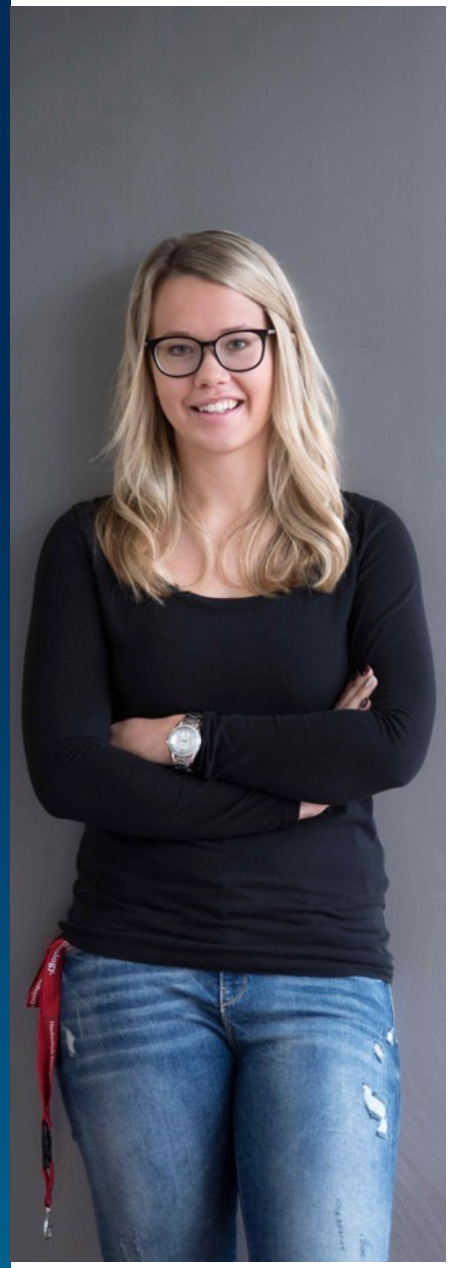
HOCHSCHULE
ESSLINGEN



LECTURE COMPUTER ARCHITECTURE

PERIPHERALS, DIGITAL/ ANALOG CONVERSION AND TIMED PERIPHERALS

RAINER KELLER

HOCHSCHULE
ESSLINGEN

CONTENT

HOCHSCHULE ESSLINGEN

Misc. Interfaces:

- Serial Interface / Serial Peripheral Interface (SPI)
- I^2C Bus
- CAN Bus

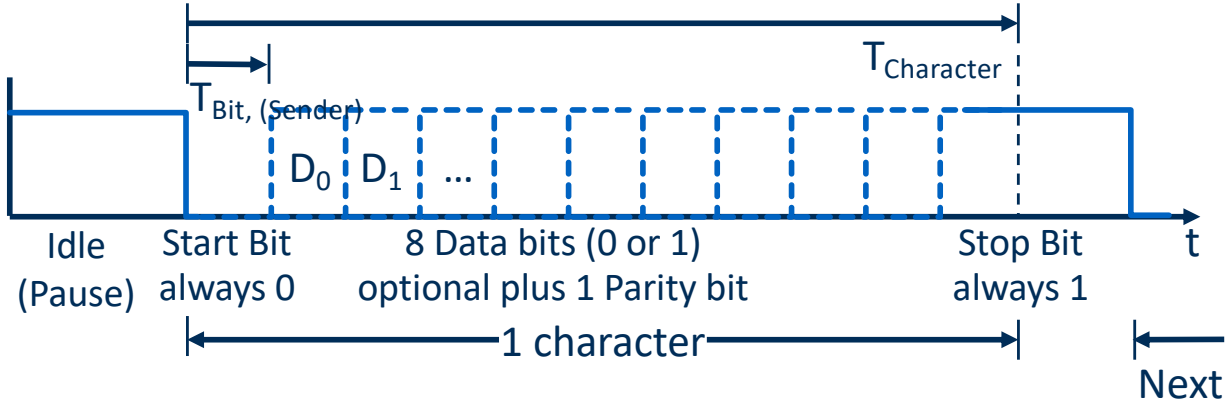
SERIAL INTERFACE 1/5

Simple full-duplex communication: (see SCI `S12SCIV2.pdf`)

← 3 wires with SW-Handshake

→ +2 wires for HW-Handshake
(CTS and RTR/RTS line)

- Data transfer is byte-wise ("**Character**"). The communication line is at 1 when idle. Each character begins with a **Start Bit (S)**, which is always value 0.
- A character typically consists of **8 Data Bits (D)**, beginning with the LSB.
- **Optionally**, followed by a **Parity Bit (P)** (even/uneven parity of 8 data bits)
- The transmission ends with a **Stop Bit** (always value 1). This level remains constant until the next transmission starts (transmission line idle)
- Several standards: **RS-232**, RS-485, RS-422, Typical conventions for D/P/S are 8/N/1.



