Embedded Systems

C language in Embedded Systems, direct access to hardware resources
LAB 01

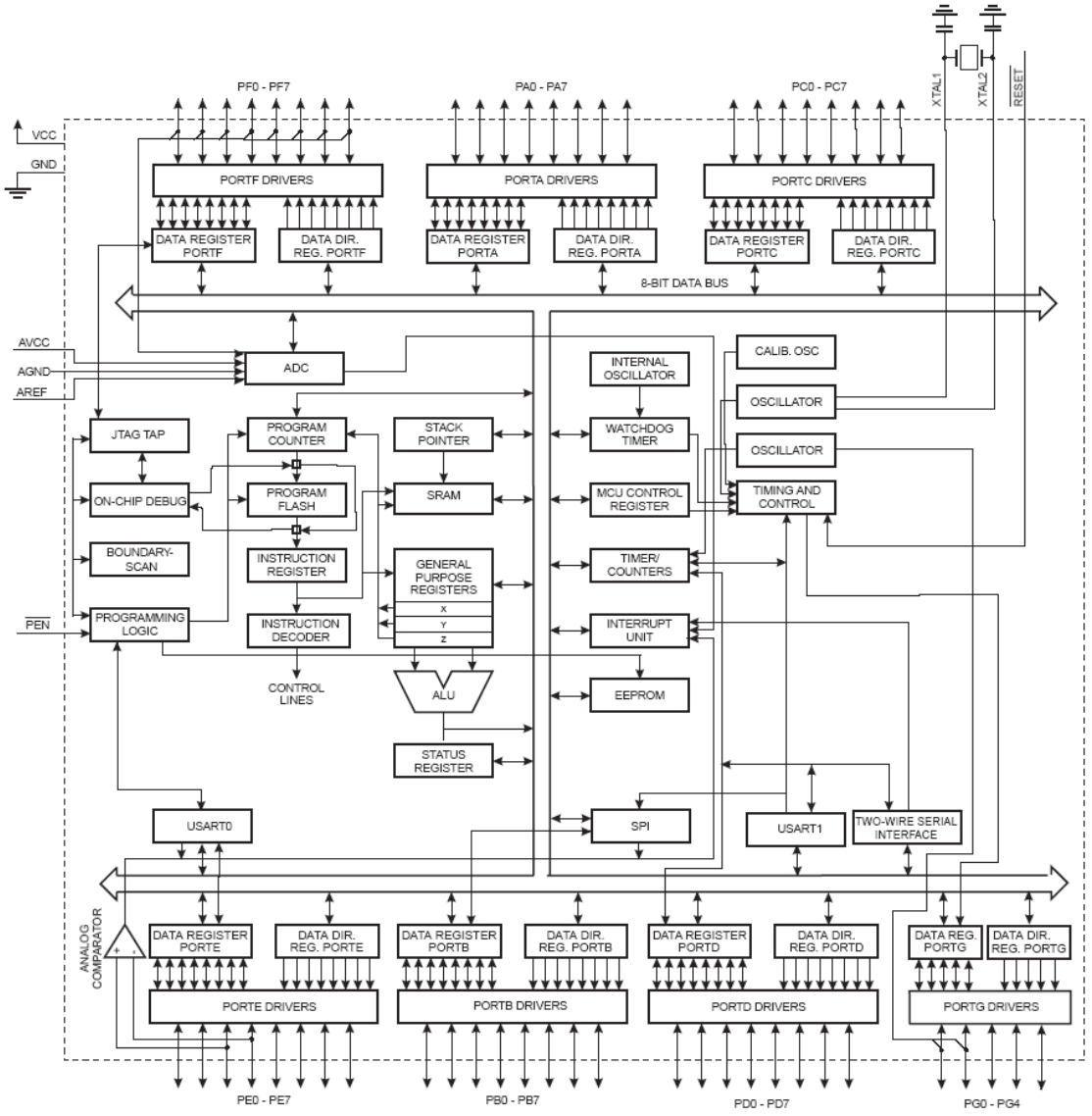
Francesco Menichelli

francesco.menichelli@uniroma1.it

Outline

- Programming embedded systems in C
- Memory mapped I/O
- Reading and Writing I/O registers in assembly
- Reading and Writing I/O registers in C
- Application: USART device in Atmega family
- Serial asynchronous communications

