week	lecture	topics
5	Micro-controller Programming I	- Digital I/O - A/D converter - Simple serial communication (RS-232, polling)

Digital input, digital output

- Most straight forward task for a microcontroller: Detecting the state of a binary sensor (digital input) and switching things on and off (digital output)
- All microcontrollers have a certain number of digital I/O lines which can be configured as inputs or as outputs; the output driver can be either open-collector/open-drain or push-pull (TTL); sometimes, either variant can be selected
- When a pin is configured as input, the corresponding output drivers are switched *tri-state* (also: hi-Z, high-impedance → switched off)

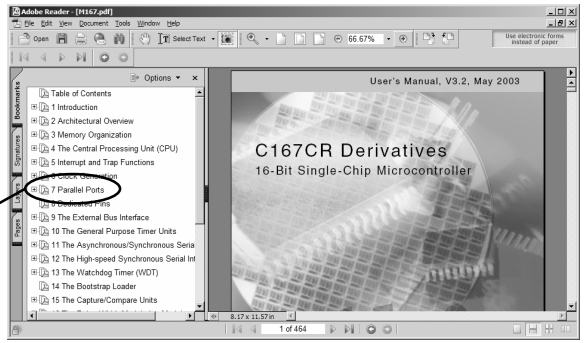
Digital input, digital output

- To configure a pin as input or output, the *Data Direction Register (DDR)* has to be programmed; each bit corresponds to one of the I/O lines of the controller; frequently, programming a '1' makes the corresponding I/O line act as output, a '0' produces an input; the reset value is commonly '0' (input)
- The state of an output line depends on the associated *output (latch) register*; each bit corresponds to a single pin on the microcontroller
- The logic level of an input line can be determined by reading the corresponding *input register*

Digital input, digital output

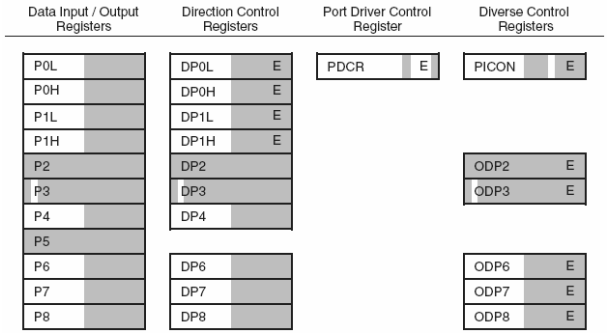
- For information on digital I/O, search the user manual of the microcontroller for *I/O ports* or *General Purpose I/O*

Example:
Infineon
C167CR-LM
→
Chapter 7:
Parallel Ports



Digital input, digital output

- The digital I/O unit of the C167CR features a variety of *8-bit and 16-bit ports,* each of which is controlled by 2/3 Special Function Registers (SFR)

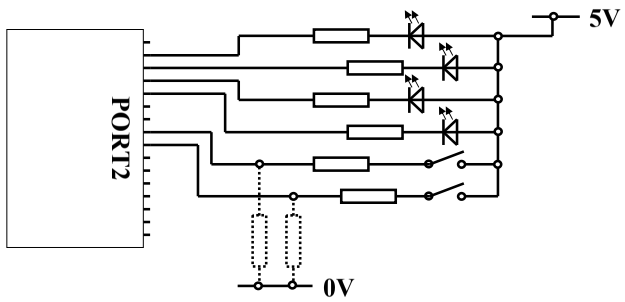


Digital input, digital output

- Every port has its own *data register* (Px or PxH, PxL, where x = 0 … 8); each bit of a data register corresponds to a different I/O line
- The data direction (input or output) of an I/O line is defined in the *direction control register* (DPx or DPxH/L); port 5 is always an input and therefore does not need a direction control register
- Ports 2, 3 and 6 – 8 can specify the output driver to be used (push-pull or open-drain); the SFR which controls this is called *Output Driver Control register* (ODx)

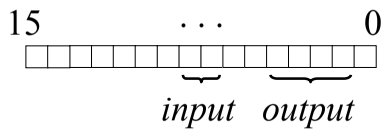
Digital input, digital output

- Example:
  Switch-on an array of LEDs conntected to pin 1 – 4 of port 2 and read the states of a pair of switches connected to pin 7 and 8 of the same port