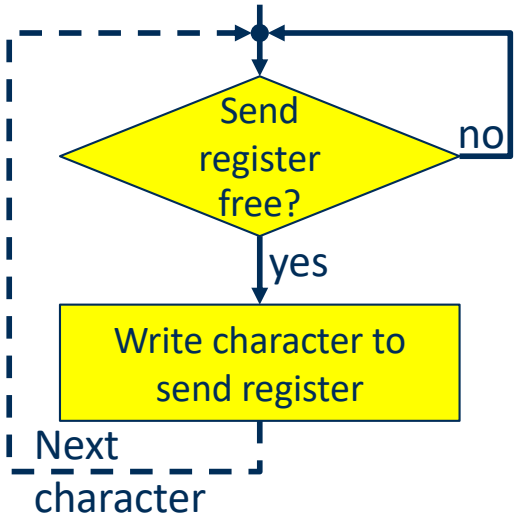
Baud rate = Bit rate:

$$f_{Bit} = \frac{1}{T_{Bit}}$$

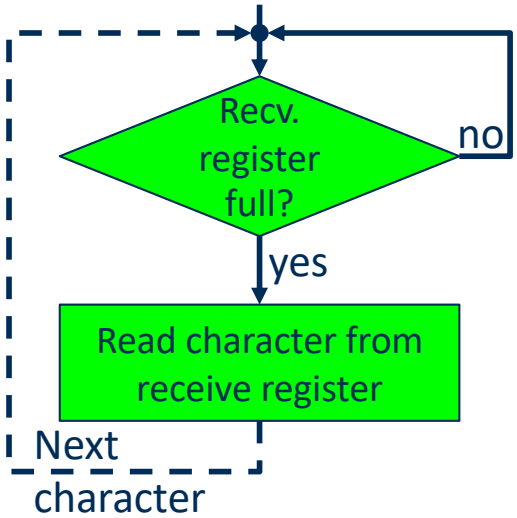
Standardized bit rates are:
9600, 19200, 38400 bit/s
up to 115. 2 kbit/s

Note: Non-standardized faster bit rates are possible.

- Sender and receiver must use the same bit rate and the same character format, e.g. 8N1. Sender and receiver do not have a common clock signal. Due to slightly different clock frequencies (asynchronous data transmission), only small bit rates are possible.
- The HW unit, which transmits single character automatically is called a **U**niversal **A**synchronous **R**eceiver and **T**ransmitter (**UART**) or **S**erial **C**ommunication **I**nterface (SCI). If SW writes a character into the transmit register of the UART, start, data and stop bits are shifted out to the receiver by the HW. When a character has been received, SW must read it from the UART's receive register. The code structure in **Polling Mode** is as follows:



Send character

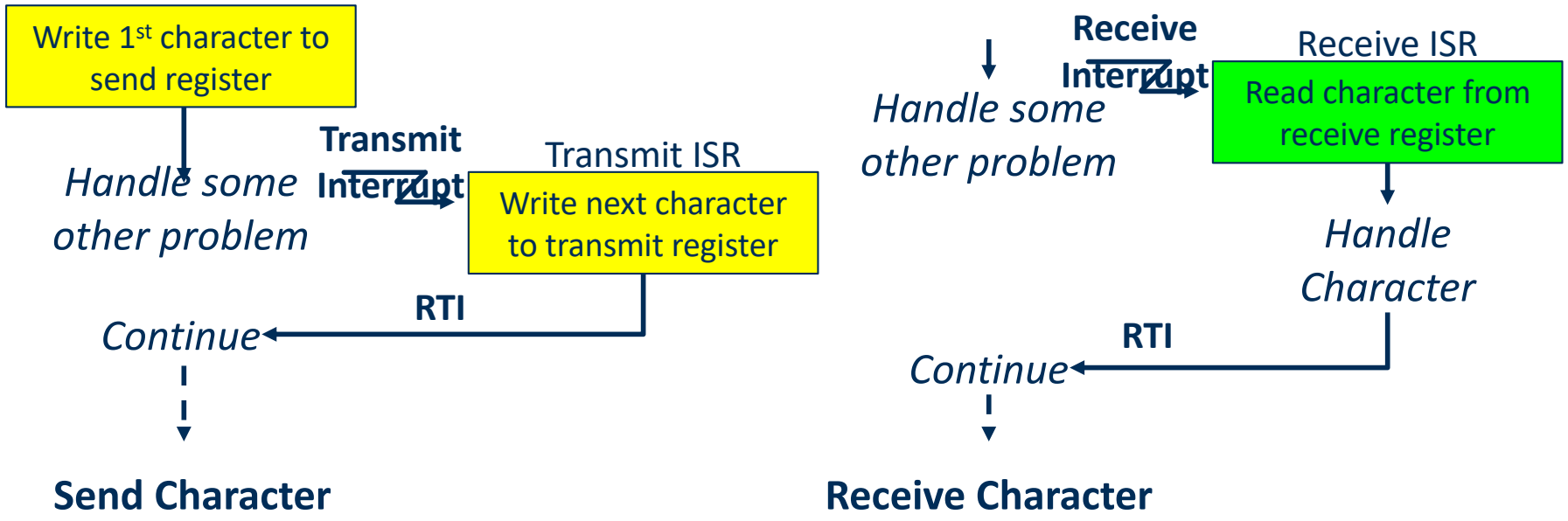


Receive character

SERIAL INTERFACE 3/5

HOCHSCHULE ESSLINGEN

- Before writing a character into the transmission register, the SW must make sure, that it does not overwrite the previous character. When reading a character from the receive register, the SW must make sure, that it does not read a character which it already read before. Rather than polling the status register, an interrupt may be used. The interrupt does signal, when a character has been received (**Receive Interrupt**, used by most programs, because there may be long pauses between character) and/or that the transmission register is free again, when a character has been sent (**Transmission Interrupt**, may or may not be used).



SERIAL INTERFACE 4/5

The HCS12 of the Dragon12 has two UARTs SCI0 and SCI1. SCI0 is used for the debugger communication. SCI1 is free for user programs, e.g. to communicate with a terminal (Hyperterminal or terminal-component of the HCS12 debugger) running on a PC.

Each serial interface has 3 configuration registers, which must be configured once:

• Baud Rate Register (x=0 for SCI0,…)	SCIxBD (16 bit register!)	Clock divider: $SCIxBD = \frac{f_{BUSCLK}}{16 * f_{bit}}$ (Dragon12 @ f_{BUSCLK} = 24 MHz)
• Control Register 1	SCIxCR1 (8 bit register)	Default after reset: 8N1 (8 data bits, no parity, 1 stop bit) If parity is required: Bit 0 = 1 Odd Parity (0 = even Parity) Bit 1 = 1 Use parity bit Bit 7..2 Do not change
• Control Register 2	SCIxCR2	Sender and receiver control Bit 2 = 1 Receiver Enable Bit 3 = 1 Transmitter (Sender) Enable Bit 5 = 1 Receive Interrupt Enable Bit 7 = 1 Transmit Interrupt Enable Set other bits to zero Send and receive interrupts use the same interrupt vector. If both types of interrupts are enable, the ISR must poll status register SCIxSR1 to figure out, why the interrupt was triggered.

SERIAL INTERFACE 5/5

Sending and receiving uses the same data register:

- Data Register

SCIxDRL (8 bit register)	If written:	TX data register (for transmission)
	If read:	RX data register (for reception)
SCIxDRH (8 bit register)	Higher 8 bits of the data register. Used only if the SCI is configured for character length > 8 bit.	

Polling the status of the serial interface:

- Status Register 1

SCIxSR1 (8 bit register)	Bit 7 = 1	TX data register free
	Bit 5 = 1	RX data register full = new character
	The other status bits indicate various error conditions, e.g.	
	Bit 3 = 1	Receive register overwritten by next character before the previous character was read.
	Bit 0 = 1	Parity error

- Status Register 2

SCIxSR2 (8 bit register)	Optional for operating modes not described here.

Due to implementation details of the hardware, the status register SCIxSR1 should always be read, before reading or writing characters from/to the data register. This will automatically clear the status flags for polling and interrupt mode.

SERIAL INTERFACE – EXAMPLE 1/2

HOCHSCHULE ESSLINGEN

Example: Sending and receiving in Polling Mode (see `SerialPolling.mcp`)

```
SCIxBD:  EQU SCI1BD   ; On Dragon12 use SCI1, in the simulator use SCI0
SCIxCR1: EQU SCI1CR1  ; here, too
SCIxCR2: EQU SCI1CR2  ; here, too
SCIxDRL: EQU SCI1DRL  ; here, too
SCIxSR1: EQU SCI1SR1  ; here, too
CR:      EQU $13
LF:      EQU $10
…
.const: SECTION       ; ROM: Constant data
message1: DC.B   "Please enter a character",CR, LF, 0
…
.init: SECTION        ; ROM: Code section
main:
    …                 ; Initialize the serial interface SCIx (0 or 1)
    MOVW #13, SCIxBD  ; Set baud rate to 115200 bit/sec
    MOVB #0, SCIxCR1  ; Default format: 8 data bits, no parity, 1 stop bit
    MOVB #$0C, SCIxCR2; Enable receiver and transmitter, no interrupts
    LDX #message1     ; Send String to SCIx
    JSR puts          ;
    JSR getch         ; Get Character from SCIx, returns character in B
    JSR putch         ; Send Character in B to SCIx
    …
```