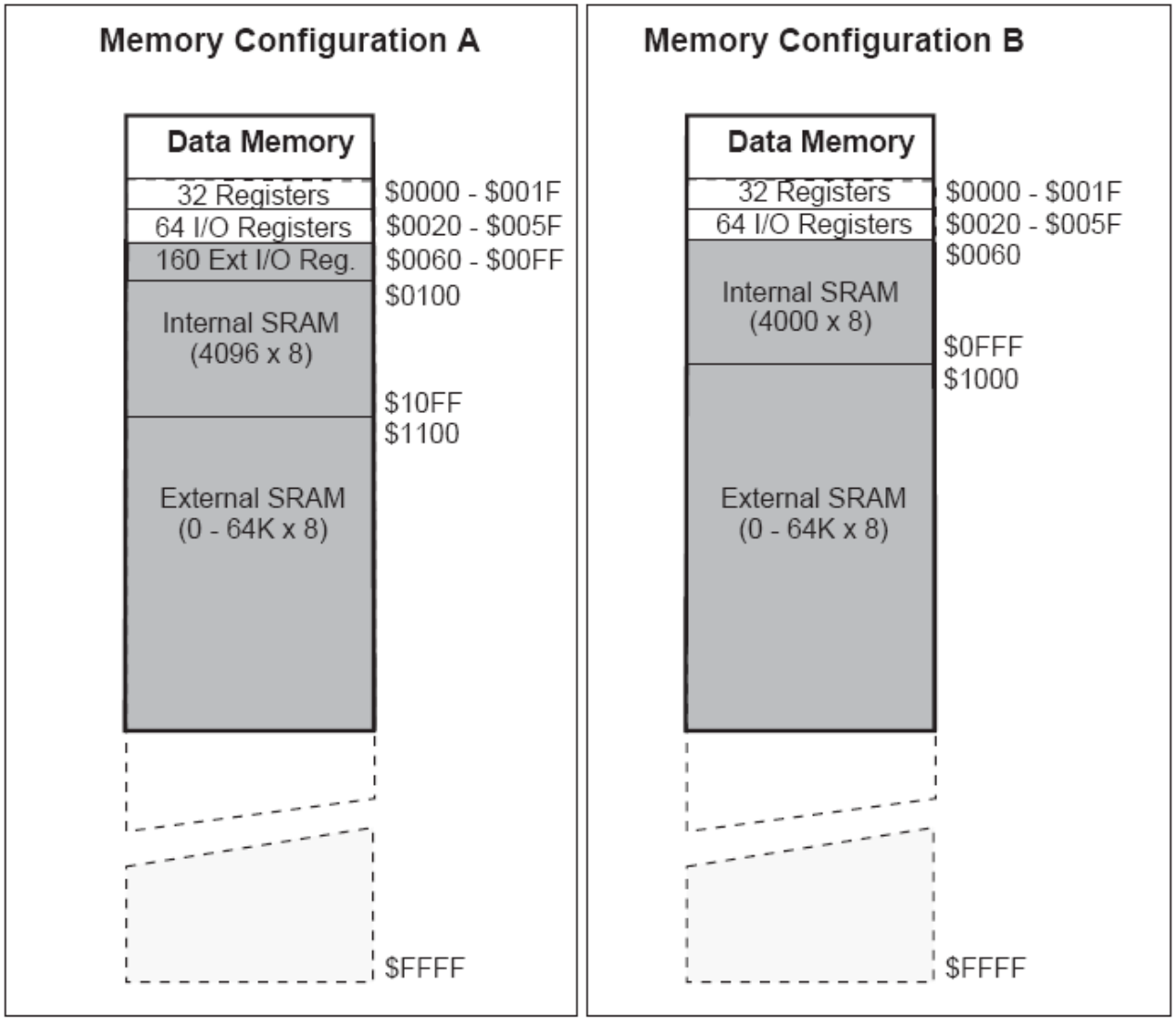
ATmega128 – Block diagram



Memory mapped I/O

- Memory mapped resources
- Each device is configured/used by accessing its set of registers
  - Can be distinguished in control and status registers
  - Multiple devices can be present, have the same register set, but at different base address
- I/O registers can be
  - Read/Write
  - Read only
  - Write only
- The only interaction with peripherals is provided by load/store instruction
  - Exception: interrupts

ATmega128 Memory Map



ATmega128 Memory Map

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
($FF)	Reserved	–	–	–	–	–	–	–	–	
..	Reserved	–	–	–	–	–	–	–	–	
($9E)	Reserved	–	–	–	–	–	–	–	–	
($9D)	UCSR1C	–	UMSEL1	UPM11	UPM10	USBS1	UCSZ11	UCSZ10	UCPOL1	190
($9C)	UDR1	USART1 I/O Data Register								188
($9B)	UCSR1A	RXC1	TXC1	UDRE1	FE1	DOR1	UPE1	U2X1	MPCM1	188
($9A)	UCSR1B	RXCIE1	TXCIE1	UDRIE1	RXEN1	TXEN1	UCSZ12	RXB81	TXB81	189
($99)	UBRR1L	USART1 Baud Rate Register Low								192
($98)	UBRR1H	–	–	–	–	USART1 Baud Rate Register High				192
($97)	Reserved	–	–	–	–	–	–	–	–	
($96)	Reserved	–	–	–	–	–	–	–	–	
($95)	UCSR0C	–	UMSEL0	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0	190
($94)	Reserved	–	–	–	–	–	–	–	–	
($93)	Reserved	–	–	–	–	–	–	–	–	
($92)	Reserved	–	–	–	–	–	–	–	–	
($91)	Reserved	–	–	–	–	–	–	–	–	
($90)	UBRR0H	–	–	–	–	USART0 Baud Rate Register High				192
($8F)	Reserved	–	–	–	–	–	–	–	–	
($8E)	Reserved	–	–	–	–	–	–	–	–	
($8D)	Reserved	–	–	–	–	–	–	–	–	
($8C)	TCCR3C	FOC3A	FOC3B	FOC3C	–	–	–	–	–	135

Register access in assembly (ARM)

- Access to memory mapped registers is realized by means of load/store instructions

```
MOV R1, 0x7050;
MOV R2, 0;

STR R2, [R1]; write 0 to the register
LDR R0, [R1]; read the register to R0
```

- In case of 16bit (8 bit) registers STRH,LDRH (STRB,LDRB) can be used

```
STRH R2, [R1]; write 0 to the register
LDRH R0, [R1]; read the register to R0
```

```
STRB R2, [R1]; write 0 to the register
LDRB R0, [R1]; read the register to R0
```

Register access in C

- Access to memory mapped registers is realized by means of pointers

```
volatile short unsigned int *reg = 0x7050;
short unsigned int a;

*reg = 0x0000; write 0 to the register
a = *reg; read the register to a variable
```

- The compiler can create shortcuts using #define macros
  - Example for AVR8 CPU (ATmega)

```
/* USART0 I/O Data Register */
#define UDR0      _SFR_IO8(0x0C)

#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
```