Digital input, digital output

(1) Need to program bits 1 – 4 of port 2 as output while ensuring that bits 7 and 8 remain inputs



(2) Should change output driver constellation from push-pull to open-collector (safer)

(3) Need to set bits 1 – 4 to ‘1’ (line: logical high, output voltage: 5 V)

(4) Need poll state of bits 7, 8 of port 2 (inputs)

Digital input, digital output

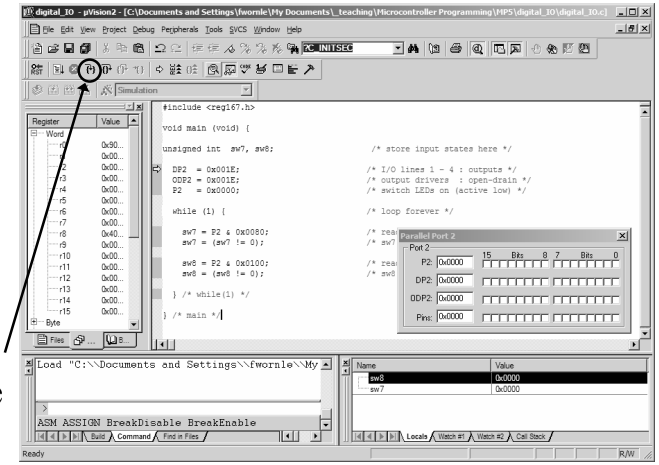
```
#include <reg167.h>

void main (void) {

unsigned int  sw7, sw8;                  /* store input states here */

  DP2  = 0x001E;                         /* I/O lines 1 - 4 : outputs */
  ODP2 = 0x001E;                         /* output drivers  : open-drain */
  P2   = 0x0000;                         /* switch LEDs on (active low) */

  while (1) {                            /* loop forever */

      sw7 = P2 & 0x0080;                /* read P2, filter out bit 7 */
      sw7 = (sw7 != 0);                 /* sw7 is '1' if switch pressed */

      sw8 = P2 & 0x0100;                /* read P2, filter out bit 8 */
      sw8 = (sw8 != 0);                 /* sw8 is '1' if switch pressed */

  } /* while(1) */

} /* main */
```

Note: the LEDs switch on when the port is driven low (*ACHTUNG!* Limit the current…)

Digital input, digital output

Start the simulator and go till *main*

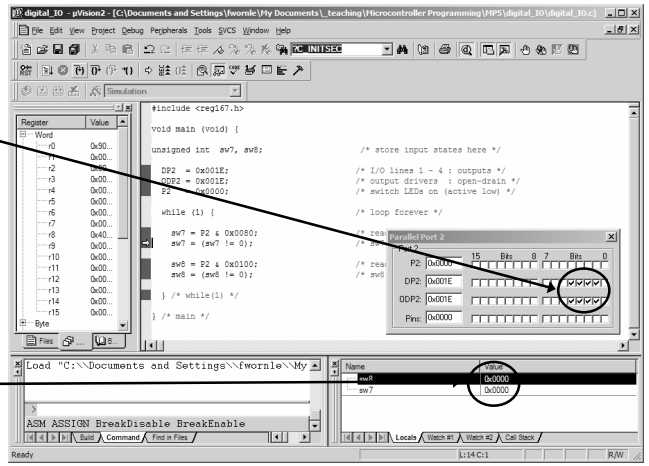
open *Port 2* and single step through the program



Digital input, digital output

Lines 1 – 4 become outputs with open-drain o/p drivers

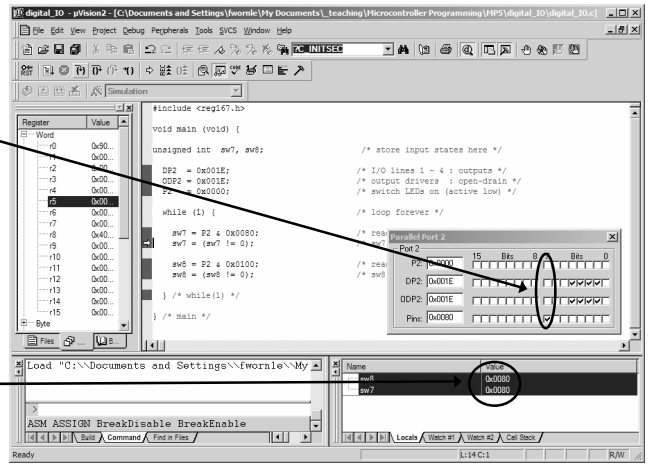
sw7 and *sw8* are both off



Digital input, digital output

Set line 7 to *high;* the data register P2 does **not** reflect this!

sw7 and *sw8* are **both** on!?!

