

ADC / DAC

$$V_{in} = V_{in+} - V_{in-}$$

$$V_{out} = (\text{digital value}) \cdot V_{REF+} / (2^N)$$

given:
APB2 clock = 48 MHz
Prescaler 2 → ADCCLK = 24 MHz
3 cycles sampling time
12 bit resolution

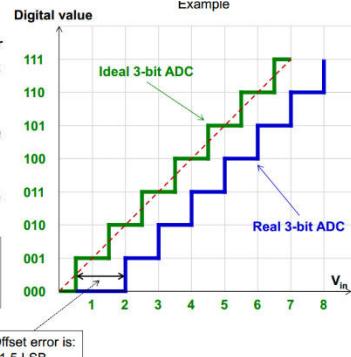
$$T_{\text{total}} = (3 + 12) * 1 / 24 \text{ MHz} = 0.625 \text{ us}$$

sampling rate < 1 / T_{total} = 1.6 Msps

Offset error

- Also called zero-scale error
- Deviation of real N-bit ADC from ideal N-bit ADC at input point zero
- For an ideal N-bit ADC, the first transition occurs at 0.5 LSB above zero
- Can be corrected using the microcontroller

Measuring the offset error:
Zero-scale voltage is applied to analog input and is increased until first transition occurs



Modular Coding/Linking

- High module cohesion
- Low module coupling

Linker tasks

- Merge code sections
- Merge data sections
- Symbol resolution
 - References to other modules
- Address relocation
 - Adapt to new positions of symbols

main.o (part II)

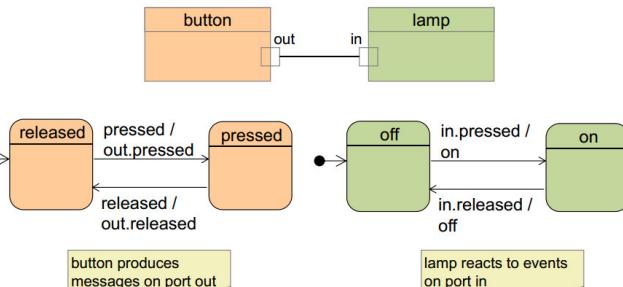
- File section #6: symbols:
 - a**: local data section symbol, at offset 0x00000000
 - b**: local data section symbol, at offset 0x00000004
 - main**: global code section symbol, at offset 0x00000000 (LSB set: Thumb code)
 - square**: global code section symbol, referenced (no definition in main.o)

...	Section #6 '.syms' (SHT_SYMSYM)	# Symbol Name	Value	Bind	Sec	Type	Vis	Size
7	a		0x00000000	Lc	4	Data	De	0x4
8	b		0x00000004	Lc	4	Data	De	0x4
11	main		0x00000001	Gb	1	Code	Hi	0x14
12	square		0x00000000	Gb	Ref	Code	Hi	
...								

Software State Machines

State-event applications

- Process control
 - Washing machines
 - Vending machines
 - Heating systems



Für eine Autowaschanlage soll eine Software Steuerung entwickelt werden, welche wie folgt spezifiziert ist:

- Die Anlage soll im Ruhezustand auf das Drücken der Starttaste warten. Wurde die Starttaste gedrückt, sollen nacheinander die drei Arbeitsschritte 'Waschen', 'Spülen' und 'Trocknen' ausgeführt werden.
- Jeder der drei Arbeitsschritte soll gleich lange dauern. Für die Kontrolle der Zeitzdauer steht ein Timer zur Verfügung.
- Beim Arbeitsschritt 'Waschen' soll Wasser und Shampoo eingeschaltet sein.
- Beim Arbeitsschritt 'Spülen' soll nur Wasser eingeschaltet sein.
- Beim Arbeitsschritt 'Trocknen' soll nur der Luftstrom eingeschaltet sein.
- Bei jedem Arbeitsschritt soll der Ablauf durch Drücken der Stopptaste abgebrochen werden können. Alle Aktionen sollen ausgeschaltet werden und die Steuerung soll in den Ruhezustand zurückkehren.

Folgende Ereignisse (Events) sind definiert:

- | | |
|----------|--------------------------------|
| start | Die Starttaste wurde gedrückt. |
| stop | Die Stoptaste wurde gedrückt. |
| time_out | Der Timer ist abgelaufen. |

Die Akteure und der Timer können mit den folgenden Meldungen angesteuert werden:

- | | |
|-------------|--|
| water_on | Wasser wird eingeschaltet. |
| water_off | Wasser wird ausgeschaltet. |
| shampoo_on | Das Beimischen des Shampoos wird eingeschaltet. |
| shampooAus | Das Beimischen des Shampoos wird ausgeschaltet. |
| air_on | Der Luftstrom für das Trocknen wird eingeschaltet. |
| air_off | Der Luftstrom für das Trocknen wird ausgeschaltet. |
| timer_start | Startet den Timer neu. |

Zeichnen Sie das State-Diagramm der Steuerung in UML, mit den oben definierten Freizeichen und Meldungen.

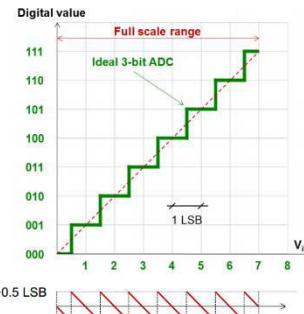
ADC Characteristics

- Sampling rate**
 - Input signal sampled at discrete points in time → discontinuities
 - Should be at least twice the highest frequency component of input signal
 - Nyquist-Shannon sampling theorem
- Conversion time**
 - Time between start of sampling and digital output available
 - Programming a higher resolution may increase conversion time
- Monotonicity**
 - Increase of V_{in} results in increase or no change of digital output and vice-versa

Quantization error

- Analog input is continuous
 - Infinite number of states
- Digital output is discrete
 - Finite number of states
- Introduces an error between -0.5 LSB and +0.5 LSB

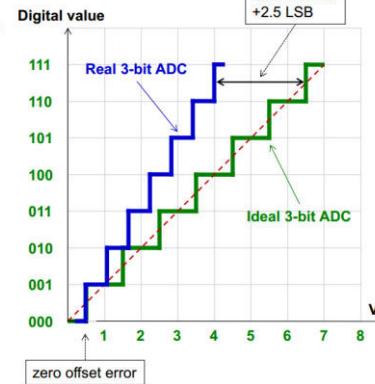
Quantization error can be reduced by reducing LSB, e.g. either by increasing number of bits (resolution) or by reducing V_{REF}
Reducing V_{REF} also reduces full scale range