Sending and receiving is handled in subroutines `puts`, `getch` & `putch`

SERIAL INTERFACE – EXAMPLE 2/2

HOCHSCHULE ESSLINGEN

```
getch:   ; --- Read a character from serial interface and put in B ----------
     BRCLR SCIxSR1, #$20, getch     ; Check 'Receive Data Flag' and loop
     LDAB SCIxDRL                   ; Read received character
     RTS                            ; … and return it's value in B

putch:   ; --- Send a character in B to serial interface --------------------
     BRCLR SCIxSR1, #$80, putch     ; Check 'Transmit Register Empty' and loop
     STAB SCIxDRL                   ; Send character by writing into data reg
     RTS                            ; and return

puts:    ; --- Send string to serial interface, X is a pointer to string -----
     BRCLR SCIxSR1, #$80, puts      ; Check 'Transmit Register Empty' and loop
     MOVB 0, X, SCIxDRL             ; Send one character
     TST 1, X+                      ; Check for end of string (NUL byte)
     BNE puts                       ; If not, go to next character
     RTS                            ; send complete, return
```

A program version using interrupts rather than polling can be found in project `serialInterrupt.mcp`

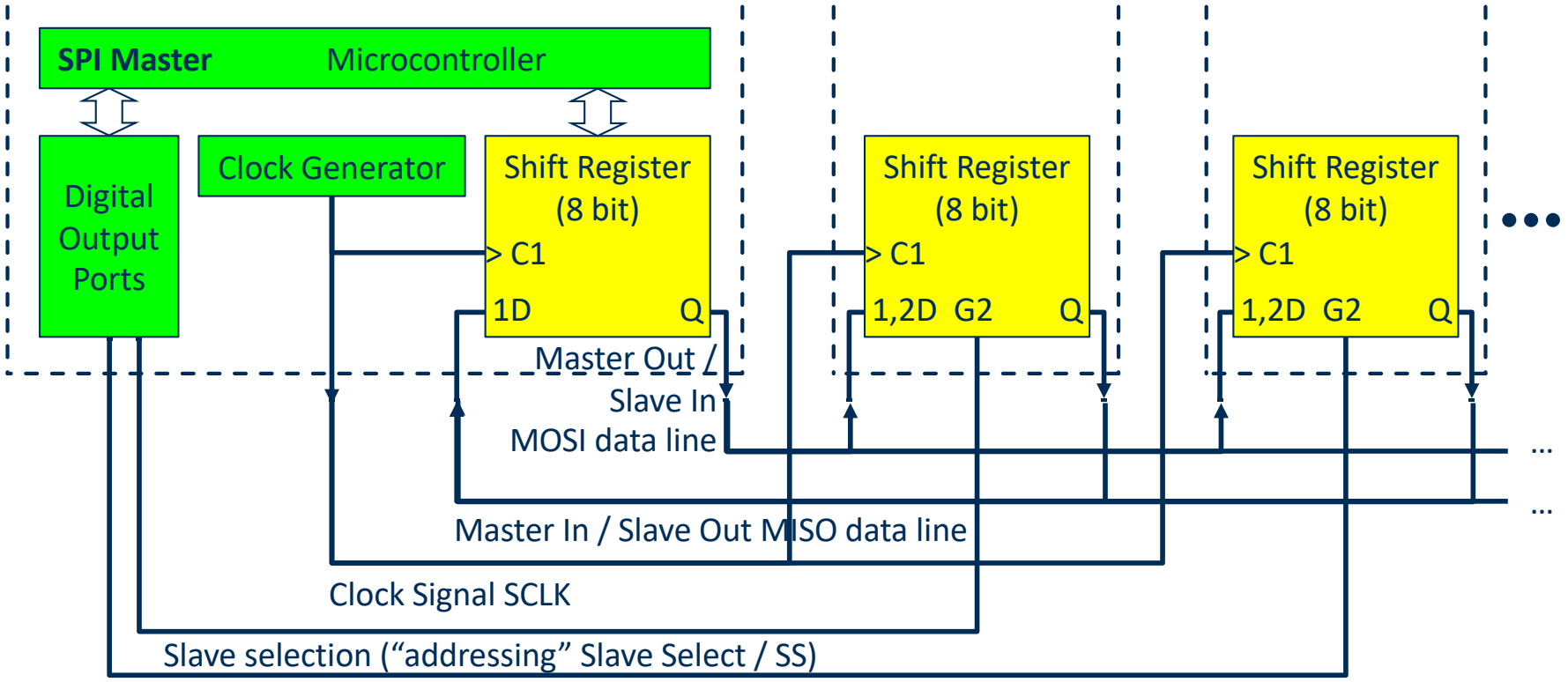
Note:

A user program must use the second serial interface SCI1 on Dragon12, since the first interface SCI0 is used by the debugger monitoring program. In the simulator, user programs must use SCI0, because the simulator does not support SCI1.