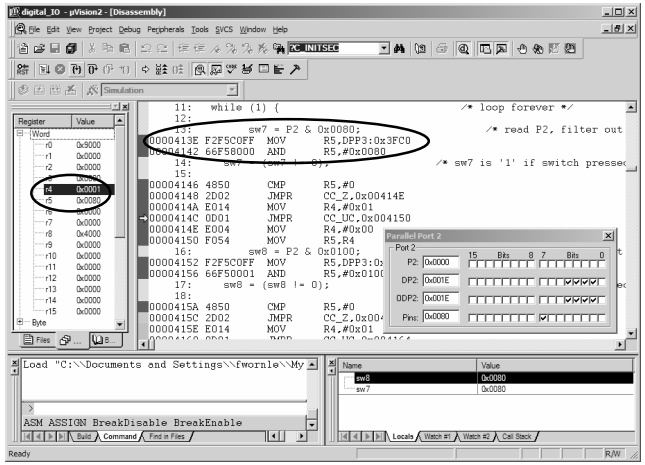


Digital input, digital output

- This should be surprising – sw8 seems to follow whatever happens to sw7 (and vice versa)!
- To find the reason for this strange behaviour it is advisable to switch the display from *source code only* to *mixed source and disassembly code*
- Single-stepping through the code reveals that *sw7* and *sw8* are both kept in *the same* register (R5) which is not saved but simply overwritten when the next value needs to be stored; this is why both variables appear as one – because *they are*

Digital input, digital output

Variables sw7 and sw8 are so-called *automatic variables;* they are often kept in registers



Digital input, digital output

- During compilation, the optimiser of the C166 compiler recognizes that *sw7* and *sw8* are both assigned values which are *never used* (i. e. they are never compared to anything or manipulated in any way); it therefore assigns both of these *automatic variables* to *the same* general purpose register (R5)
- There are a number of ways this possibly unwanted behaviour can be avoided: The most straight forward solution is to use *sw7* in a subsequent manipulation or comparison; its value must therefore be maintained while working on *sw8*…

Digital input, digital output

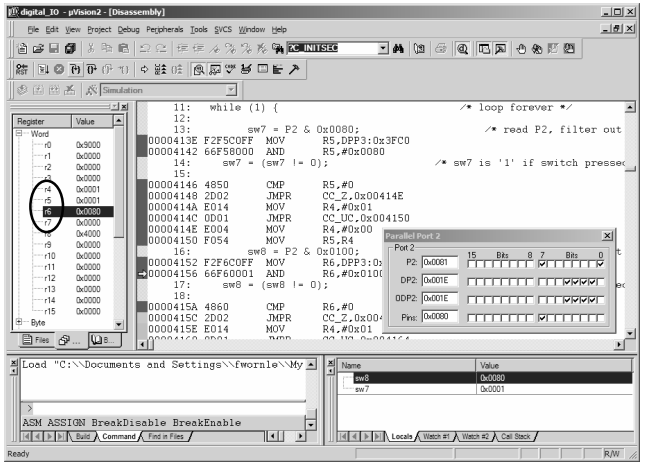
```
#include <reg167.h>

void main (void) {

unsigned int  sw7, sw8;                    /* store input states here */

  DP2  = 0x001E;                           /* I/O lines 1 - 4 : outputs */
  ODP2 = 0x001E;                           /* output drivers  : open-drain */
  P2   = 0x0000;                           /* switch LEDs on (active low) */

  while (1) {                              /* loop forever */

      sw7 = P2 & 0x0080;                  /* read P2, filter out bit 7 */
      sw7 = (sw7 != 0);                   /* sw7 is '1' if switch pressed */

      sw8 = P2 & 0x0100;                  /* read P2, filter out bit 8 */
      sw8 = (sw8 != 0);                   /* sw8 is '1' if switch pressed */

      if(sw7 == 1) P2 = P2 | 0x0001;      /* switch off LED1 */
      else         P2 = P2 & ~0x0001;     /* switch on LED1 */

  } /* while(1) */

} /* main */
```

Using sw7 after sw8 → the compiler needs to maintain its contents

Digital input, digital output

sw7 is now stored in R5, *sw8* is stored in R7

P2 now reflects the state of the line!?!