Gain error

- Indicates how well the slope of an actual transfer function matches the slope of the ideal transfer function
- Expressed in LSB or as a percent of full-scale range (%FSR)
- Calibration with hardware or software possible

$$\text{full-scale error} = \text{offset error} + \text{gain error}$$



Tasks of a Linker

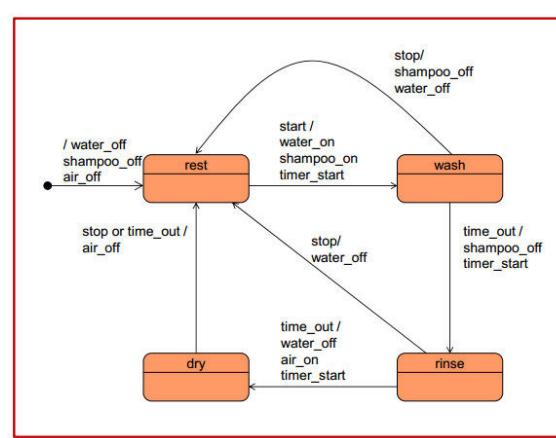
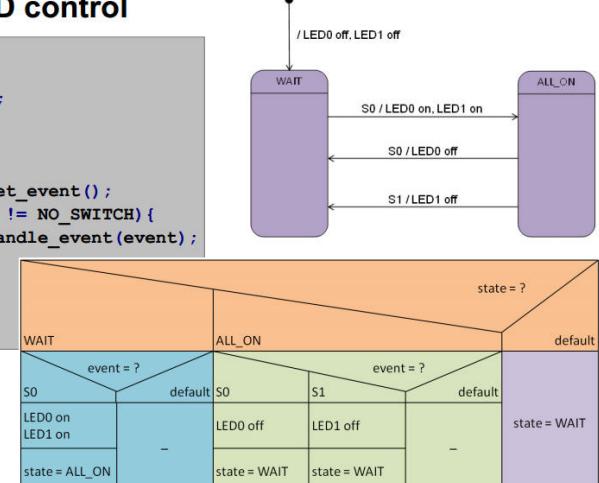
- Merge object file data sections**
 - Place all data sections of the individual object files into one data section of the executable file
- Merge object file code sections**
 - Place all code sections of the individual object files into one code section of the executable file
- Resolve used external symbols**
 - Search missing addresses of used external symbols
- Relocate addresses**
 - Adjust used addresses since merging the sections invalidated the original addresses¹⁾

Example: LED control

```
int main(void)
{
    event_t event;

    fsm_init();
    while (1) {
        event = get_event();
        if (event != NO_SWITCH) {
            fsm_handle_event(event);
        }
        wait();
    }
}
```

→ see code example



```
switch (state) {
    case STATE_A:
        switch (event) {
            case S0:
                action();
                state = NEW_STATE;
            break;
            default:
                state = WAIT;
        }
    break;
    case STATE_B:
        switch (event)
```

Interrupt Performance

Impact on system performance

- Percentage of CPU time used to service interrupts

$$\text{Impact} = f_{\text{INT}} * t_{\text{ISR}} * 100 \%$$

- Example keyboard

$$f_{\text{INT}} = 20 \text{ Hz} = 20 \frac{1}{s}$$

$$t_{\text{ISR}} = 6 \text{ us}^1)$$

$$\text{Impact} = 20 \text{ Hz} * 6 \text{ us} * 100 \% = 0.012 \%$$

- Example serial interface with 230'400 Baud

$$f_{\text{INT}} = 230'400 / 8 = 28'800 \text{ Hz}$$

$$t_{\text{ISR}} = 6 \text{ us}^1)$$

$$\text{Impact} = 28'800 \text{ Hz} * 6 \text{ us} * 100 \% = 17.3 \%$$

Ein Prozessorsystem, welches mit 1 MHz getaktet ist, empfängt über ein Peripheriegerät auf einer Schnittstelle Daten mit einer Rate von 16 kbit/s. Das Peripheriegerät kann 32 bit zwischenspeichern und zeigt dem Prozessor über eine Interrupt-Leitung an, dass die nächsten 32 bit abholbereit sind. Werden die Daten bis zum nächsten Interrupt nicht abgeholt, gehen die Daten verloren.

Die Interrupt Service Routine (ISR) benötigt inklusive Aufruf und Rücksprung im Schnitt 100 Clockzyklen.

- a) Quantifizieren Sie den Einfluss des Interrupts auf das System. D.h. welchen Anteil in Prozent der Gesamtrechenzeit verbringt das System mit der Behandlung der Interrupts?

$$\text{Interruptfrequenz} = (16 \text{ kbit/s}) / 32 \text{ bit} = 500 \text{ Hz}$$

$$\text{interrupt service time} = 100 * 1/(1 \text{ MHz}) = 100 \text{ us}$$

$$\text{Impact} = \text{Interruptfrequenz} * \text{interrupt service time} * 100 \%$$

$$= 500 \text{ Hz} * 100 \text{ us} * 100 \% = 5 \%$$

- b) Bei welcher Datenrate der Schnittstelle würde der Prozessor 100% der Rechenzeit mit der Behandlung von Interrupts verbringen?

$$(x / 32 \text{ bit}) * 100 \text{ us} * 100 \% = 100 \% \rightarrow x = 32 * (1/100) \text{ Mbit/s} = 320 \text{ kBit/s}$$

Annahme: Die Datenrate wird so gewählt, dass das Prozessorsystem 90% der Rechenzeit mit der Behandlung von Interrupts verbringt. Messungen ergeben, dass trotzdem hin und wieder Daten verloren gehen. Nennen Sie eine plausible Ursache.

Die Interrupt Service Time (inklusive Interrupt Latency) ist nicht immer gleich. Die angegebenen 100 Clockzyklen sind ein Durchschnittswert. Der effektive Wert kann schwanken, z.B. weil die Latency je nach Instruktion bei welcher ein Interrupt auftaucht unterschiedlich ist oder z.B. weil die Anzahl Instruktionen in der ISR von den bearbeiteten Daten abhängt. Dadurch hat eine nachfolgende ISR bei einer hohen Auslastung allenfalls zu wenig Zeit für die Bearbeitung. Die Daten können nicht rechtzeitig abgeholt werden.

Remaining Data Types

- Decimal: $0.00002796 = 2.796 \cdot 10^{-5}$
- Binary: $0.000010110111 = 1.0110111 \cdot 2^{-5}$

$$\text{Fraction value} = (1 + F)$$

- Example

$$1.265625 \text{ d} = 1.010001 \text{ b} \rightarrow F = 010001$$

$$\text{Exponent value} = (E - \text{Bias})$$

- Bias: 127 (float) / 1023 (double)

- Example (float):

$$0111'1010b = 122 \rightarrow 122 - 127 = -5 \rightarrow \text{Value} = 2^{-5}$$

$$\text{Value} = (-1)^S \cdot (1 + F) \cdot 2^{(E - \text{Bias})}$$

- More exponent bits \rightarrow wider range of numbers
- More fraction bits \rightarrow higher precision

$$\text{Value} = 2^{-5} \rightarrow -5 + 127 = 122 = 0111'1010b$$

Single precision (float)



- 32 bits
- Exponent bias: 127
- Dynamics: $-3.4 \cdot 10^{38} \dots -1.17 \cdot 10^{-38}$, 0, $1.17 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$
- Resolution: $2^{-24} = 0.6 \cdot 10^{-7} \rightarrow 7$ digits

Double precision (double)



- 64 bits
- Exponent bias: 1023
- Dynamics: $-1.8 \cdot 10^{308} \dots -2.2 \cdot 10^{-308}$, 0, $2.2 \cdot 10^{-308} \dots 1.8 \cdot 10^{308}$
- Resolution: $2^{-53} = 0.11 \cdot 10^{-15} \rightarrow 15$ digits

Decimal 7.5 d = ?