To enable SCI1 on new Dragon12 boards, jumper J23 must be set to position RS232 (already done in our Lab, but on new boards the factor preset is SCI1 disabled.

SERIAL PERIPHERAL INTERFACE (SPI)

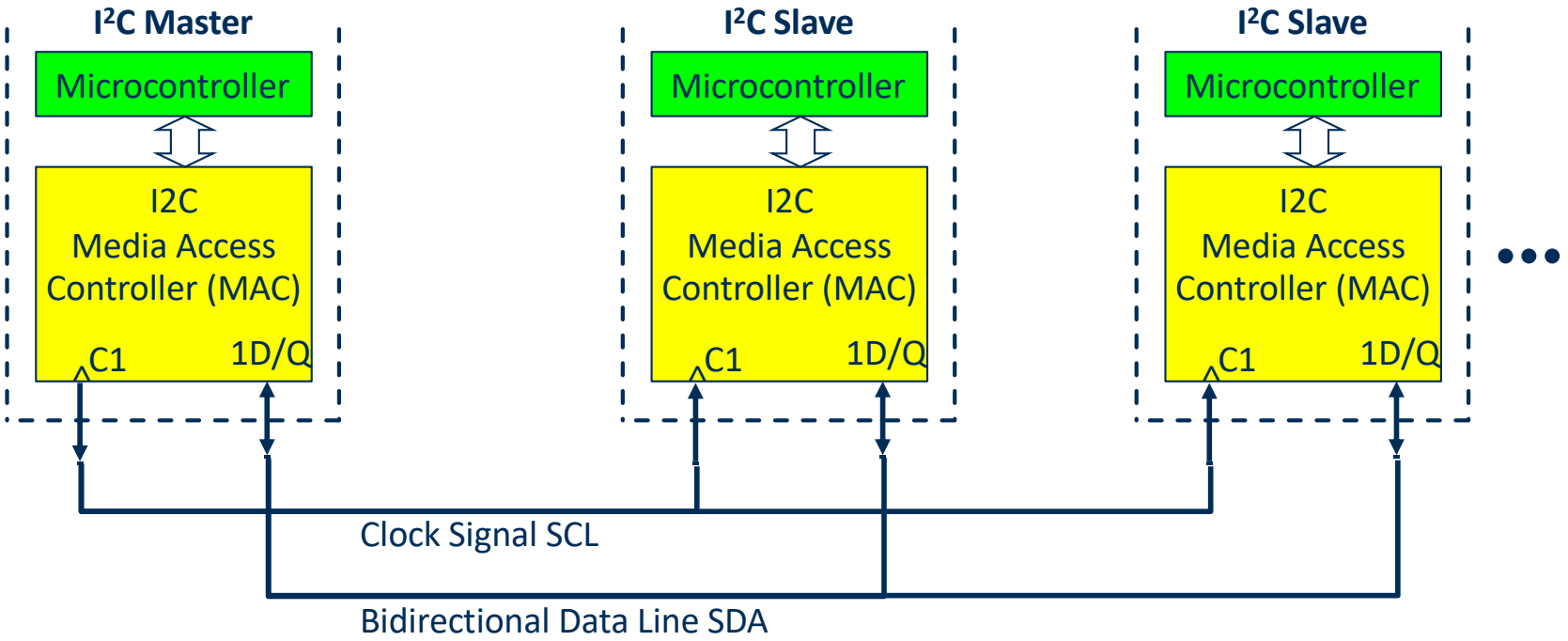
- Serial interface between a microcontroller and peripheral ICs on a printed circuit board PCB
- Industry standard to connect serial EEPROMs or external A/D- and D/A-converter ICs
- Basic principle: “Ring of shift registers”, clocked by **SPI Master** with up to 4 Mbit/s
- Byte wise data transmission clocked by master (1 byte Master → Slave, at the same time 1 byte Slave → Master), duplex communication, but **SPI Slaves** cannot start a transmission, no acknowledge or error signaling from receiver to sender
- Master selects one slave per transmission via a separate “address line” per slave



INTER IC BUS (I^2C)