Digital input, digital output

- The parallel port display window of the simulator shows the state of the associated *output latch* (P2), the state of the *direction control register*, the state of the *output driver control register* and the *line status*
- The *output latch* only changes when the port register is written to; this is what is done now:

```
if(sw7 == 1) P2 = P2 | 0x0001;
else         P2 = P2 & ~0x0001;
```

Writing to output latch **Reading from input latch**

Digital input, digital output

- Setting a bit in C:

```
P2 = P2 | 0x0001;
```

Logical OR operator **Mask**

- Example: P2 contains value 0x1234

```
P2 =        0001.0010.0011.0100
P2 = P2 |   0000.0000.0000.0001
P2 =        0001.0010.0011.0101
```

- The above line can be abbreviated as follows:

```
P2 |= 0x0001;
```


Digital input, digital output

- Clearing a bit in C:

Logical NOT operator

```
P2 = P2 & ~0x0001;
```

Logical AND operator **Mask**

- Example: P2 contains value 0x1234

```
P2 =        0001.0010.0011.0100
P2 = P2 |  ~0000.0000.0000.0001
P2 =        0001.0010.0011.0100
P2 = P2 |   1111.1111.1111.1110
P2 =        0001.0010.0011.0100
```

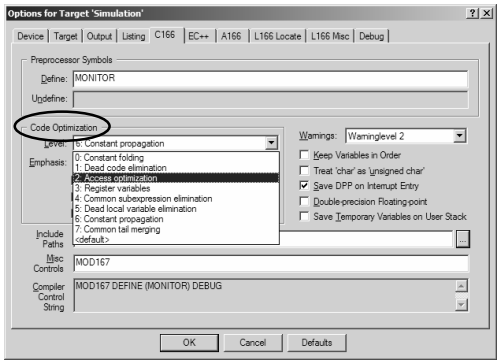
- The above line can be abbreviated as follows:

```
P2 &= ~0x0001;
```

Digital input, digital output

- A less intrusive way of avoiding an unwanted ‘optimization’ of automatic variables is by modifying the optimization options of the compiler

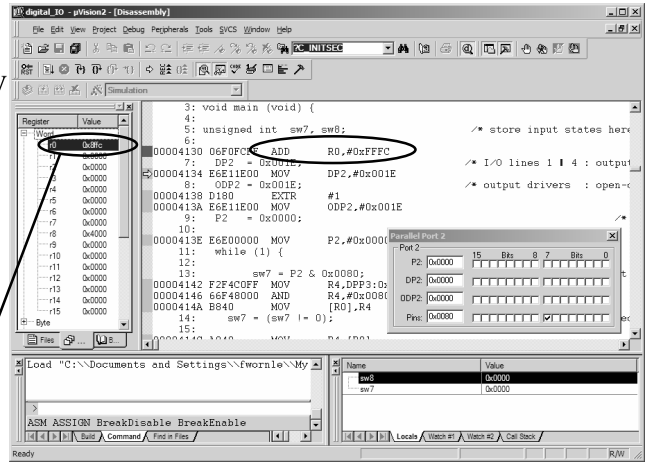
In the C166 compiler options menu a *level of code optimization* can be selected; choosing level ‘2’ avoids the use of register variables



Digital input, digital output

sw7 and *sw8* are now kept at different addresses *on the user stack;*

user stack pointer: R0



Digital input, digital output

- A more direct way of keeping sw7 and sw8 separate is to declare their *storage class* as *‘static’:*

