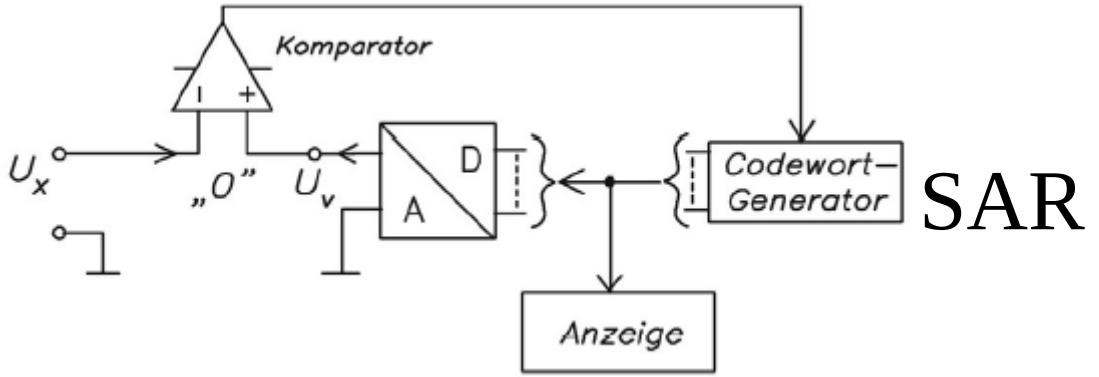
Abtast-Halte-Schaltung

- Die Abtast-Halte-Schaltung (Sample&Hold, Track&Hold) entnimmt dem Eingangssignal einen Messwert (Sample) und hält diesen während der Wandlung (auf einem Kondensator) fest (Hold).



Sukzessive Approximation

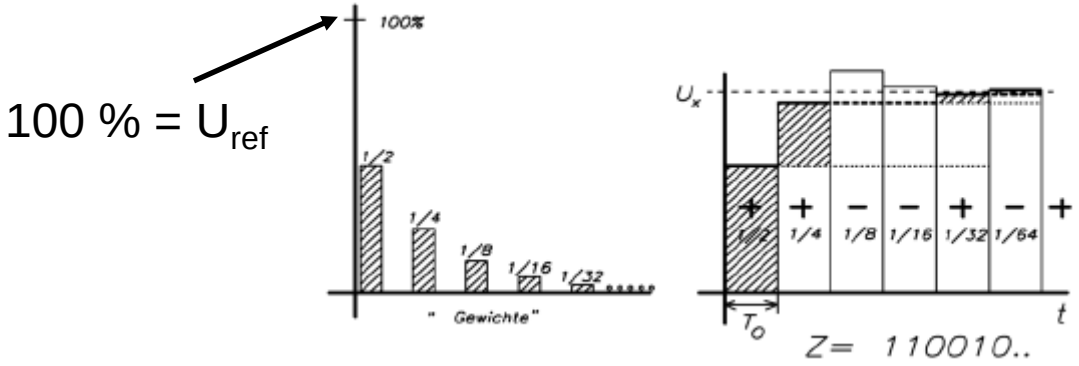
- Der (analoge) Komparator vergleicht die Eingangsspannung U_x mit der von einem D/A-Wandler erzeugten Spannung U_v.
- Der Komparator steuert den digitalen Codewort-Generator, welcher wiederum die Daten für den D/A-Wandler (DAC) liefert.
- Die Spannung U_v des DAC liegt zwischen 0 V und U_{ref}. Über U_{ref} kann daher der aufzulösende Bereich eingestellt werden.
- Der Codewort-Generator wird auch als SAR (Sukzessive-Approximations-Register) bezeichnet. Im SAR steht nach abgeschlossener Wandlung das Ergebnis.
- Die Grundidee des Verfahrens besteht darin, durch sukzessive Veränderung des digitalen Codeworts mittels DAC eine Spannung U_v zu erzeugen, die genauso groß ist wie U_x, so dass die Differenz beider Spannungen Null ist.



SAR

Sukzessive Approximation (2)