$$\begin{aligned} 7.5 : 4 &= 1 \\ 3.5 : 2 &= 1 \\ 1.5 : 1 &= 1 \\ 0.5 : 2^{-1} &= 1 \end{aligned}$$

$$\begin{aligned} &1.111*2^2 \\ &\text{EXP} = 2 \\ &\text{E} = 2 + 127 = 129 \\ &= 1000'0001 \end{aligned}$$

0 1000'0001 111000... (so geschrieben reicht es)
In Hex = 40F0000h (muss man nicht)

Binary 10111111'01010000'00000000'00000000 = ?

$$S = -1$$

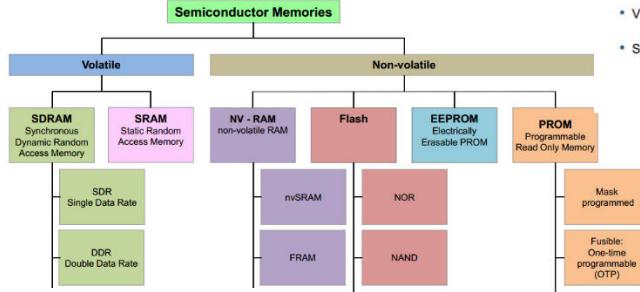
$$\begin{aligned} E &= 0111'1110 \\ &- 0111'1111 \\ &= -1 \end{aligned}$$

$$\begin{aligned} F &= 101 \\ \text{Fraction} &= 1.101 \end{aligned}$$

$$\begin{aligned} \text{Value} &= 1.101 * 2^{-1} \\ &= 0.1101 \\ &= (-1) * 1/2 + 1/4 + 1/16 \text{ wegen sign} \end{aligned}$$

Memory

Solid State Devices



■ SRAM Static Random Access Memory

- Read and write
 - All accesses take roughly the same time
 - Access time independent of location of data item in memory
 - Access time independent of previous access¹⁾
- Volatile
 - Memory content retained only as long as device is powered
- Static
 - Storage elements similar to flip-flops / latches
 - No refresh required
 - Refresh: periodic reading and rewriting of memory cell to maintain the content

■ Address Regions

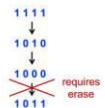
- SRAM1 = 112K bytes = 28K words
- SRAM2 = 16K bytes = 4K words
- SRAM3 = 64K bytes = 16K words
- CCM = 64K bytes = 16K words

■ Flash

- Non-volatile memory
- Memory content retained after power off
- Used to store code and persistent data
- Like most on-chip flash memories STMF4xx uses a so-called NOR topology

■ Flash Is Partitioned into Sectors

- Sectors can only be erased as a whole
- Writing through control registers – no direct memory write accesses



■ Write Operations (Programming)

- Can only change bits from '1' to '0'
 - Otherwise an erase operation is required
- Word, half-word or byte access possible
- Writing a double word ~16 us
 - I.e. around 1000 times slower than on an SRAM

■ Erase Operations

- Change all bits from '0' to '1'
 - Only possible by sector or by bank, not on a word
- Erase of a 128 Kbytes sector takes between 1 and 2 seconds¹⁾
- Endurance: 10'000 erase cycles²⁾
- Sector may not be accessed (write or read) during erase
 - I.e. execute program from another sector or from SRAM during erase

Extend On-chip Memory

- Flexible Memory Controller (FMC)
 - SRAM and NOR flash
 - Synchronous DRAM
 - NAND flash

Flexible Memory Controller

zhaw School of Engineering and Embedded Systems

Writing a 32-bit Word from System Bus (32 Data Lines)

- to a 32-bit wide external memory (32 data lines)
- system bus [write word] → external bus [write word] → t
- to a 16-bit wide external memory (16 data lines)
 - Word stored in FMC-FIFO
 - System bus is released for other accesses
 - FMC-FIFO content is transferred to external memory using 1 to 4 bus cycles
- to an 8-bit wide external memory (8 data lines)
 - system bus [write word] → external bus [write hword 1, write hword 2] → t
- to an 8-bit wide external memory (8 data lines)
 - system bus [write word] → external bus [write byte 1, write byte 2, write byte 3, write byte 4] → t

Asynchronous SRAM Device

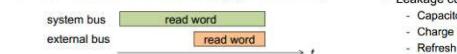
- No clock input

ADDSET and DATAST

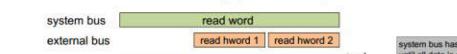
- Configuring length of access cycles
- HCLK programmed to 84 MHz during start-up of CT-Board
 - HCLK = Frequency of CPU and internal bus

Reading a 32-bit Word from

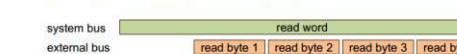
- a 32-bit wide external memory (32 data lines)



- a 16-bit wide external memory (16 data lines)



- an 8-bit wide external memory (8 data lines)



Floating Gate Transistor Technology

- NOR flash topology
 - n-channel transistors arranged in NOR gate fashion
 - connected in parallel to ground
 - Allows one-to-one replacement of PROMs
 - Medium cell area → medium density, medium cost per bit
- NAND flash topology
 - n-channel transistors arranged in NAND gate fashion
 - connected in series to ground (requiring addressing overhead)
 - Low cell area → high density, low cost per bit
- EEPROM (Electrically Erasable PROM)
 - Similar to NOR flash but allows byte-wise erase instead of only block erase
 - High cell area → low density, high cost per bit
 - Special applications

On-chip Memory

- SRAM Static, volatile, random access, flip-flop based structure
- Flash Non-volatile, higher latency than SRAM
 - sector structure: erasing only for whole sectors

Flexible Memory Controller

- Bridge between system bus and external bus
- Single system bus cycle may require multiple cycles on the external bus

Asynchronous SRAM

- Off-chip version of SRAM, control signals CS, OE, WE

SDRAM – Synchronous Dynamic RAM

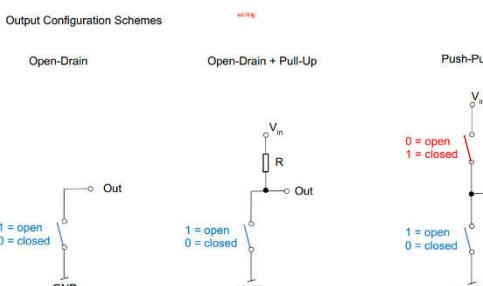
- Capacitor-based, refresh, high density, synchronous interface, latency

Flash

- NOR medium density, suitable for random read access
- NAND high density, high latency → disks

General Purpose I/O (GPIO)

GPIO Configurations



Port bit configuration

GP = general-purpose
PP = push-pull
PU = pull-up
PD = pull-down
OD = open-drain
AF = alternate function

■ SRAM Static Random Access Memory

- Read and write
 - All accesses take roughly the same time
 - Access time independent of location of data item in memory
 - Access time independent of previous access¹⁾
- Volatile
 - Memory content retained only as long as device is powered
- Static
 - Storage elements similar to flip-flops / latches
 - No refresh required
 - Refresh: periodic reading and rewriting of memory cell to maintain the content

■ Address Regions

- SRAM1 = 112K bytes = 28K words
- SRAM2 = 16K bytes = 4K words
- SRAM3 = 64K bytes = 16K words
- CCM = 64K bytes = 16K words

■ Flash

- Non-volatile memory
- Memory content retained after power off
- Used to store code and persistent data
- Like most on-chip flash memories STMF4xx uses a so-called NOR topology

■ Flash Is Partitioned into Sectors

- Sectors can only be erased as a whole
- Writing through control registers – no direct memory write accesses