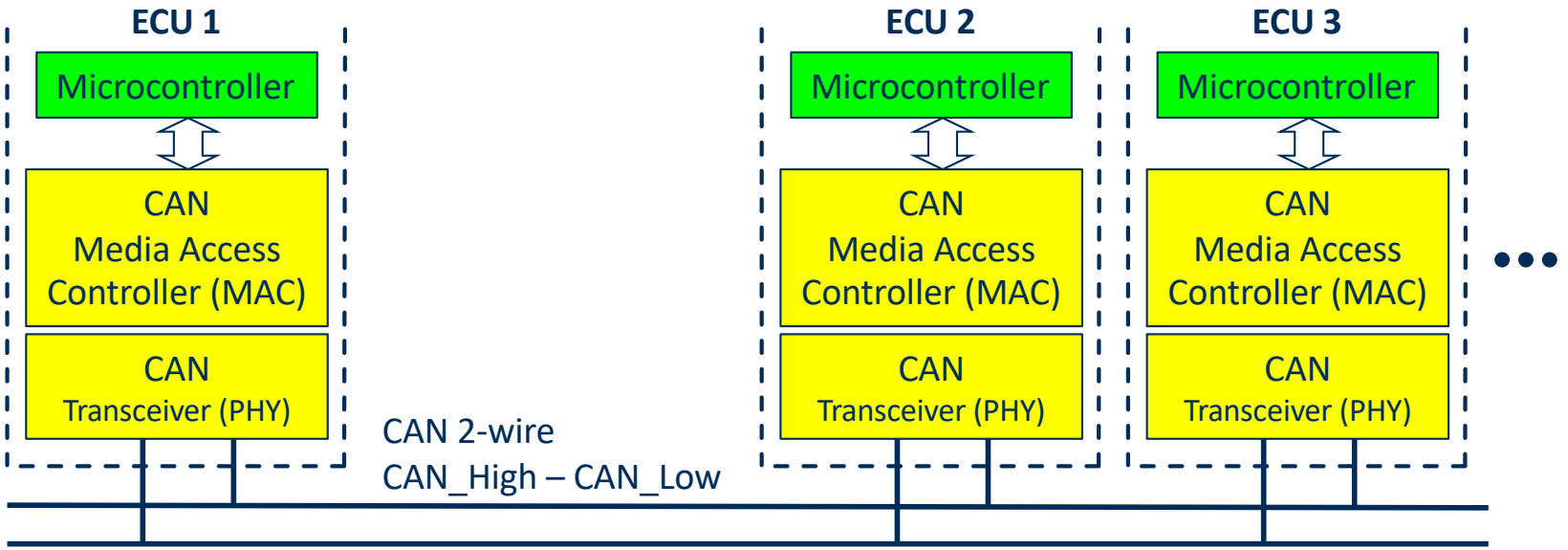
Serial interface between microcontroller and peripheral ICs on a printed circuit board PCB

- Industry standard (Philips/NXP & others), originated in consumer electronics, e.g. TV
- Bidirectional Master-Slave system, clocked by Master with up to 400 kbit/s. Limited multimaster capability.
- Messages with 7bit-address header to select one of the slaves and an arbitrary number of data bytes. If data shall be sent from one of the slaves to the master or to other slaves, the Master sends the slave's address. Then the Master stops (but still outputs a clock signal) and the Slave continues with sending the data bytes.
- Each received byte is confirmed by the receiver with an Acknowledge Bit.



CONTROLLER AREA NETWORK 1/4

- Serial interface of automotive & automation Electronic Control Units (ECUs). Dev. by Bosch, industry standard ISO 11898, licensed to most microcontroller suppliers
- Multi-master bus with **bit rates up to 1 Mbit/s**, in cars typically f_{bit}=500 kBit/s (High-Speed CAN) and f_{bit}=100 kBit/s (Low-Speed CAN) with different physical layers, uses two wires with differential voltage signals. Due to arbitration (see below) the max. bus length L limited by bit rate f_{bit}. At L ≅ 40m → 1 Mbit/s, shorter lengths up to x20!



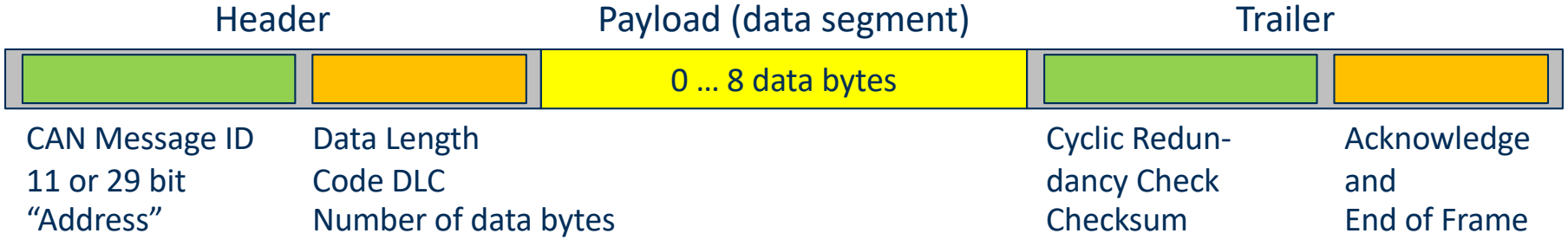
- Messages with 11 bit (or 29 bit) addresses (**CAN Identifier**), up to 8 data byte and CRC check sum. Receiver acknowledges receipt, automatic retransmissions upon errors.
- **Content based** rather than station based **addressing:** Receiving ECUs are selecting messages based on content. This acceptance filtering via the CAN Identifier is done by hardware in CAN communication controllers. (Note: Ethernet MAC addressing is station based)