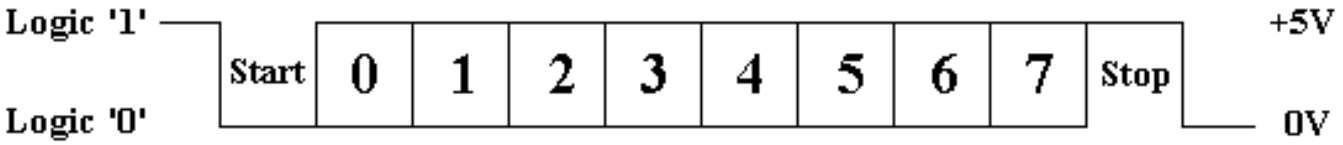
AVR gcc – register constants definition

Microcontroller registres are defined in
Toolchain path:
C:\Program Files (x86)\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1061\avr8-gnu-toolchain\
Include file:
avr\include\avr

```
/* USART0 Control and Status Register A */
#define UCSR0A    _SFR_IO8(0x0B)
/* USART0 I/O Data Register */
#define UDR0      _SFR_IO8(0x0C)
/* SPI Control Register */
#define SPCR      _SFR_IO8(0x0D)
/* SPI Status Register */
#define SPSR      _SFR_IO8(0x0E)
/* SPI I/O Data Register */
#define SPDR      _SFR_IO8(0x0F)
```

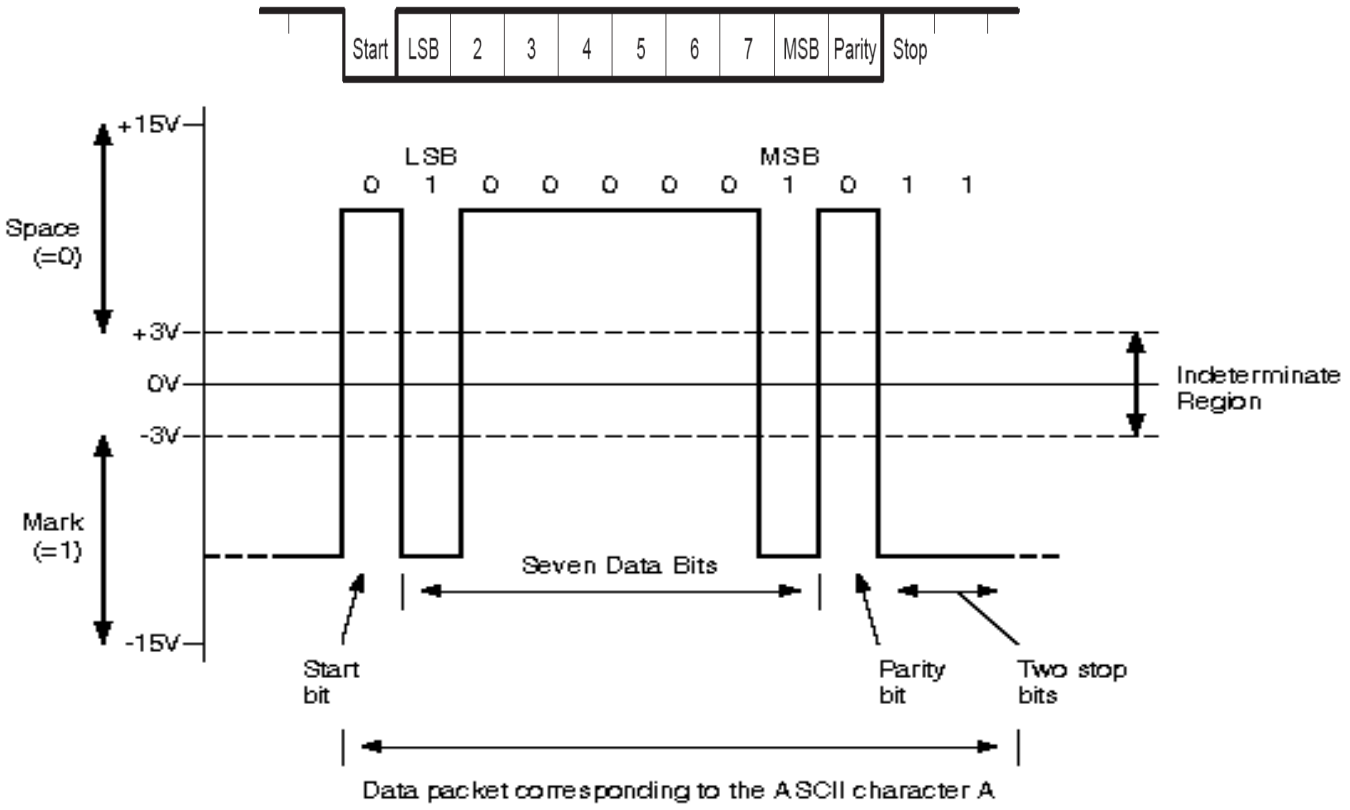
Asynchronous serial communications

- serial
  - a single data line is used for communication
- asynchronous
  - Tx device does not provide an external clock signal
  - Rx device must autonomosly find the sampling time on data signal



Asynchronous serial communications

- Electrical levels for ‘0’ and ‘1’ simbols can vary
  - TTL/CMOS: ‘0’=0V ; ‘1’=Vdd (5V/3.3V)
  - RS232: ‘0’=+12V ; ‘1’=-12V