■ Write Operations (Programming)

- Can only change bits from '1' to '0'
 - Otherwise an erase operation is required
- Word, half-word or byte access possible
- Writing a double word ~16 us
 - I.e. around 1000 times slower than on an SRAM

■ Erase Operations

- Change all bits from '0' to '1'
 - Only possible by sector or by bank, not on a word
- Erase of a 128 Kbytes sector takes between 1 and 2 seconds¹⁾
- Endurance: 10'000 erase cycles²⁾
- Sector may not be accessed (write or read) during erase
 - I.e. execute program from another sector or from SRAM during erase

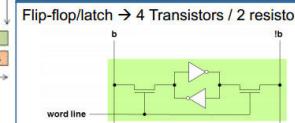
Synchronous Dynamic Random Access Memory



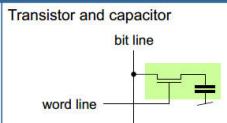
zhaw School of Engineering and Embedded Systems

SDRAM – Synchronous Dynamic RAM

Static RAM (SRAM)



Synchronous Dynamic RAM (SDRAM)



Large cell

- Low density, high cost
- Up to 64 Mb per device

Almost no static power consumption

- Static i.e. no accesses taking place

Asynchronous interface (no clock)

- Simple connection to bus
- All accesses take roughly the same time
- ~5ns per access → 200 MHz
- Suitable for distributed accesses

Small cell

- High density, low cost
- Up to 4 Gb per device

Leakage currents

- Requires periodic refresh

Synchronous interface (clocked)

- Requires dedicated SDRAM Controller
- Long latency for first access of a block
- Fast access for blocks of data (bursts)
- Large overhead for single byte

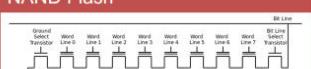
Non-volatile Memory – Flash

zhaw School of Engineering and Embedded Systems

NOR Flash



NAND Flash



Applications

- Execute code directly from memory
- Persistent device configurations (replacement of EEPROM)
- File-based IO, disks
- Large amounts of sequential data (images, SD cards, SSD)
- Load programs into RAM before executing

Density

- Medium Up to 2 GBit = 256 MByte
- High Up to 1 Tbit

Interface

- Read same as asynchronous SRAM
- Types with serial interface available
- Special NAND flash interface
- Error correction for defective blocks

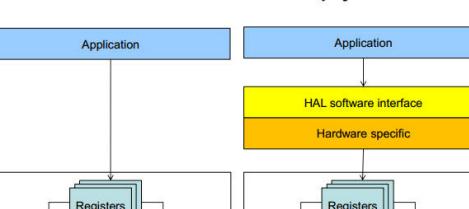
Access

- Random access read ~0.12 µs
- Writing individual bytes possible
- Slow writes ~180 µs / 32 Byte
- Slow random access read: 1. Byte 25 µs, then 0.03 µs each
- Writing individual bytes difficult
- Fast block write ~300 µs / 2'112 Bytes

Hardware Abstraction Layer (HAL)

zhaw School of Engineering and Embedded Systems

Abstraction between software and physical hardware



Functionality

- Defines a set of routines, protocols and tools for interacting with hardware
- Provides abstract, high level functions to access hardware without detailed knowledge of HW

Properties

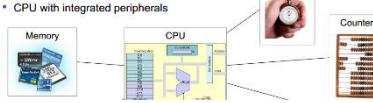
- Located between software and physical hardware
- Function as driver interface or as wrapper for providing a common interface between existing drivers and higher level code
- Can provide interfaces for common software stacks (RTOS or middleware components like serial communication, Ethernet or file systems)



Microcontroller

Single Chip Solution

- CPU with integrated peripherals



Embedded Systems

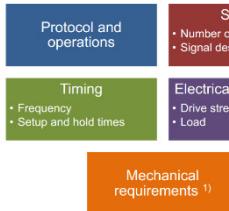


Real Time

anti-lock braking system, process control



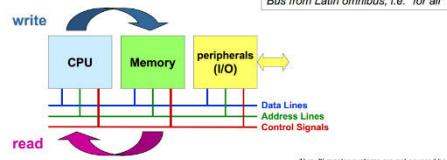
Bus Specification



System Bus

- Interconnects CPU with memory and peripherals
- CPU acts as master¹⁾
 - Initiating and controlling all transfers
- Peripherals and memory act as slaves
 - Responding to requests

Etyymology: Bus from Latin *omnibus*, i.e. "for all"



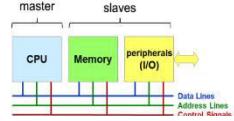
Signal Groups

- Address lines
 - Unidirectional: From master to slave
 - Number of lines → yields size of address space

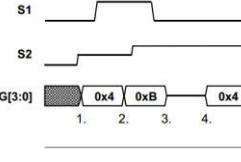
Cortex-M 32 parallel address lines
 $2^{32} = 4$ Giga addresses
 $0x0000'0000 - 0xFFFF'FFFF$

- Data lines
 - 8, 16, 32 or 64 parallel lines of data
 - bidirectional (read/write)

- Control signals
 - Control read/write direction
 - Provide timing information



Timing Diagrams: Notations



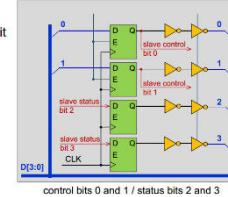
Signal S1 changing from '0' to '1' (rising edge) and back from '1' to '0' (falling edge)

Signal S2 changing from '0' to high-impedance ('Z', tri-state) and then to '1'

- Group G of 4 signals changing
- from an unknown value to 0x4 i.e. G[3] = '0', G[2] = '1', G[1] = '0', G[0] = '0'
 - from 0x4 to 0xB
 - from 0xB to all signals high-impedance ('Z', tri-state)
 - from high-impedance back to 0x4



Controlstatus register on example bus



Control Bits

- Allow CPU to configure slave
 - CPU (software) writes to register bit
 - Slave hardware uses output of register bit
 - E.g. to select a mode
 - Usually read/write
- Status Bits**
- Allow CPU to monitor a slave
 - Slave writes status into register bit
 - CPU (software) reads register bit
 - Usually read-only

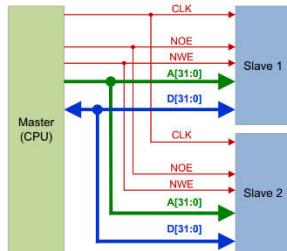
Same register may contain control and status bits

Block Diagram

- Address lines A[31:0]

- Data lines D[31:0]

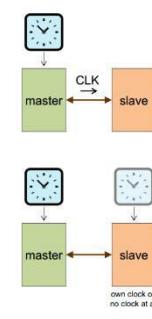
- Control
 - NWE write enable active-low
 - NOE output enable (read) active-low



Bus Timing Options

- Synchronous
 - Master and slaves use a common clock¹⁾
 - Clock edges control bus transfer on both sides
 - Used by almost all on-chip busses
 - Off-chip: DDR and synchronous RAM
- Asynchronous
 - Slaves have no access to the clock of the master
 - Control signals carry timing information to allow synchronization
 - Widely used for low data-rate off-chip memories
 - parallel flash memories and asynchronous RAM

¹⁾ Often this is a dedicated clock signal from master to slave but the clock could e.g. also be encoded in a data signal