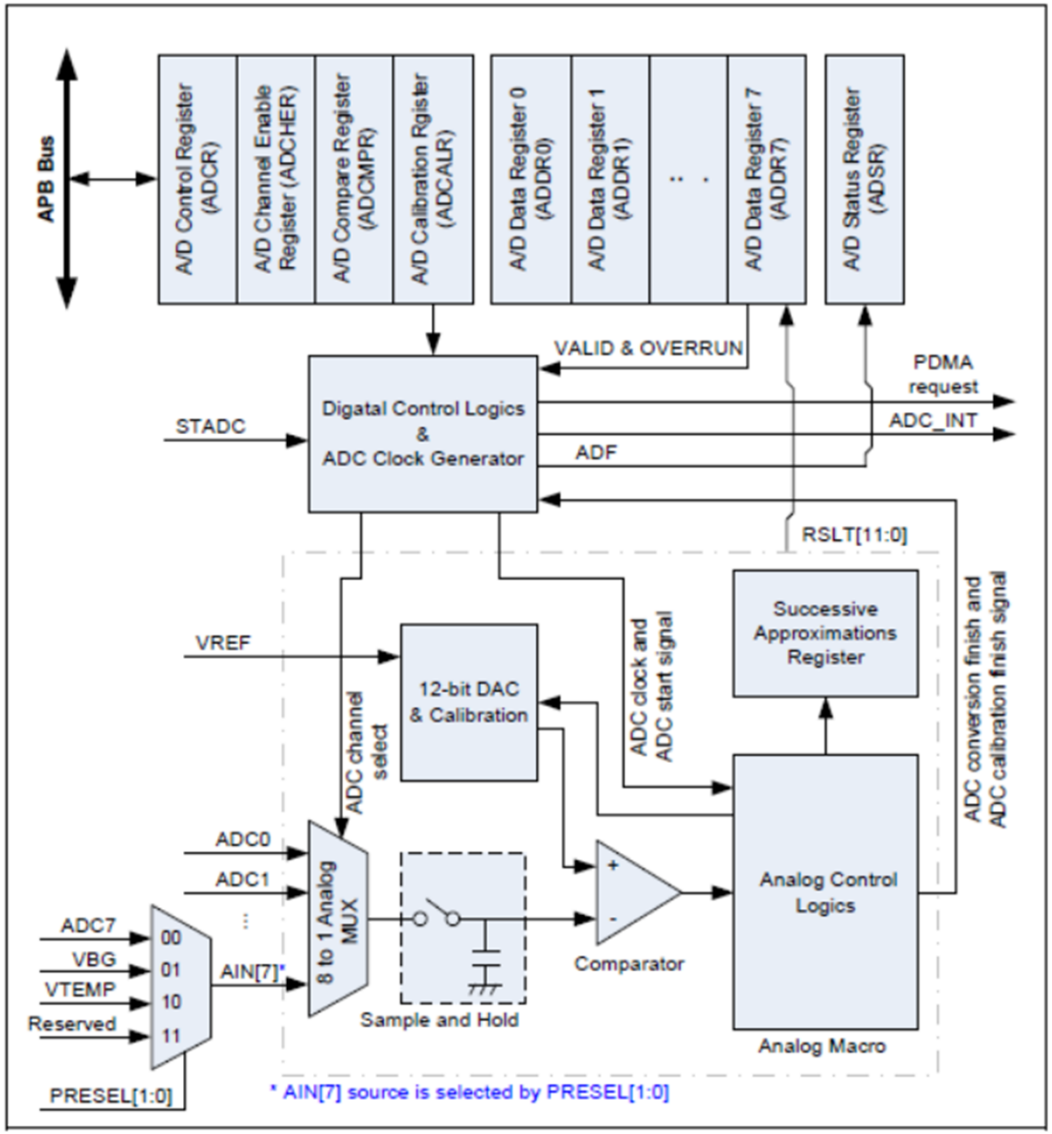
- Es handelt sich um ein „Wägeverfahren“ (hier für 6 Bit):
  1. Zu Beginn ist SAR = 000000
  2. Nun wird das MSB gesetzt, d.h. SAR=100000 und damit $U_v = 1/2 \cdot U_{ref}$
  3. Ist $U_x > U_v$ (Komparator), so bleibt das Bit (= „Gewicht“) gesetzt, anderenfalls wird es wieder auf 0 gesetzt.
  4. Nun wird MSB-1 gesetzt, d.h. SAR=110000 und damit $U_v = 3/4 \cdot U_{ref}$
  5. Ist $U_x > U_v$ (Komparator), so bleibt das Bit gesetzt, anderenfalls wird es wieder auf 0 gesetzt.
  6. Weiter mit MSB-2, d.h. iterative Wiederholung bis das LSB erreicht ist.
- Im Beispiel wäre die Eingangsspannung zu $25/32 \cdot U_{ref}$ bestimmt worden. Bei $U_{ref} = 3{,}3$ V also zu $U_x \approx 2{,}6$ V.