CONTROLLER AREA NETWORK 2/4

HOCHSCHULE ESSLINGEN

- The CAN Identifier is additionally used for message **arbitration**, i.e. if two ECUs start sending at the same time, a bus collision will be detected (because the electrical signal on the bus will be corrupted). CAN controllers will detect the collision and the message with the smaller valued **CAN identifier** (=higher **priority**) will continue, while the other message will be stopped and resent later, when the bus is idle again. Carrier Sense Multiple Access/Collision Resolution (CSMA/CR)

Format of a CAN message:



(Various control bits in the header and trailer not shown here, message format automatically generated by the HW)

Simple HCS12 CAN driver (CAN bitrate 250 kbit/s, 11 bit CAN identifier):

```
typedef struct {                                // Data structure for CAN message
    unsigned long canID;                        // 11-/29-bit CAN identifier
    unsigned char dataLength;                   // Data length (max. 8 bytes)
    unsigned char data[8];                      // The actual data
} CANMESSAGE;
CANMESSAGE sendMessage, receiveMessage;         // Variables to send message
void CanInit(void);                             // Function calls by user app
void CanSendMessage(CANMESSAGE * pMsg);
void CanRecvMessage(CANMESSAGE * pMsg);
```

CONTROLLER AREA NETWORK 3/4

HOCHSCHULE ESSLINGEN

```
void CanInit(void){           // CAN controller initialization
  CAN0CTL0 = CAN0CTL0 | 0x01; // Request initialization
  while(!(CAN0CTL1 & 0x1)){}; // Wait for init start acknowledge

  CAN0CTL1  = 0x80;           // CAN enable, use OSCCLK (Quartz-Takt)
  CAN0BTR0  = 0xC1;           // 250 kbit/s@fOSCCLK=8MHz, 0xC0 @ fOSCCLK=4MHz
  CAN0BTR1  = 0x58;           // Receive CAN messages with any CAN id
  CAN0IDAC  = 0x20;
  CAN0IDAR0 = 0x00;
  CAN0IDMR0 = 0xFF;
  CAN0CTL0 = CAN0CTL0 & 0xFE; // End initialization
  while ((CAN0CTL1 & 0x1)){}; // Wait for end of init acknowledge

  CAN0RFLG_RXF = 1;           // Clear receiver flags
  CAN0RIER = CAN0RIER | 0x01; // Receive interrupt enable
}
void CanSendMessage(CANMESSAGE *pMsg){// Send a CAN message (typ. pMsg = &sendMessage)
  volatile unsigned char txBuf;
  while (!(CAN0TFLG&0x7)){};  // Wait for a free transmit buffer

  txBuf = CAN0TFLG & 0x7;     // Select the transmit buffer
  CAN0TBSEL = txBuf;
  txBuf = CAN0TBSEL & 0x7;
  CAN0TXIDR0 = (unsigned char)(pMsg->CanId>>3);// Copy message to transmit buffer
  CAN0TXIDR1 = (unsigned char)(pMsg->CanId<<5);// -- here: CAN identifier CAN0TXIDR2=0x00;
  CAN0TXIDR3 = 0x00;
  memcpy(&CAN0TXDSR0, pMsg->Data, pMsg->DataLength);// -- here: CAN data
  CAN0TXDLR = pMsg->DataLength;// Set data length (number of bytes in message, max.8)
  CAN0TXTBPR = CAN0TXIDR0;
  CAN0TFLG = txBuf;           // Send the message
}
```

CONTROLLER AREA NETWORK 4/4

HOCHSCHULE ESSLINGEN

```
void CanReadMessage(CANMESSAGE *pMsg) { // Receive a CAN msg (polling or used in ISR)
  unsigned long temp;                   // (typ. pMsg = &receiveMessage)

  if (!(CAN0RFLG & 0x01)) {             // Return at once, if the receive buffer is empty
    pMsg->DataLength = 0;
    return;
  }
  pMsg->CanId = CAN0RXIDR0;             // Copy CAN identifier to CAN data structure
  pMsg->CanId = (pMsg->CanId << 3) + (CAN0RXIDR1 >> 5);

  // Copy CAN data bytes to CAN data structure
  memcpy(pMsg->Data, &CAN0RXDSR0, CAN0RXDLR);
  pMsg->DataLength = CAN0RXDLR;         // Copy CAN data length

  CAN0RFLG = CAN0RFLG | 0x01;           // Clear receive flag
}

void interrupt 38 CanReceiveISR(void) {
  CanReadMessage(&receiveMessage); // Copy CAN message to global variable

  // Process received data or notify user program by setting an event variable

  CAN0RFLG = CAN0RFLG | 0x01;           // Clear receive flag
}
```