Full Address Decoding

- All address lines are decoded (checked)
- A control register can be accessed at exactly one location
- 1 : 1 mapping
- a unique address maps to a single hardware register (physical memory location)

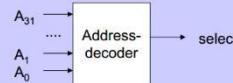
Partial Address Decoding

- Only a sub-set of the address lines are decoded
- Detects an address range or a set of addresses
- n : 1 mapping
 - n unique addresses map to the same hardware register (physical memory location)

Example

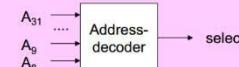
- 32-bit address space of Cortex-M

Full address decoding → all addresses from A31 to A0



select is active for exactly one address
e.g. at 0x4000'8234

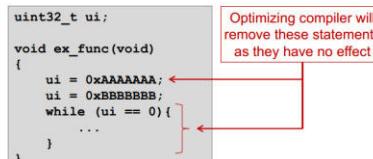
Partial address decoding → only addresses from A31 to A8



select is active for any address within a given range (e.g. ignoring some lower address lines).
e.g. from 0x4000'8200 to 0x4000'82FF
→ 0x4000'82xx

Situation

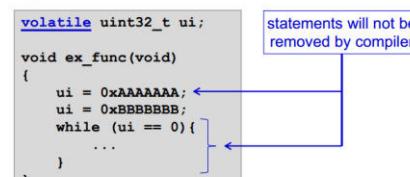
- Compiler may remove statements that have no effect from the compiler's point of view



- Accesses to control registers (read/write) are memory accesses
- If writing has a side effect on the hardware and/or if reading may result in a different value than was set before in the code, the program will not behave as intended.

Solution

- Use qualifier `volatile` in variable declaration



- Tell compiler that variable may change outside the control of the compiler
 - E.g. by hardware or by an interrupt handler

- Compiler cannot make any assumption on the value

- Needs to execute all read/write accesses as programmed

- Prevents compiler optimizations

Using Preprocessor Macros → #define

```
dereference pointer           cast
#define LED31_0_REG    (*((volatile uint32_t *)(0x60000100)))
#define BUTTON_REG     (*((volatile uint32_t *)(0x60000210)))

// Write LED register to 0xBBCC'DDDE
LED31_0_REG = 0xBBCC'DDDE;

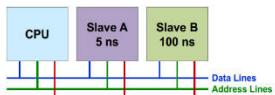
// Read button register to aux_var
aux_var = BUTTON_REG;
```

Problem: Individual Slave Access Times

- If slowest slave defines bus cycle time

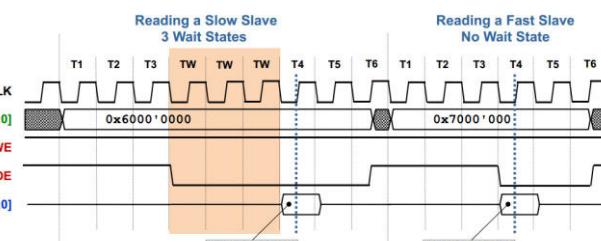
- Reduced bus performance

- How can we get an individual bus cycle time for each slave?



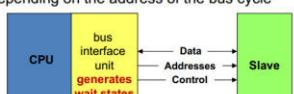
Introduce Individual Wait States for Slow Slaves

- Wait states are inserted depending on the address of an access



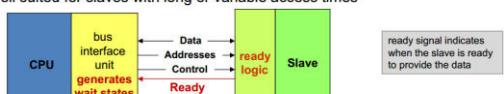
Two Possibilities

- Individual wait states can be programmed at a bus interface unit
 - Depending on the address of the bus cycle



- Slave tells bus interface unit when it is ready

- Well suited for slaves with long or variable access times



Serial Data Transfer

Communication Modes

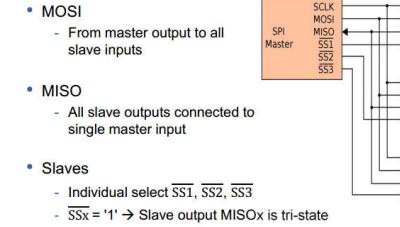
- Simplex Unidirectional, one way only
- Half-duplex Bidirectional, only one direction at a time
- Full-duplex Bidirectional, both directions simultaneously

Timing

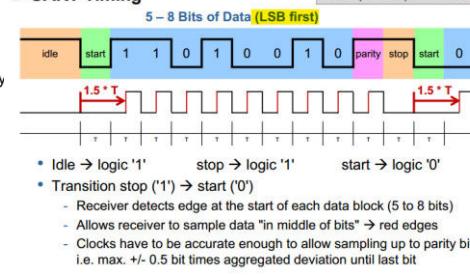
- Asynchronous Each node uses an individual clock
- Synchronous Both nodes use same clock
Clock often provided by master

Single Master – Multiple Slaves

- Master generates a common clock signal for all slaves



UART Timing

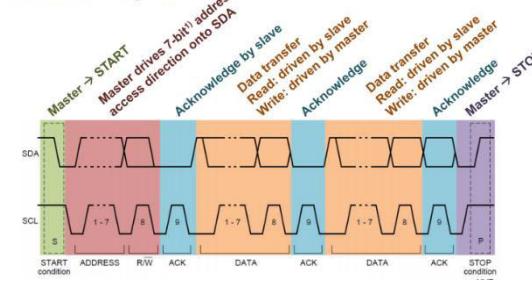


UART Characteristics

- Synchronization
 - Each data item (8-bits) requires synchronization
- Asynchronous data transfer
 - Mismatch of clock frequencies in TX and RX
 - Requires overhead for synchronization → additional bits
 - Requires effort for synchronization → additional hardware
- Advantage
 - Clock does not have to be transmitted
 - Transmission delays are automatically compensated
- On-board connections
 - Signal levels are 3V or 5V with reference to ground
 - Off-board connections require stronger output drivers (circuits)

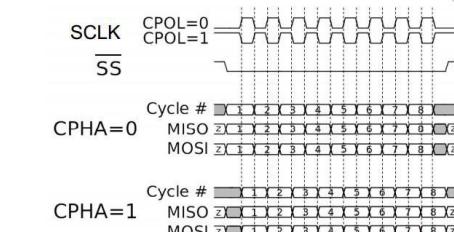
I²C Bus

Addressing Slaves

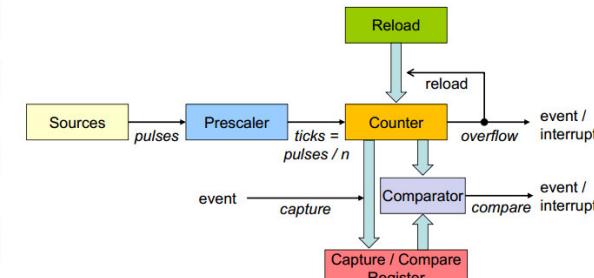


Data and Clock

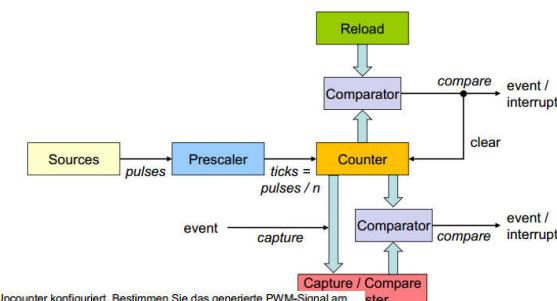
- Summary of all possible combinations



Downcounting functional block diagram (simplified)



Upcounting functional block diagram (simplified)



Der Timer ist als Upcounter konfiguriert. Bestimmen Sie das generierte PWM-Signal am Ausgang (Zahlen + Skizze). Die Source liefert ein Signal der Frequenz 0,5 MHz.