Der ADC im NUC 130

- 12-Bit SAR-Wandler
  - 12 Bit Auflösung
  - 10 Bit Genauigkeit (8 Bit realistisch)
- 8 Kanäle
- 3 Betriebsmodi:
  - „Single Mode“: Eine Wandlung an einem Kanal
  - „Scan Mode“: Eine Wandlung über alle Kanäle
  - „Continuous Scan Mode“: Kontinuierlicher Scan Mode

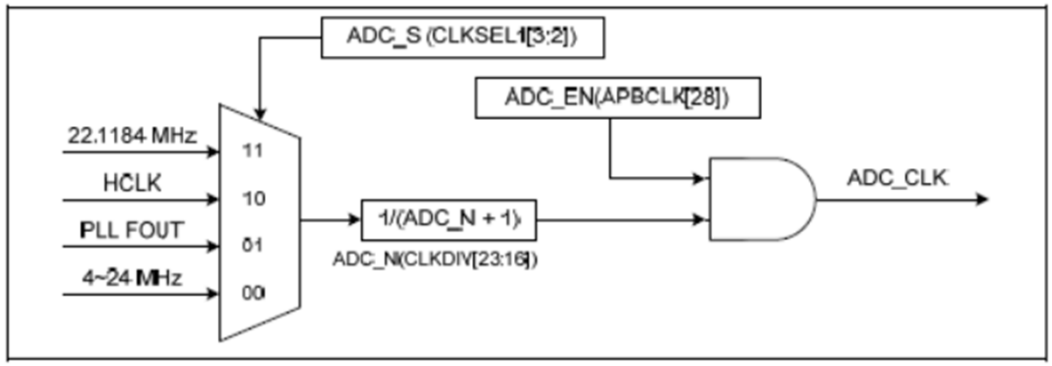
ADC Blockdiagramm



Quelle: Technical Reference Manual NUC130

ADC Taktversorgung

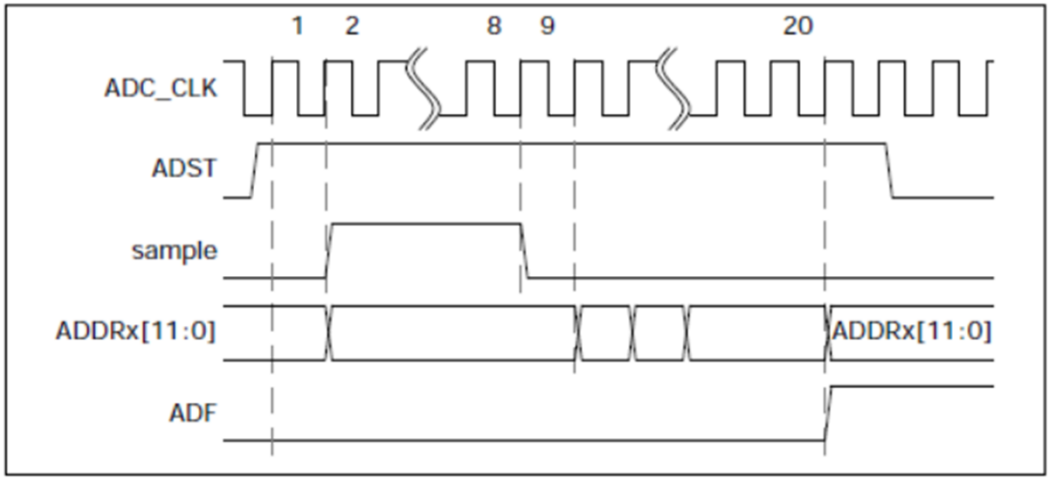
- Taktfrequenz darf 16 MHz nicht überschreiten.
- Taktquelle und Vorteiler kann ausgewählt werden.



Quelle: Technical Reference Manual NUC130

Wandlung im Single-Mode

1. Eine Wandlung wird gestartet, indem das ADST-Bit im Register ADCR (A/D Control Register) gesetzt wird.
2. Nach Ablauf der Wandlung ist das Ergebnis im „A/D Data Register“
3. Das ADF-Bit im ADSR (Status Register) wird gesetzt und das ADST-Bit wird rückgesetzt.



Quelle: Technical Reference Manual NUC130

Wie lange dauert eine Wandlung?

- Annahme: Wir takten den ADC mit 12 MHz
- Eine Wandlung benötigt 21 Takte
- Zeitdauer: T = 21 * 1/12 MHz = 1,75 µs
- Maximale Abtastfrequenz $f_{A,max}$ = 571,4 kHz
- Maximale Signalfrequenz $f_{S,max}$ = 285,7 kHz

Ausführen einer AD-Wandlung

```
uint8_t ADC_Convert(void) {
  uint32_t result;

  M_ADC_CONVERT_START;                  // Start a conversion
  while(M_ADC_CONVERT_DONE == 0);       // Wait until conversion is done
  M_ADC_INT_CLR_ADF;                    // Reset ADF-Flag by writing a 1
  result = M_ADC_DATA_READ(CHANNEL_2);  // Get data
  result = result>>4;                   // Skip 4 LSBs
  return (uint8_t)result;
}
```

M_ADC_CONVERT_START: Makro, ADC->ADCR.ADST = 1
M_ADC_CONVERT_DONE: Makro, ADC->ADSR.ADF
M_ADC_DATA_READ(): Makro, Holt 12-Bit Ergebnis vom ADC
M_ADC_CLR_ADF: Setze ADF-Flag zurück

Wir schneiden die unteren 4 Bit des 12 Bit Ergebnisses ab (durch Rechtsschieben) und erhalten somit ein 8 Bit Ergebnis.