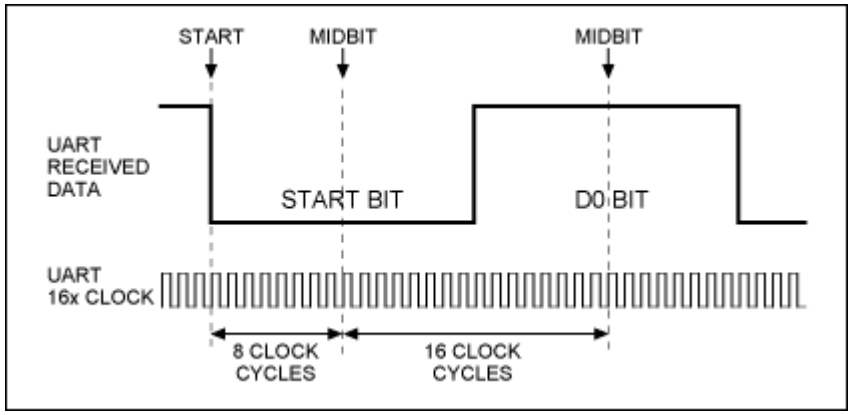
Asynchronous serial communications

On a typical device, software can configure

- Bitrate
- Data bit number
- Parity
- Stop bit number
- Flux control

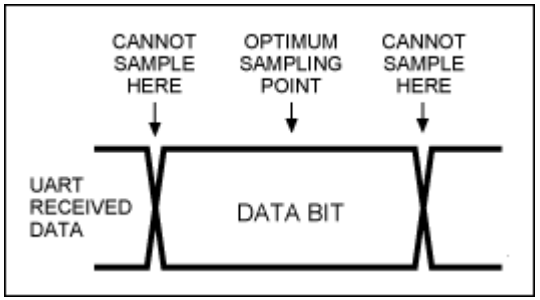
Clock recovery

- The receive UART uses a clock that is a multiple of the data rate (for example 16 times)
- A new frame is recognized by the falling edge at the beginning of the active-low START bit. This occurs when the signal changes from the active-high STOP bit or bus idle condition.
- The receive UART resets its counters on this falling edge, expects the mid-START bit to occur after 8 clock cycles, and anticipates the midpoint of each subsequent bit to appear every 16 clock cycles thereafter.
- The START bit is typically sampled at the middle of bit time to check that the level is still low and ensure that the detected falling edge was a START bit, not a noise spike.
- Another improvement is to sample the START bit three times (clock counts 7, 8, and 9, out of 16) instead of sampling it only at the midbit position (clock count 8 out of 16).

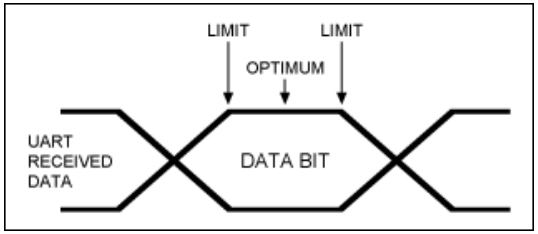


Timing Accuracy

- Since the UART receiver synchronizes itself to the start of each and every frame, we only care about accurate data sampling during one frame.
- There is no buildup of error beyond a frame's STOP bit
- The goal is to sample each bit at the midpoint, with maximum one-half a bit-period too early or too late



- In reality, we cannot sample close to the bit-transition point reliably.