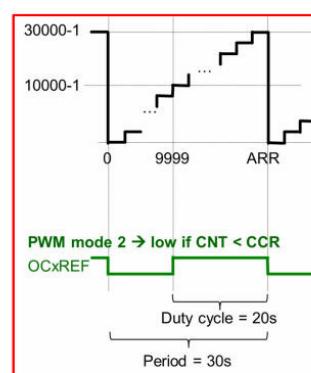
Die relevanten Konfigurationsregister sind wie folgt initialisiert:

```

TIM3_PSC = 0x01F3
TIM3_ARR = 0x7530
TIM3_CCR1 = 0x2710
TIM3_CCMR1 = 0x0070
  
```

Welches Signal wird erzeugt? Zeichnen Sie das Signal und geben Sie die Werte für Period und Duty Cycle an (Zeitangaben).

Lösung:
Gegebene Register entsprechen:
 TIM3_PSC = 0x01F3 // (500-1) -> 1 kHz
 TIM3_ARR = 0x7530 // (30000-1) -> Periode 30 Sekunden
 TIM3_CCR1 = 0x2710 // (10000-1) -> 10 Sekunden
 TIM3_CCMR1 = 0x0070 // PWM Mode 2

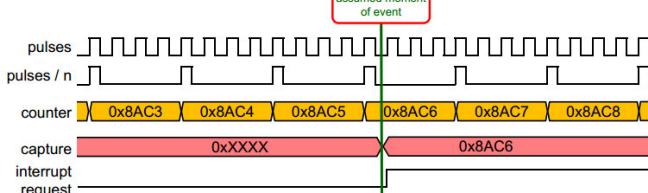


Timer / Counter

Capture example

- Stop watch
- At which moment in time does the user push the button?

- Time of event is captured
- Count continues



Timer 4 des STM32F429 ist bereits als Upcounter konfiguriert und läuft. Das Reload Register enthält folgenden Wert:

TIM4_ARR = 0x9C3F

- a) Geben Sie den Wert f für das CCR-Register an, damit ein Duty Cycle von 25% mittels PWM Mode 1 erzeugt wird.

TIM4_CCR1 = 0x752F // entspricht (30000-1) -> 0x2710=(10000)

TIMx_CNT zählt von 0 ...39999 = 40000 Ticks
-> Duty Cycle 25% entspricht 10000 Ticks

PWM Mode 1 (Upcounting): OC_REFx='1' as long as TIMx_CNT<TIMx_CCR
otherwise OC_REFx='0'
-> 0...9999 = 10000 Ticks -> TIMx_CNT<TIMx_CCR -> OC_REFx='1'
-> 10000...39999 = 30000 Ticks -> TIMx_CNT>TIMx_CCR -> OC_REFx='0'

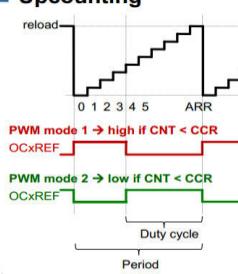
- b) Nun ist Timer 4 als Downcounter im PWM Mode 2 statt als Upcounter im PWM Mode 1 konfiguriert. Was müssen Sie ändern, um ein identisches elektrisches Signal zu erhalten (Werte)?

TIM4_CCR1 anpassen

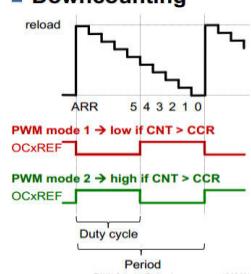
TIM4_CCR1 = 0x752F // entspricht (30000-1)

PWM Mode 2 (Downcounting): OC_REFx='1' as long as TIMx_CNT>TIMx_CCR
otherwise OC_REFx='0'
-> 39999...30000 = 10000 Ticks -> TIMx_CNT>TIMx_CCR -> OC_REFx='1'
-> 29999...0 = 30000 Ticks -> TIMx_CNT<=TIMx_CCR -> OC_REFx='0'

Upcounting



Downcounting



Microcontroller

Gegeben ist ein System Bus mit den 6 Adresslinien A[5:0].

- a) Wie viele Bytes können damit adressiert werden?

$$2^6 = 64 \text{ Byte Adressen}$$

- b) Unter wie vielen Adressen kann ein 8-bit Control Register angesprochen werden, wenn dafür 4 dieser Adressleitungen dekodiert werden?

$$2^{6-4} = 4 \text{ Byte Adressen}$$

- c) Unter welchen Adressen (in Hex) kann das Control Register angesprochen werden, wenn nur die oberen 4 Adressleitungen wie folgt dekodiert werden? Adresse 5 4 3 2 1 0
select = A[5] & A[4] & !A[3] & !A[2]

$$1100XXb \rightarrow 0x30, 0x31, 0x32, 0x33$$

- d) Unter welchen Adressen (in Hex) kann das Control Register angesprochen werden, wenn nur die unteren 4 Adressleitungen wie folgt dekodiert werden:
select = !A[3] & A[2] & A[1] & !A[0]

$$X00110b \rightarrow 0x06, 0x16, 0x26, 0x36$$

- e) Unter welchen Adressen (in Hex) kann das Control Register angesprochen werden, wenn nur die mittleren 4 Adressleitungen wie folgt dekodiert werden:
select = !A[4] & !A[3] & A[2] & !A[1]

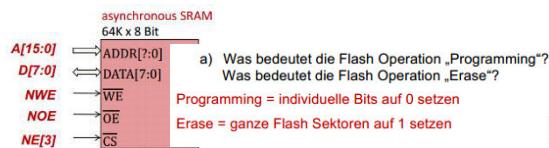
$$X0010Xb \rightarrow 0x04, 0x05, 0x24, 0x25$$

- f) Wie müssen die Adressen dekodiert werden, wenn das Control Register genau unter der Adresse 0x28 angesprochen werden soll?

$$\text{select} = A[5] \& !A[4] \& A[3] \& !A[2] \& !A[1] \& !A[0]$$

Memory

Gegeben ist der folgende 'asynchronous SRAM' Baustein.



- a) Wie viele Adresspins benötigt der Baustein?

$$64K = 2^{16} \rightarrow 16 \text{ Adresspins} \rightarrow ADDR[15:0]$$

- d) Beim Entwickeln der Software stellen Sie fest, dass Sie unter der Adresse 0x6BFF'0000 auf das identische Byte des Bausteins wie unter 0x6800'0000 zugreifen. Was ist die Erklärung?

Partial Address Decoding: Die Bits A[25:16] sind nicht angeschlossen und werden deshalb nicht dekodiert.

- e) Unter wie vielen 64KByte Adressblöcken kann auf den Baustein zugegriffen werden?

$$A[25:16] \rightarrow 10 \text{ Adresslinien} \rightarrow 2^{10} = 1024 \text{ Adressblöcke}$$

0x68XX'0000
0x69XX'0000
0x6AXX'0000
0x6BXX'0000

Serial datatransfer

Auf einer seriellen asynchronen Übertragungsleitung (UART) mit 19'200 Bit/s, 7 Daten-Bits, und einem Stop-Bit (ohne Parity Bit) soll die Zeichenfolge "AC" übertragen werden.