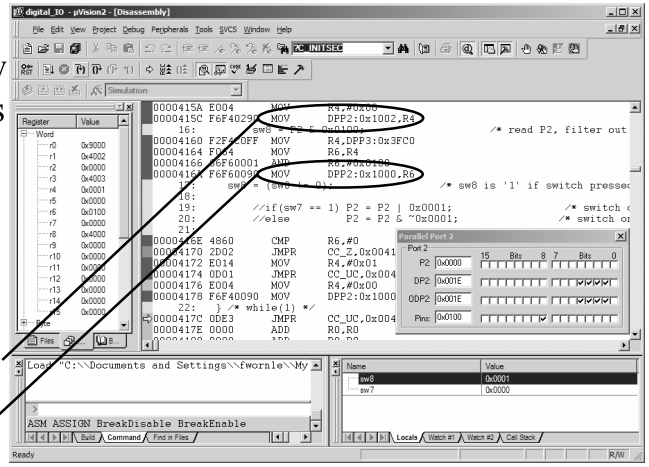
```
static unsigned int sw7, sw8;
```

- This informs the compiler that *sw7* and *sw8* are to be assigned their own individual space in memory

- As we are working with a SMALL memory model, these variables will end up in near memory (object class NDATA); access to *sw7* and *sw8* will thus be implemented using Data-Page Pointers (DPP)

Digital input, digital output

sw7 and *sw8* are now *static* and as such are allocated their own space in memory: **`DPP2:0x1002`** and **`DPP2:0x1000`**



Digital input, digital output

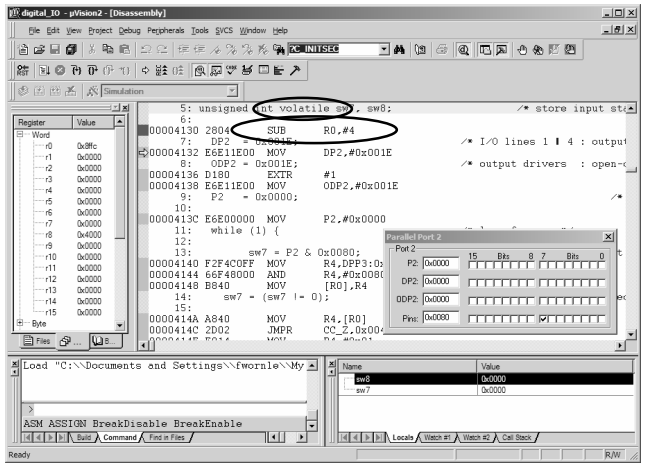
- The most elegant way of keeping *sw7* and *sw8* separate is to declare them using the *scope modifier 'volatile'*:

```
unsigned int volatile sw7, sw8;
```

- This informs the compiler that *sw7* and *sw8* are '*volatile*' in nature, i. e. their contents *can be modified from outside their scope* (e.g. by an interrupt routine)
- The optimiser is therefore stopped from assuming that the contents of *sw7* and *sw8* are never used

Digital input, digital output

automatic variables *sw7* and *sw8* have been declared *volatile* and thus are *not optimised away…*



Storage class specifiers [1]

- The term *storage class* refers to the method by which an object is assigned space in memory; common storage classes are *static*, *automatic* and *external*
- *Static* variables are given space in memory, at some fixed location within the program. Its value is faithfully maintained until we change it deliberately
- *Automatic* variables are dynamically allocated on the user stack when the block they are defined in is entered; they are discarded when this block is left
- The *external* storage class is used for objects that are defined outside the present source module

Storage class specifiers: *static*

- Making a global variable *static* limits its scope to the file it is defined in
- *Static local* variables are allocated their permanent storage location in RAM, unlike *non-static local (automatic)* variables which are created on the user stack; they thus retain their value from one instance of a function call to the next
- The assembly language name of the *static local* is a unique compiler-generated name (e.g. L1, L2, …); this allow the same C-language variable name to be used in different functions

Storage class specifiers: *static*

- Example:

```
void  main (void) {
unsigned int myLocal;

     myLocal = 100;
}
```

⇩

```
ADD      R0,#0xFFFE
MOV      R4,#0x0064
MOV      [R0],R4
ADD      R0,#2
```

```
void  main (void) {
static unsigned int myLocal;

     myLocal = 100;
}
```

⇩

```
MOV      R4,#0x0064
MOV      DPP2:0x1000,R4
```

↑ **Now a fixed address**

- Note that *static* variables can be made read-only by adding the modifier *const;* the linker commonly places these variables in ROM

Storage class specifiers: *automatic*

- *Automatic* variables are only required temporarily; they are local and only exist during the lifespan of the currently executed block
- If a function has *automatic* variables, the compiler generates instructions which – upon entry to this function – reserve the required amount of space on the user stack; this is commonly done by subtracting the required number of bytes from the stack pointer
- At the exit from the function, the old value of the stack pointer is restored, thereby discarding all local variables