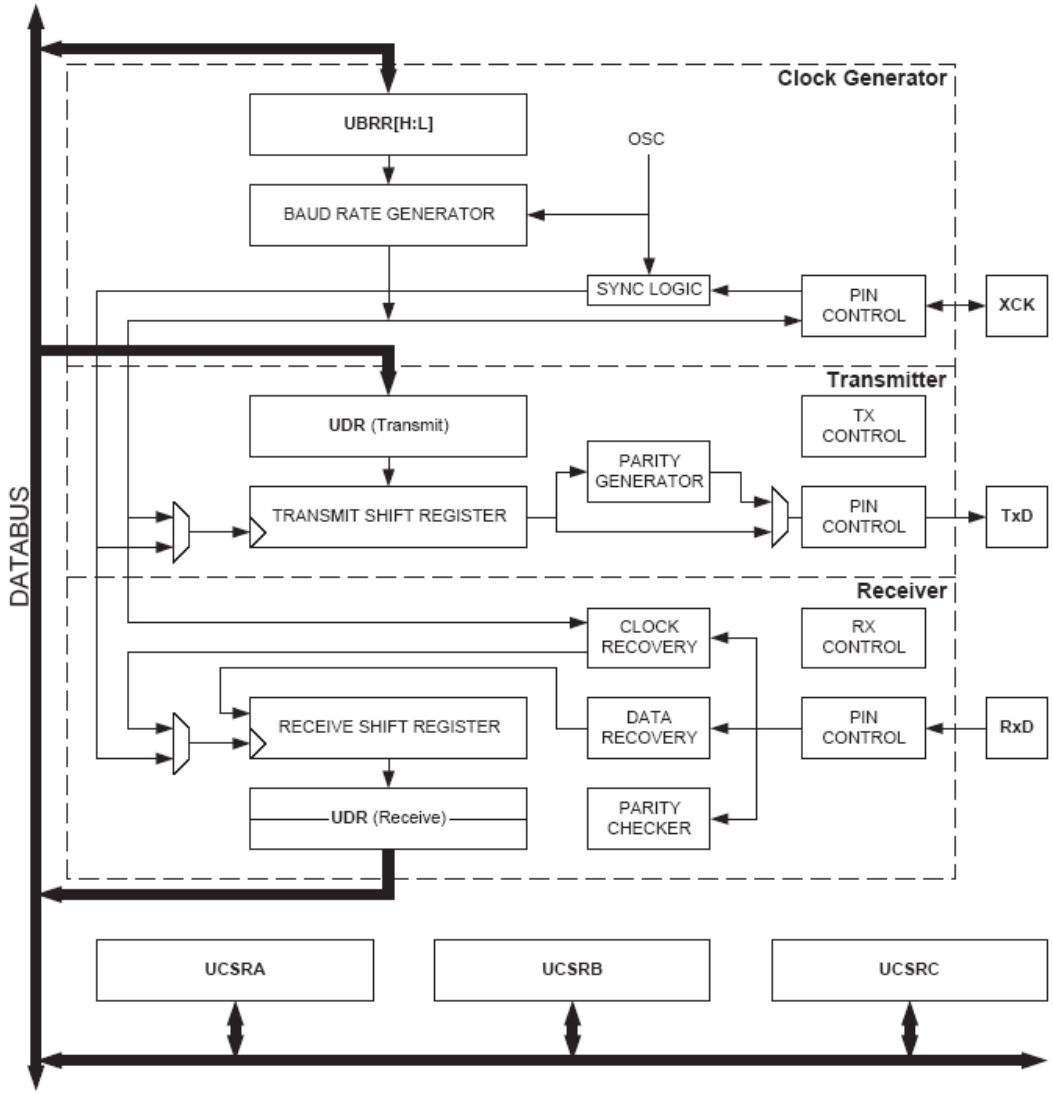
Timing Accuracy

- Supposing 25% rise time + 25% falling time, and referring to the 16x UART receive clock, we have a maximum allowable error of 8 cycles, or ±4 cycles from the center of the bit
- Another error to include in this budget is the synchronization error when the falling edge of the START bit is detected (±1 cycle), hence ±3 cycles from the center of the bit is the maximum allowable error
- Since the timing is synchronized at the falling edge of the START bit, the worst-case timing error will be at the last data sampling point, which is the STOP bit1.
- The optimum sampling point for the STOP bit is its bit center, which is calculated as:

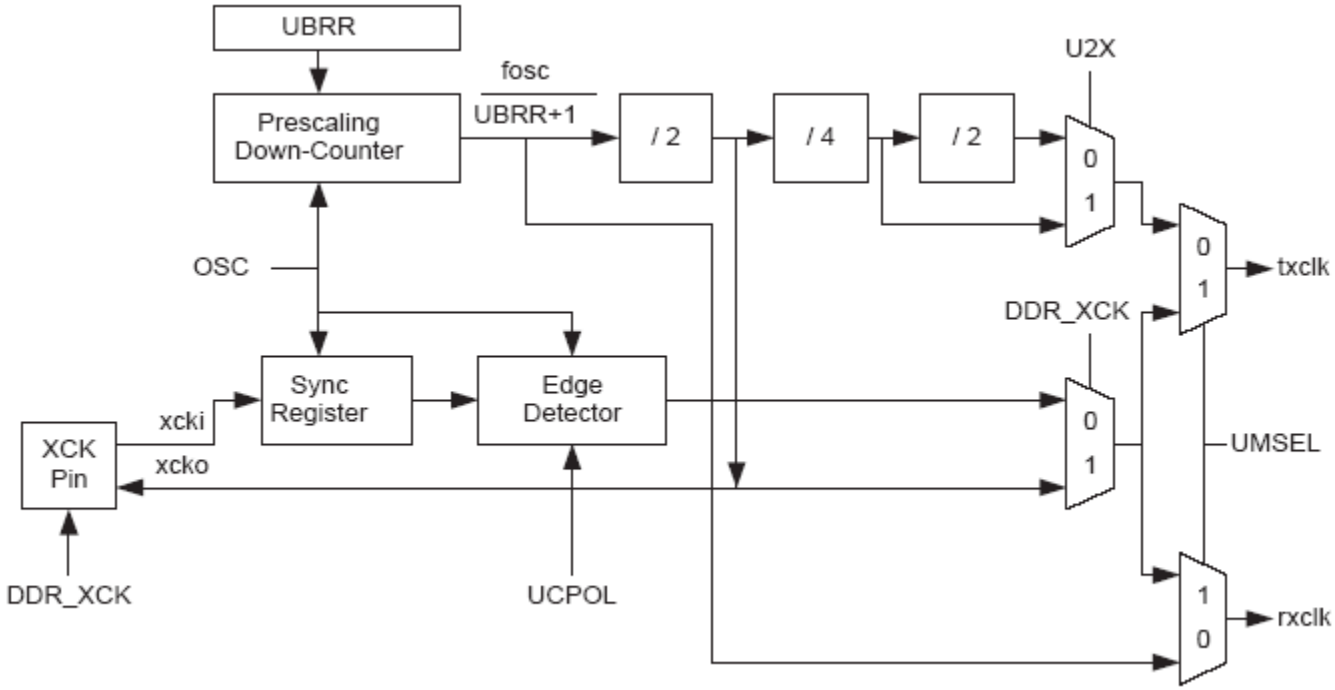
(16 internal clock cycles per bit) × (1 start bit + 8 data bits + ½ a stop bit) = (16) × (9.5) = 152 cycles

- The clock mismatch error can be ±3/152 = ±2%
- Although the problem will materialize at the receive end of the link, clock mismatch is actually a tolerance issue shared between the transmit and receive UARTs. So presuming that both UARTs are attempting to communicate at exactly the same bit rate (baud), the allowable error can be shared, in any proportion, between the two UARTs.