ASCII('A') = 0xA1 = 100 0001b

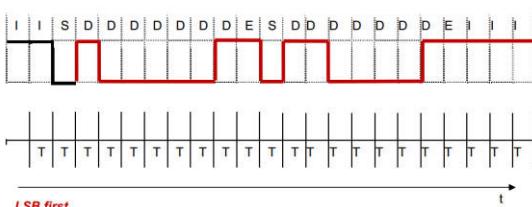
ASCII('C') = 0xC3 = 100 0011b

- a) Wie lange dauert die Übertragung eines Bits (Periode T)?

$$1/19\,200 \text{ s} = 52.1 \mu\text{s}$$

- b) Zeichnen Sie den zeitlichen Verlauf der Übertragung ein. Bezeichnen Sie die einzelnen Bits wie folgt:

S → Start-Bit
E → Stop-Bit (End)
D → Daten-Bit
I → Idle-Bit (keine Übertragung)



LSB first

- a) Wie synchronisieren sich Sender und Empfänger bei einer UART?

Mit Hilfe der negativen Flanke des ersten übertragenen Bits

Letztes Bit (Stop) oder Idle → '1'

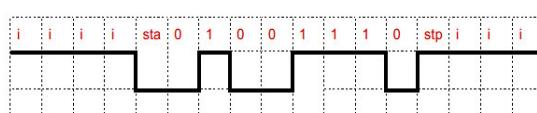
Erstes Bit (Start) = '0'

- b) Wie viele Nutzdaten-Bytes kann man pro Sekunde übertragen, wenn die UART eingestellt ist auf 9600 baud (entspricht hier 9600 bit/Sek.), 8 Datenbits, 2 Stopbits, 1 Paritybit. (bedeutet: Frage an die Prüfung)

Pro Nutzdatei-Byte werden 1 Startbit, 8 Datenbits, 2 Stopbits und 1 Paritybit, d.h. total 12 Bits übertragen

$$9600/12 \text{ bits/s} = 800 \text{ Bytes/s}$$

Auf einer seriellen Datenleitung (UART) messen Sie den folgenden zeitlichen Verlauf. Die Übertragung verwendet ein Startbit, ein Stopbit, 8 Datenbits ohne Parity-bit bei 4'800 baud.



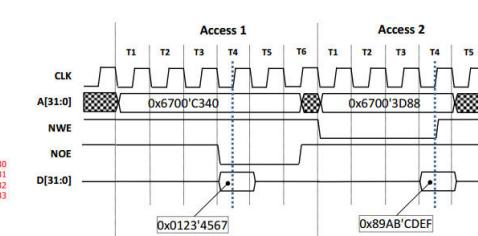
- a) Notieren Sie die Bedeutung der Bits im Zeitverlauf und geben Sie die übertragenen Daten als Hexwert an.

0x72

- b) Wie viele Daten-Bytes (ohne Synchronisations-Overhead) kann man pro Sekunde maximal übertragen?

$$4800/(1+8+1) = 480 \text{ Bytes/s}$$

Gegeben sind die folgenden Buszugriffe



Schreiben Sie Codesequenzen in C für die folgenden Fälle. Verwenden Sie unsigned integer Typen aus stdint.h

- a) Lesen Sie den Wert eines 8-bit Control Registers an der Adresse 0x6100'0007 in eine von Ihnen zu definierende Variable ein.

```
#define MY_BYTE_REG (*((volatile uint8_t *) (0x61000007)))
```

```
uint8_t my_var;
```

```
my_var = MY_BYTE_REG;
```

- b) Setzen Sie sämtliche Bits eines 16-Bit Control Registers an Adresse 0x6100'0008 auf '1'.

```
#define MY_HALFWORD_REG (*((volatile uint16_t *) (0x61000008)))
```

```
MY_HALFWORD_REG = 0xFFFF;
```

- c) Warten Sie in einer Schleife, bis Bit 15 im 32-bit Control Register an der Adresse 0x6100'000C auf '1' gesetzt ist.

```
#define MY_WORD_REG (*((volatile uint32_t *) (0x6100000C)))
```

```
while (! (MY_WORD_REG & 0x00008000)) { }
```

- d) Setzen Sie Bit 16 im Control Register an Adresse 0x6100'0010 auf '1' ohne die anderen Bits des Registers zu verändern.

```
#define MY_WORD_REG2 (*((volatile uint32_t *) (0x61000010)))
```

```
MY_WORD_REG2 |= 0x00010000;
```

Adresse	Wert (Byte)
0x6700'3D88	0x89
0x6700'3D8A	0xAB
0x6700'3D89	0xCD
0x6700'3D8B	0xEF

GPIO

Welche GPIO-Ports müssen für die Ein- bzw. Ausgabe angesprochen werden?

Geben Sie den Namen und die Basisadressen an.

GPIO Port A → Eingabe 0x40020000

GPIO Port G → Ausgabe 0x40021C00

Welche Register müssen für die Ausgabe an der LED konfiguriert werden?

Geben Sie die Namen und die Adressen an.

MODER 0x40021C00

TYPER 0x40021C04

SPEEDR 0x40021C08

PUPDR 0x40021C0C

- a) Zeichnen Sie das Timing Diagramm (Signale SCLK, SS#, MOSI, MISO) für ein SPI Interface

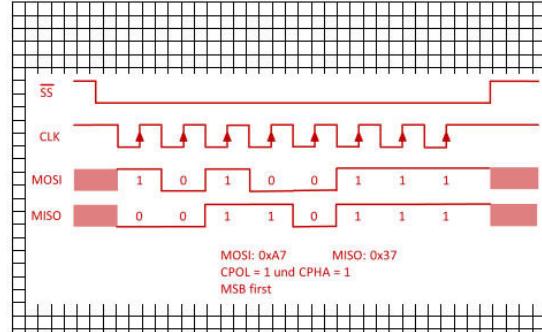
- 8 Bit Daten (MOSI: 0xA7, MISO: 0x37)

- Mode 3 d.h. CPOL = 1 und CPHA = 1

- b) Zeichnen Sie die Sampling Edges ein.

- c) Wie lange dauert eine Bit Cell d.h. eine Clockperiode wenn SCLK eine Frequenz von 100kHz hat?

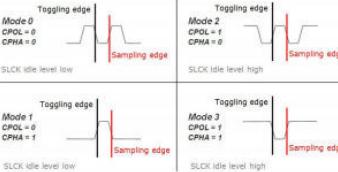
0.01 ms



MOSI: 0xA7

CPOL = 1 und CPHA = 1

MSB first



LSB first

- a) Wie synchronisieren sich Sender und Empfänger bei einer UART?