Storage class specifiers: *automatic*

- Declaring an *automatic* variable as *const* causes the compiler to consider them as read-only; such a variable is still an automatic and, as such, is defined on the stack, i. e. in RAM – this is in contrast to *constant statics* which are commonly stored in ROM
- Common mistake: A pointer is returned to a variable which is destroyed at the end of the function

```
int *myFunct(void) {
int myLocal;
   myLocal = 100;
   return &myLocal;
}
```


Storage class specifiers: *external*

- The *external* storage class is used for objects that are defined outside the present source module
- When a variable is declared *external,* the compiler learns about its type (unsigned int, char, etc.) but it does not allocate memory for it or even know where it is kept; it's simply a *reference by name*
- *External* references are resolved at link time; when the linker scans all objects of a program for external references and inserts their final address into the instruction that refers to it

The Analog-to-Digital converter unit (ADC)

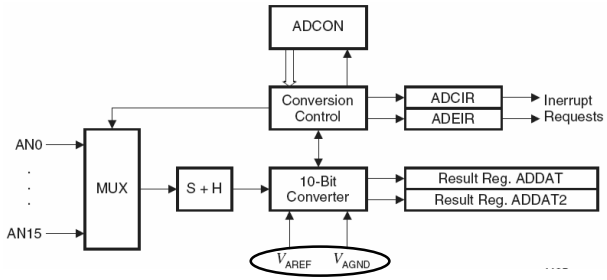
- Most sensors output analogue voltages to represent the value of the current measurement
- An embedded system can access analogue sensory information with the help of the integrated ADC unit; most frequently, this unit accepts input voltages in the range from 0 to 5 V
- A 10-bit ADC divides this range into $2^{10} = 1024$ steps, i. e. the resolution (defined by the least significant bit) is: $5\ V / 1024 \approx 4.88\ mV \approx 5\ mV$

The Analog-to-Digital converter unit (ADC)

- When interfacing a sensor to a microcontroller the sensor output signal should always be amplified to match the microcontroller input range; otherwise the effective resolution of the ADC is reduced
- Most ADC units can be set up to perform a *single conversion* on one or an entire group of channels, or they can perform *continuous conversions* on a single channel or a group of channels
- To configure the ADC unit, a number of control registers are provided

The Analog-to-Digital converter unit (ADC)

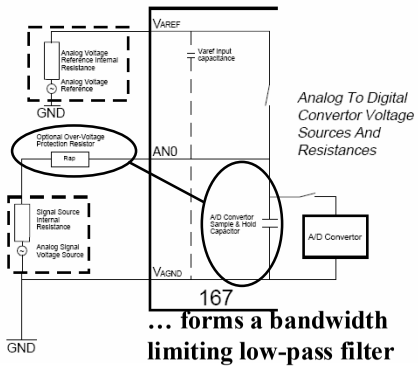
- Example: Infineon C167CR-LM
  This controller combines a 10-bit successive approximation A/D converter with a sample-and-hold amplifier (SHA) and a 16 channel multiplexer



Most ADC units have an isolated reference (noise reduction)

The Analog-to-Digital converter unit (ADC)

- The overall bandwidth of the ADC is limited by the input capacitance of the SHA
- The over-voltage protecting resistors have to be chosen with respect to the required bandwidth; high-bandwidth operation requires small resistors



… forms a bandwidth limiting low-pass filter

The Analog-to-Digital converter unit (ADC)

- ADCs are timed units – the conversion process is split into a number of phases which are scheduled at the clock frequency of the unit; this frequency is derived from the CPU clock
- Data sheets provide information about the minimum conversion time at various ADC clock frequencies and internal resistances of the voltage reference