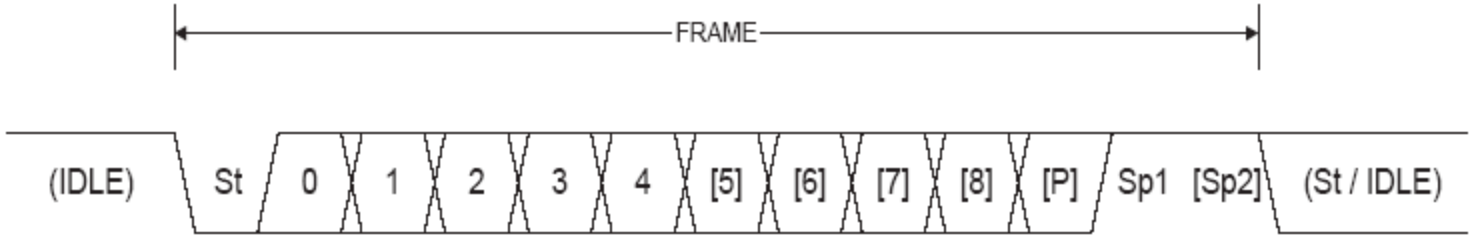
USART device on AVR



Clock generator for USART on AVR



USART Data frame format



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high.

Clock recovering – bit sampling

Figure 83. Start Bit Sampling

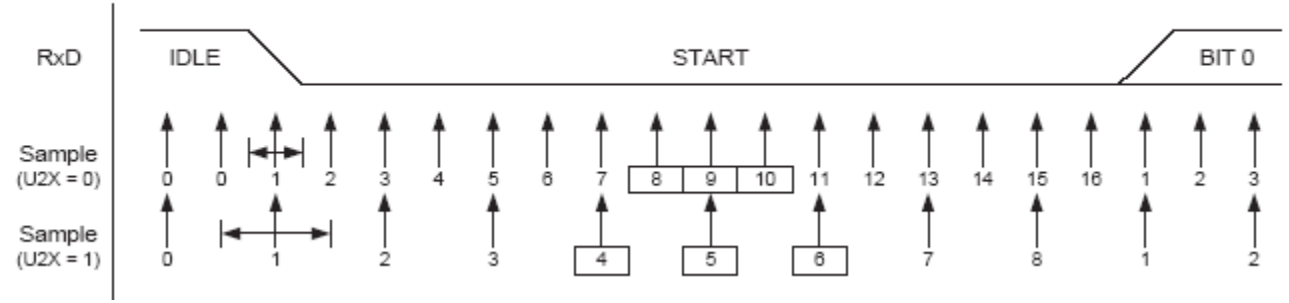


Figure 84. Sampling of Data and Parity Bit

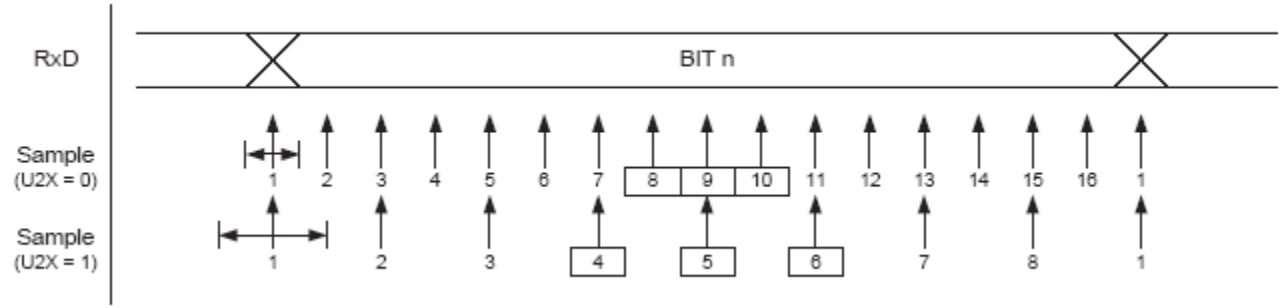
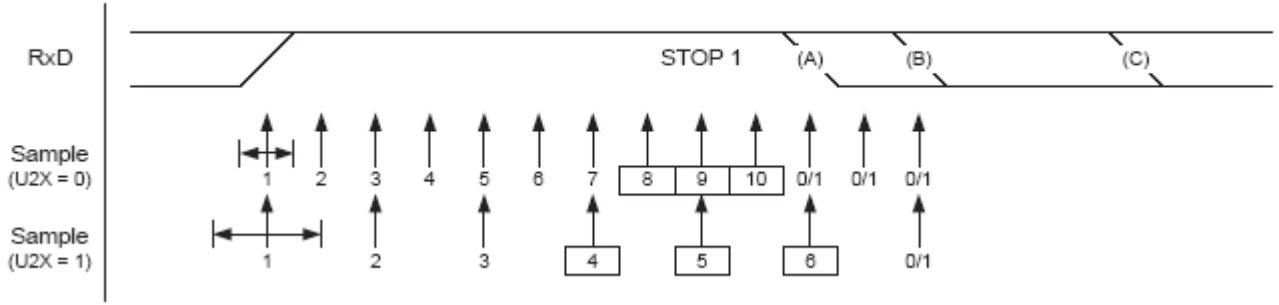


Figure 85. Stop Bit Sampling and Next Start Bit Sampling



USART Baud rate Registrs

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal Mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(\boldsymbol{UBRR}+1)}$	$\boldsymbol{UBRR} = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(\boldsymbol{UBRR}+1)}$	$\boldsymbol{UBRR} = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(\boldsymbol{UBRR}+1)}$	$\boldsymbol{UBRR} = \frac{f_{OSC}}{2BAUD} - 1$

$$\text{Error}[\%] = \left(\frac{\text{BaudRate}_{\text{Closest Match}}}{\text{BaudRate}} - 1\right) \bullet 100\%$$