Hauptprogramm

```
#include "BoardConfig.h"
#include "Driver_M_Dongle.h"
#include "init.h"
#include "GLCD.h"

#include "ADC.h"
#include "Display.h"
#include <string.h>

int main (void){
   uint8_t adResult;        //AD result
   uint8_t resultString[3]; //Result string

   DrvSystem_ClkInit();     //Setup clk system
   Board_Init();            //Initialize peripherals
   DrvTimer0_Init();        //Initialize Timer 0
   GLCD_Init();             //Initialize LCD
   DrvADC_Init(CHANNEL_2_SELECT); //Initialize ADC

   GLCD_PrintText(0,0,"A/D Wandler");
   GLCD_PrintText(1,0,"-----------");
   GLCD_PrintText(3,0,"A/D Ergebnis: ");

   while(1) {
      adResult = ADC_Convert(); //Convert and get result
      disp8Hex(adResult, resultString); //Result as ASCII-string
      GLCD_PrintText(3,16,resultString); //Print result to LCD
   }
}
```

Umwandlung von Zahl in ASCII-Zeichen

```
void disp8Hex(uint8_t value, uint8_t *stringPtr){
   uint8_t i;

   i = value>>4;                 //Get upper nibble
   stringPtr[0] = int2ascii(i);  //Convert to ASCII
   i = value & 0x0F;             //Get lower nibble
   stringPtr[1] = int2ascii(i);  //Convert to ASCII
   stringPtr[2] = 0;             //Zero termination
}

char int2ascii(uint8_t digit) {
   if(digit < 10) {
      return digit + 0x30; //ASCII character 0-9
   }
   else {
      return digit - 10 + 0x41; //ASCII A-F
   }
}
```

Ausgabe des Programms

0	**A**	**/**	**D**		**W**	**a**	**n**	**d**	**l**	**e**	**r**									
1	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**									
2																				
3	**A**	**/**	**D**		**E**	**r**	**g**	**e**	**b**	**n**	**i**	**s**	**:**				**C**	**0**		
4																				
5																				
6																				
7																				

Umrechnung der ADC-Codes in Spannung

- Gegeben sei ein zu messendes Signal, z.B. Spannung U_m am Potentiometer zwischen 0 und Uref = 3,3 Volt.
- 8 Bit ADC-Auflösung, daher 1 LSB = 0,012891 V = 12,9 mV
- Fullscale-Code 255 (FF) entspricht 3,3 V – 1 LSB = 3,287 V.
  Umrechnung der Codes in Spannung:
  Spannungswert = Code * 0,012891 V
- Da Auflösung 12 mV, ist Darstellung von 3 Nachkommastellen für den Spannungswert ausreichend
- Um eine Rechnung mit Gleitpunktzahlen zu vermeiden, verschieben wir die Nachkommastellen des LSBs durch Multiplikation mit 10^6 um 6 Stellen nach links und rechnen:
  Spannungswert = Code * 12891
- Wir erzeugen die einzelnen Ziffern des dezimalen Spannungswerts indem wir den Code fortgesetzt durch 10^6 wieder dividieren (nächste Seite).

Umrechnungsalgorithmus

1. X = ADC-Code * 12891
2. Von i = 0 bis i = 4
   i = 0: Vorkommastelle, i = 1: 1. Nachkomma-stelle, bis zur 3. Nachkommastelle
3. Ziffer = X / 10^6
4. X = (X % 10^6) * 10
5. Gehe zu 2. für nächste Ziffer

Beispiel

1. ADC-Code = 128
2. X = 128 * 12.891 = 1.650.048
3. 1. Ziffer = X / 10^6 = 1 (Vorkommastelle)
4. X = (X % 10^6) * 10 = 6.500.480
5. 2. Ziffer = X / 10^6 = 6 (1. Nachkommastelle)
6. X = (X % 10^6) * 10 = 5.004.800
7. 3. Ziffer = X / 10^6 = 5 (2. Nachkommastelle)
8. X = (X % 10^6) * 10 = 0.048.000
9. 4. Ziffer = X / 10^6 = 0 (3. Nachkommastelle)
10. Ergebnis: 1,650 V

Fehlerbetrachtung

- Wenn wir 6 Nachkommastellen für die Rechnung benutzen, rechnen wir auf 1 Mikrovolt genau.
- Allerdings stellen wir nur 3 Nachkommastellen dar und schneiden die restlichen Stellen ab.
- Da wir nicht Runden sondern Abschneiden, machen wir einen Fehler von maximal 1 Millivolt. Dies ist allerdings schon deutlich kleiner als die Auflösung des ADC und daher vernachlässigbar.