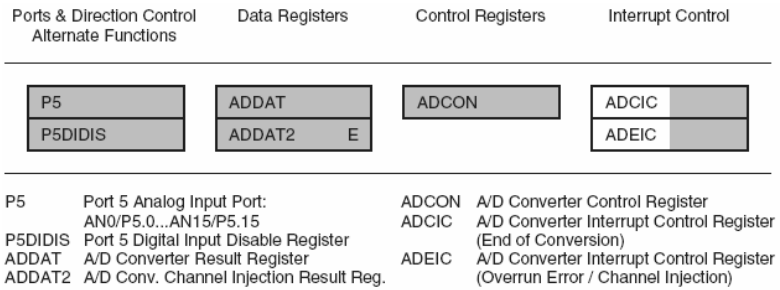
ADCON15-14 = 0	ADCTC	ADCON: Bits 13-12	0	1	10	11	ADSTC
Sample Time (us)			1.2	2.4	4.8	9.6	us
Overall Conversion Time (us)			9.7	10.9	13.3	18.1	us
Varef Source Resistance (Ohms)			3386	3386	3386	3386	Ohms
Signal Source Resistance (Ohms)			3386	7023	14295	28841	Ohms

ADCON15-14 = 10	ADCTC	ADCON: Bits 13-12	0	1	10	11	ADSTC
Sample Time (us)			4.8	9.6	19.2	38.4	us
Overall Conversion Time (us)			38.5	43.3	52.9	72.1	us
Varef Source Resistance (Ohms)			14295	14295	14295	14295	Ohms
Signal Source Resistance (Ohms)			14295	28841	57932	116114	Ohms

ADCON15-14 = 11	ADCTC	ADCON: Bits 13-12	0	1	10	11	ADSTC
Sample Time (us)			2.4	4.8	9.6	19.2	us
Overall Conversion Time (us)			19.3	21.7	26.5	36.1	us
Varef Source Resistance (Ohms)			7023	7023	7023	7023	Ohms
Signal Source Resistance (Ohms)			7023	14295	28841	57932	Ohms

The Analog-to-Digital converter unit (ADC)

- The following Special Function Registers (SFR) have to be set-up on the C167 before the ADC can be used:



The Analog-to-Digital converter unit (ADC)

- The pins of *port 5* are connected to the multiplexer of the ADC unit as well as to an array of input buffer amplifiers (port 5 can also be a digital input)
- It is good practice to deactivate these buffers when the port is used as analogue input. This reduces the amount of leakage current drawn from the sensors and avoids switching noise caused by the digital signal level detector for signals between V_{iL} and V_{iH}
- The digital input buffer amplifiers can be switched tri-state (→ disconnected) using register P5DIDIS

The Analog-to-Digital converter unit (ADC)

- The *conversion mode* is specified in ADCON (ADC Control Register); 4 modes are possible:
  - single channel, single conversion
  - single channel, continuous conversion
  - channel scan, single conversion
  - channel scan, continuous conversion
- In *single conversion* mode the ADC unit stops when the selected channel(s) has/have been converted once
- In *continuous conversion* mode the ADC unit starts over at the end of every (single or scan) conversion

The Analog-to-Digital converter unit (ADC)

- In *scan conversion* mode, the ADC unit converts the signals of a number of successive channels; the number of channels to be read can be programmed
- ADCON also reflects the state of the ADC:

ADC Control Register — SFR ($FFA0_H/D0_H$) — Reset value: 0000_H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCTC		ADSTC		AD CRQ	AD CIN	AD WR	AD BSY	AD ST	-	ADM		ADCH			
rw		rw		rwh	rw	rw	rwh	rwh	-	rw		rw			

ADCH: channel number, ADM: conversion mode, ADST: start/stop bit, ADBSY: busy flag, etc.

The Analog-to-Digital converter unit (ADC)

- The result of a conversion is stored in the ADDAT register; as this register is used by all channels the channel number is stored with the result:

ADC Result Register — SFR ($FEA0_H/50_H$) — Reset value: 0000_H

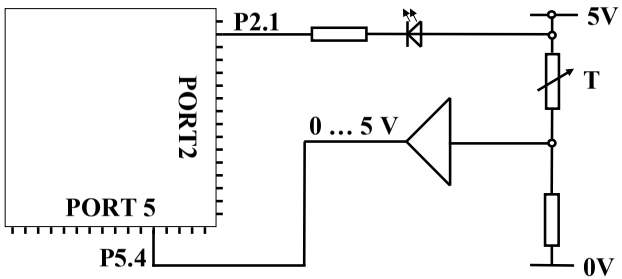
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CHNR				-	-	ADRES									
rwh				-	-	rwh									

CHNR: channel number of the last conversion
ADRES: corresponding result (10-bit)

- In *scanning mode,* the program has to make sure that ADDAT is read before its value gets overwritten