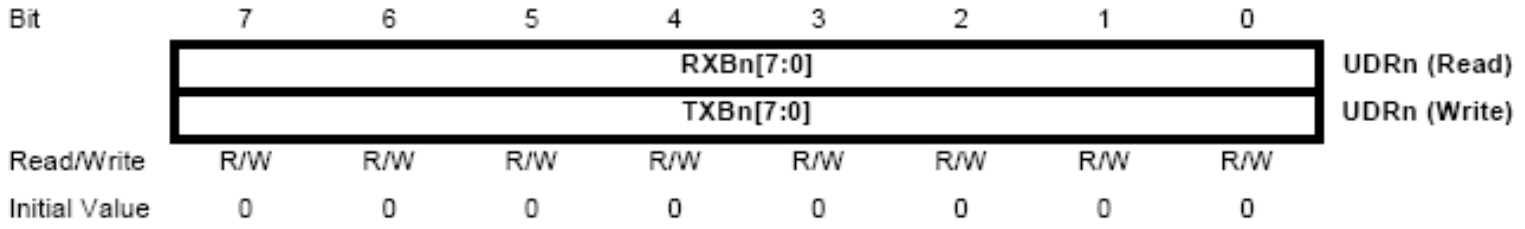
USART –CONTROL & STATUS registers

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

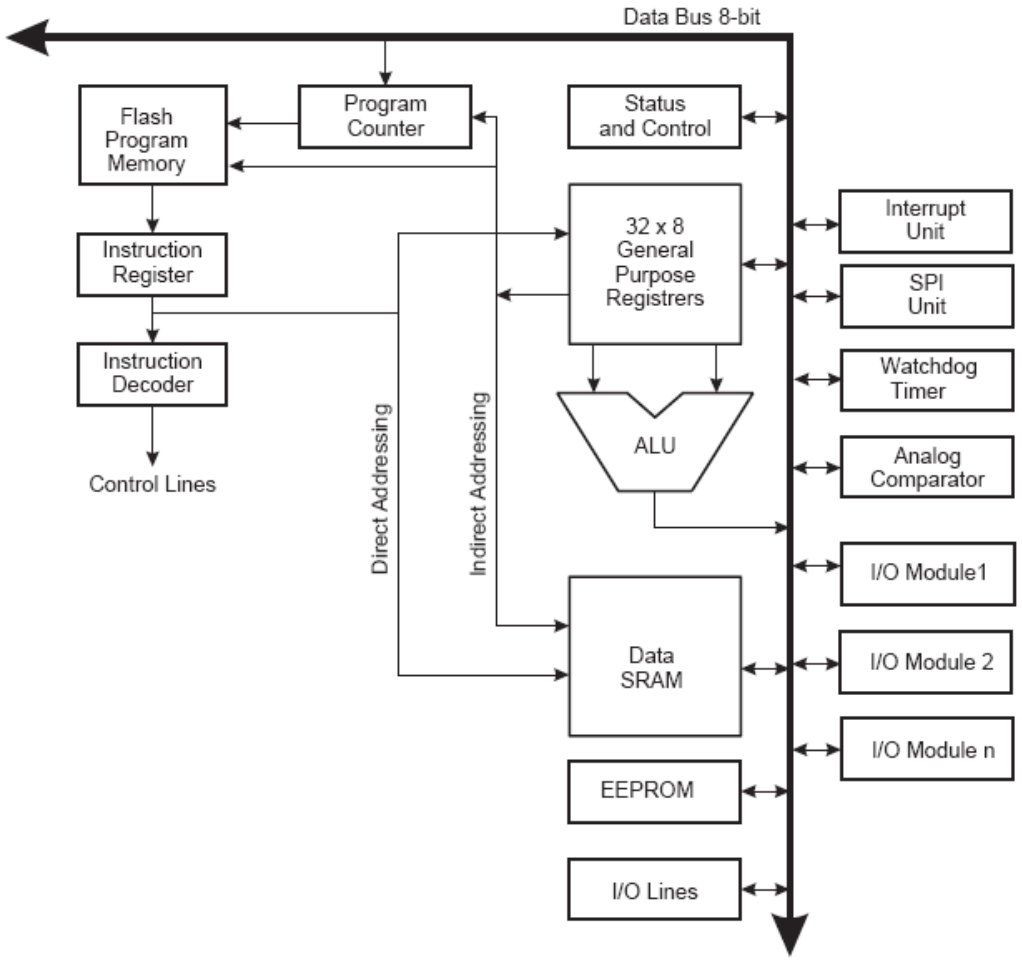
Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	–	UMSELn	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

USART data register



AVR Core (CPU)


Data Bus 8-bit
Flash Program Memory
Program Counter
Status and Control
Interrupt Unit
32 x 8 General Purpose Registrers
SPI Unit
Instruction Register
Watchdog Timer
Instruction Decoder
ALU
Analog Comparator
Control Lines
Direct Addressing
Indirect Addressing
I/O Module1
Data SRAM
I/O Module 2
I/O Module n
EEPROM
I/O Lines

AVR register file and status register

	7 … 0	Addr.	
General Purpose Working Registers	R0	$00	
	R1	$01	
	R2	$02	
	…		
	R13	$0D	
	R14	$0E	
	R15	$0F	
	R16	$10	
	R17	$11	
	…		
	R26	$1A	X-register Low Byte
	R27	$1B	X-register High Byte
	R28	$1C	Y-register Low Byte
	R29	$1D	Y-register High Byte
	R30	$1E	Z-register Low Byte
	R31	$1F	Z-register High Byte

The AVR status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Instruction execution timing