APPENDIX B: CLOCK GENERATOR AND CLOCK DIVIDER

HOCHSCHULE ESSLINGEN

RTI Interrupt:

Interrupt period $T_{RTI} = 2^{9+X} * (Y + 1) / f_{OSCCLK}$. All values in ms @ f_{OSCCLK} = 4 MHz

	Y=0	Y=1	Y=2	Y=3	Y=4	Y=5	Y=6	Y=7	Y=8	Y=9	Y=10	Y=11	Y=12	Y=13	Y=14	Y=15
X=1:	0.256	0.512	0.768	1.024	1.280	1.536	1.792	2.048	2.304	2.560	2.816	3.072	3.324	3.584	3.840	4.096
X=2:	0.512	1.024	1.536	2.048	2.560	3.072	3.584	4.096	4.608	5.120	5.632	6.144	6.656	7.168	7.680	8.192
X=3:	1.024	2.048	3.072	4.096	5.120	6.144	7.168	8.192	9.216	10.240	11.264	12.288	13.312	14.336	15.360	16.384
X=4:	2.048	4.096	6.144	8.192	10.240	12.288	14.336	16.384	18.432	20.480	22.528	24.576	26.624	28.672	30.720	32.768
X=5:	4.096	8.192	12.288	16.384	20.480	24.576	28.672	32.768	36.864	40.960	45.056	49.152	53.248	57.344	51.440	65.536
X=6:	8.192	16.384	24.576	32.768	40.960	49.152	57.344	65.536	73.728	81.920	90.112	98.304	106.496	114.688	122.880	131.072
X=7:	16.384	32.768	49.152	65.356	81.920	98.304	114.68	131.072	147.456	163.840	180.224	196.608	212.992	229.376	245.760	262.144

ECT Timer: @ f_{BUSCLK} = 24 MHz

Clock period $T_{TCNT} = 1/f_{TCNT} = 2^X/f_{BUSCLK}$. In µs

X=0	X=1	X=2	X=3	X=4	X=5	X=6	X=7
0.042	0.083	0.167	0.333	0.667	1.333	2.667	5.333

Clock period $T_P = 2^{16} * TTC_{NT}$ Values in ms

X=0	X=1	X=2	X=3	X=4	X=5	X=6	X=7
2.731	5.461	10.293	21.845	43.691	87.381	174.763	349.525

PWM: @ f_{BUSCLK} = 24 MHz

Clock period of fast clock $T_{A/B} = 2^X/f_{BUSCLK}$. In µs

X=0	X=1	X=2	X=3	X=4	X=5	X=6	X=7
0.042	0.083	0.167	0.333	0.667	1.333	2.667	5.333

Clock period of slow clock $T_{SA/B} = 2*y*T_{A/B}$. In µs

	X=0	X=1	X=2	X=3	X=4	X=5	X=6	X=7
Y=1	0.083	0.167	0.333	0.667	1.333	2.667	5.333	10.667
…	…	…	…					
Y=255	21.250	42.500	85.000	170000	340000	680000	1360000	2720000

Serial Interface: @ f_{BUSCLK} = 24 MHz

Clock Divider Register SCIxBD = $f_{BUSCLK}/(16*f_{BIT})$

f_{BIT}=300 bit/s	600 bit/s	1,2 kbit/s	2,4kbit/s	4,8kbit/s	9,6kbit/s	19,2kbits	38,4kbit/s	57,6kbit/s	115,2kbit/s
5000	2500	1250	625	313	156	78	39	26	13

LECTURE: LITERATURE

HOCHSCHULE ESSLINGEN

According to list of literature:

- Patterson, D., Hennessy, J.: *Computer Organization and Design*, Kaufmann, 2011
  (Deutsche Übersetzung: Rechnerorganisation und –entwurf, Spektrum)
- Hennessy, J., Patterson, D.: *Computer Architecture: A quantitative Approach, **5th edition***, Morgan Kaufmann, 2017
- Tanenbaum, A., Austin, T.: *Structured Computer Organization*, Pearson, 6th edition, 2013
- Tanenbaum, A., Austin, T.: *Rechnerarchitektur: von der digitalen Logik zum Parallelrechner*, Pearson, 6th ed., 2014
- Huang, H.W.: *The HCS12/9S12. An Introduction to the HW and SW interface*, Thomson Learning, 2009
- Beierlein, T.: *Taschenbuch Mikroprozessortechnik*, Carl-Hanser Verl., 2011