Mit Hilfe der negativen Flanke des ersten übertragenen Bits

Letztes Bit (Stop) oder Idle → '1'

Erstes Bit (Start) = '0'

- b) Wie viele Nutzdaten-Bytes kann man pro Sekunde übertragen, wenn die UART eingestellt ist auf 9600 baud (entspricht hier 9600 bit/Sek.), 8 Datenbits, 2 Stopbits, 1 Paritybit. (bedeutet: Frage an die Prüfung)

Pro Nutzdatei-Byte werden 1 Startbit, 8 Datenbits, 2 Stopbits und 1 Paritybit, d.h. total 12 Bits übertragen

$$9600/12 \text{ bits/s} = 800 \text{ Bytes/s}$$

Auf einer seriellen Datenleitung (UART) messen Sie den folgenden zeitlichen Verlauf. Die Übertragung verwendet ein Startbit, ein Stopbit, 8 Datenbits ohne Parity-bit bei 4'800 baud.



- a) Notieren Sie die Bedeutung der Bits im Zeitverlauf und geben Sie die übertragenen Daten als Hexwert an.

0x72

- b) Wie viele Daten-Bytes (ohne Synchronisations-Overhead) kann man pro Sekunde maximal übertragen?

$$4800/(1+8+1) = 480 \text{ Bytes/s}$$

Counter

Erklären Sie in Stichworten die einzelnen Funktionseinheiten eines Timers anhand der Tabelle.

Register	Inhalt	Funktion(en)
Prescaler	Divisor für Eingangssignal	Es wird nur jeder n-te Wert gezählt.
Counter	Aktueller Timerwert	Der Wert wird mit jedem n-ten Tick um eins erhöht oder erniedrigt
Reload	Wert für Timer-Überlauf	Upcounter: Timer zählt bis zu diesem Wert, dann Überlauf Downcounter: Startwert für Timer
Capture/Compare	Vergleichswert	Capture: Bei einem Event wird der Wert des Counters hier gespeichert Compare: Sobald der Wert vom Counter erreicht wird, wird ein Event ausgelöst.

b) Erläutern Sie die Funktion Capture.

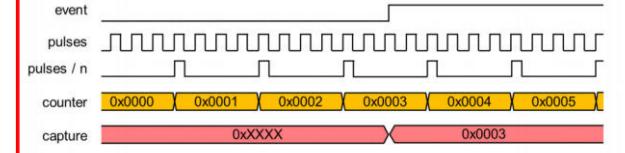
Bei einem Event wird der Inhalt des Counter Registers in das Capture / Compare Register kopiert. Der Counter läuft weiter.

c) Erläutern Sie die Funktion Compare.

Sobald der Counter den Wert des Capture / Compare Register erreicht hat, wird ein Event oder ein Interrupt ausgelöst. Der Counter läuft weiter.

Ergänzen Sie das gegebene Timing Diagramm. Die Funktion Capture wird bei steigender Flanke des Signals „event“ ausgelöst. Der Timer ist als Upcounter konfiguriert, und der Prescaler ist auf 4 (im Register steht 0x03) eingestellt. Die Startwerte entsprechen sonst dem Zustand nach einem Reset.

Lösung:



- a) Als Source soll die interne Clock CK_INT mit 84 MHz verwendet werden. Setzen Sie die Bits im entsprechenden Register auf die notwendigen Werte.
Hinweis: andere Bits des Registers sollen nicht verändert werden.

```
TIM3_SMCR &= 0xFFFF; // TIM3_SMCR [2:0] 0b000
```

- b) Der Timer soll als Upcounter konfiguriert werden. Hinweis: andere Bits des Registers sollen nicht verändert werden.

```
TIM3_CR1 &= 0xFF8F
```

- c) Die Zeit für den Timerüberlauf soll 200 ms betragen. Welche Werte müssen Sie in die Register PSC und ARR schreiben (Angabe hexadezimaler Werte)?
Hinweis: Es sind verschiedene richtige Lösungen möglich.

```
TIM3_PSC = 0x20CF // (8400 - 1) entspricht 10 kHz
TIM3_ARR = 0x07CF // (2000 - 1) entspricht 200 ms
```

Modulares Programming

Nennen Sie mindestens vier Punkte bei welchen die modulare Programmierung besonderen Nutzen bringt.

Thema	Nutzen
Teamarbeit	Mehrere Entwickler können auf denselben Sourcen verteilt arbeiten
Nützlich Aufteilung und Strukturierung von Programmen in konsistente Komponenten	Ermöglicht Wiederverwendung von Komponenten
Individuelle Verifikation einzelner Komponenten	Zum Nutzen aller welche die Komponente verwenden
Bibliotheken von mehrfach verwendbaren Funktionen und Typen	Wiederverwendung anstelle von „das Rad neu erfinden“

Weitere:

Mischen von Komponenten die in unterschiedlichen Programmiersprachen geschrieben wurden Komponenten können unabhängig von der Programmiersprache wiederverwendet werden

Partielles Kompilieren Nur geänderte Komponenten müssen neu kompiliert werden.

Vom C Programm zum Executable: Nennen und beschreiben Sie die vier Schritte (Tools), welche bei der Übersetzung eines C Programmes in ein ausführbares Executable File notwendig sind.

Name des Schrittes (Tools)	Was macht der Schritt?
Preprocessor	Verarbeitet die Preprocessor Statements (z.B. #define, #include). Textprocessing: Ersetzen und Ergänzen von Inhalten. Ergebnis: Textfile mit modifiziertem C Sourcecode
Compiler	Übersetzt C Sourcecode in prozessorspezifische, symbolische Assemblerbefehle. Ergebnis: Textfile mit Assemblercode
Assembler	Übersetzt Assemblercode in binäre Maschinenbefehle. Ergebnis: binäres Objectfile mit Maschinenbefehlen, Daten, Symbol Tabelle, Relocation Tabelle.
Linker	Fügt mehrere Objectfiles zu einem ausführbaren Objektfile zusammen (Merge Sections), löst die Symbol Referenzen zwischen den einzelnen Objektfiles auf (Resolution) und passt Zugriffe auf Symbole an (Relocation). Ergebnis: Ausführbares Objektfile (Executable)

Welche C Linkage haben die unterstrichenen Namen der gegebenen C Definitionen?

Code	External Linkage	Internal Linkage	No Linkage
int <u>square</u> (int v) { return v * v; }	x		
int <u>square</u> (int v) { int res = v * v; return res; }			x
static int <u>square</u> (int v) { return v * v; }		x	

Linker-Task	Situation
1) Merge code sections	a) Modul A ruft eine Funktion aus Modul B auf
2) Merge data sections	b) Modul A und Modul B enthalten Instruktionen
3) Resolve referenced symbols	c) Modul A und Modul B enthalten globale Daten
4) Relocate addresses	d) Verwendete Referenzen im Code müssen an die neue Lage der Symbole angepasst werden

1 – b
2 – c
3 – a
4 – d

Nennen Sie die Object File Sections welche die entsprechenden Informationen enthalten.

Object File Section Name	Enthaltene Information
Code Section	Instruktionen
Data Section	Globale Daten
Symbol Table	Liste der internen, importierten und exportierten Symbole
Relocation Table	Adressen der Instruktionen und Daten deren Zugriff auf Symbole im Laufe des Linkens angepasst werden müssen

Was ist eine Native-Compiler Tool Chain, was eine Cross-Compiler Tool Chain?

- 1) Native-Compiler Tool Chain:

Erzeugt Programme für die gleiche Architektur/Umgebung in welcher die Tool-Chain ausgeführt wird.

- 2) Cross-Compiler Tool Chain:

Erzeugt Programme für eine andere Architektur/Umgebung als jene in welcher die Tool-Chain ausgeführt wird.