Figure 6. The Parallel Instruction Fetches and Instruction Executions

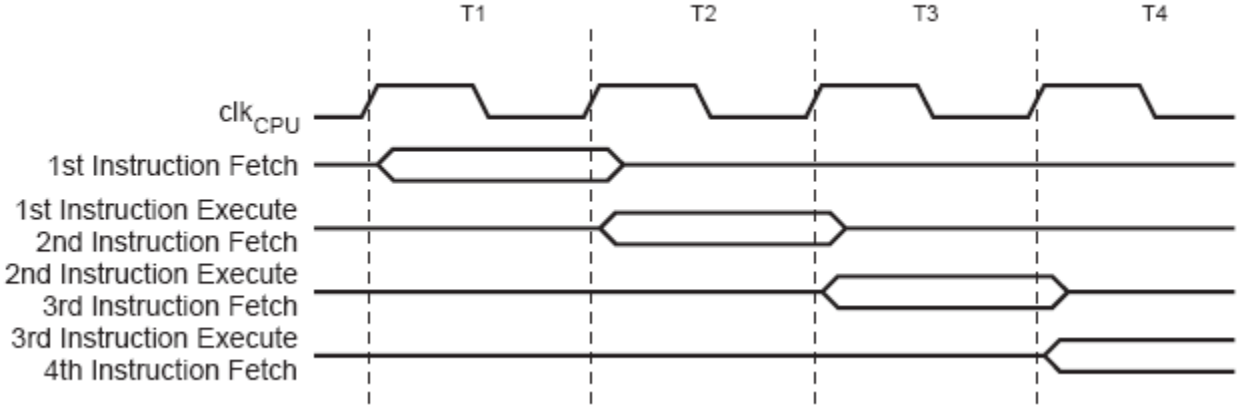
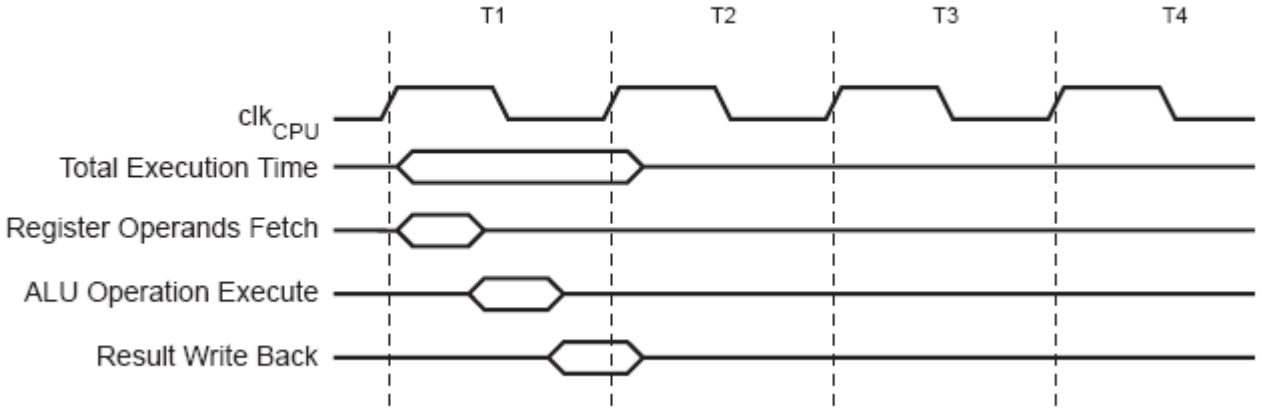


Figure 7. Single Cycle ALU Operation



Interrupt vector table (AVR8)

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	USART0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete

Interrupt vector table

```
Address   LabelsCode              Comments
$0000           jmp     RESET       ; Reset Handler
$0002           jmp     EXT_INT0    ; IRQ0 Handler
$0004           jmp     EXT_INT1    ; IRQ1 Handler
$0006           jmp     EXT_INT2    ; IRQ2 Handler
$0008           jmp     EXT_INT3    ; IRQ3 Handler
$000A           jmp     EXT_INT4    ; IRQ4 Handler
$000C           jmp     EXT_INT5    ; IRQ5 Handler
$000E           jmp     EXT_INT6    ; IRQ6 Handler
$0010           jmp     EXT_INT7    ; IRQ7 Handler
$0012           jmp     TIM2_COMP   ; Timer2 Compare Handler
$0014           jmp     TIM2_OVF    ; Timer2 Overflow Handler
$0016           jmp     TIM1_CAPT   ; Timer1 Capture Handler
$0018           jmp     TIM1_COMPA  ; Timer1 CompareA Handler
$001A           jmp     TIM1_COMPB  ; Timer1 CompareB Handler
$001C           jmp     TIM1_OVF    ; Timer1 Overflow Handler
$001E           jmp     TIM0_COMP   ; Timer0 Compare Handler
$0020           jmp     TIM0_OVF    ; Timer0 Overflow Handler
$0022           jmp     SPI_STC     ; SPI Transfer Complete Handler
$0024           jmp     USART0_RXC  ; USART0 RX Complete Handler
$0026           jmp     USART0_DRE  ; USART0,UDR Empty Handler
$0028           jmp     USART0_TXC  ; USART0 TX Complete Handler
```

Interrupt routine in C (AVR8)

- Declaration of the interrupt handler
  - There is not a standard way of declaring the handler
  - In general it is a function declaration with a special attribute

```
interrupt void func_name(void);
```

- Filling vector table with the Interrupt handler:

```
unsigned int* vector_table=0x00000000;
vector_table[10]=(unsigned int)&func_name;
```

Interrupt routine in C (AVR8)

- **ISR macro define in avr-gcc :**

```
#define ISR(vector)                                    \
void vector (void) __attribute__ ((signal));    \
void vector (void)
```

- **Interrupt enable and disable:**

```
# define sei()  __asm__ __volatile__ ("sei" ::)
# define cli()  __asm__ __volatile__ ("cli" ::)
```