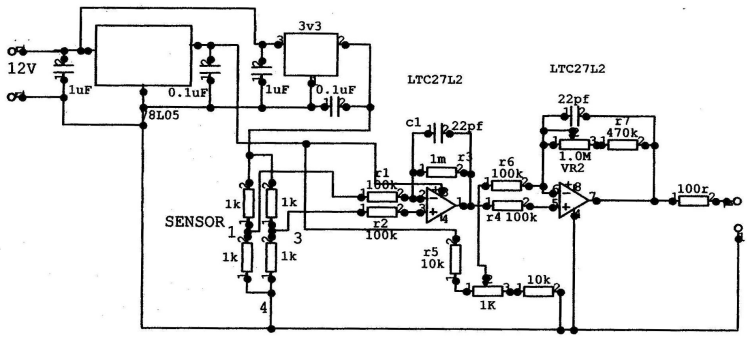
The Analog-to-Digital converter unit (ADC)

- Example:
  Read out a temperature sensor connected to channel 4 of the A/D converter unit (Port 5, pin 4 → P5.4); switch on an LED (P2.1) if the value is above 2 V



The Analog-to-Digital converter unit (ADC)

- A sample design of the level shifting amplifier is shown below (removes offset, scales to 0 … 5 V):



The Analog-to-Digital converter unit (ADC)

```
#include <reg167.h>

/* reg167.h doesn't seem to define P5DIDIS... */
#define  P5DIDIS (*((unsigned int volatile sdata *) 0xFFA4))

void main (void) {

float myVoltage;                    /* store scaled result */

  DP2  |= 0x0002;                   /* I/O lines 2 : output */
  ODP2 |= 0x0002;                   /* output drivers  : open-drain */
  P2   |= 0x0002;                   /* switch LED off (active low) */

   (…)
```

The register definition file of the KEIL compiler *(reg167.h)* appears to be missing the definition of macro P5DIDIS; it therefore has been created manually as a pointer to an *unsigned integer* in system data (sdata, page 3: 0xC000 – 0xFFFF) with the address 0xFFA4. The modifier *volatile* has been used to ensure that the optimiser does not remove lines such as 'P5DIDIS = 0x0004' (see code, next slide)

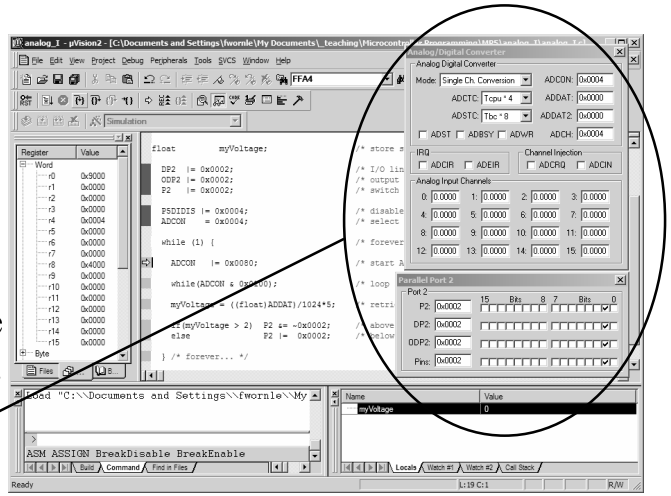
The Analog-to-Digital converter unit (ADC)

```
  P5DIDIS |= 0x0004;              /* disable i/p buffer in parallel ch 4 */
  ADCON    = 0x0004;              /* select channel 4 for single conversion */

  while (1) {                     /* forever... */

     ADCON   |= 0x0080;           /* start ADC */
     while(ADCON & 0x0100);       /* loop in this line until ADBSY is clear */

     /* retrieve result, eliminate channel number and rescale to 0 ... 5 */
     myVoltage = ((float)(ADDAT & 0x03FF))/1024*5;

     if(myVoltage > 2)  P2 &= ~0x0002;   /* above 2 V -> switch LED on */
     else               P2 |=  0x0002;   /* below or at 2V -> switch LED off */

  } /* forever... */
} /* main */
```

Upon retrieval of the result from ADDAT, the channel number (top 4 bits) needs to be deleted (only retaining the bottom 10 bits) and the result needs to be rescaled from '0 – 1023' to '0 – 5'

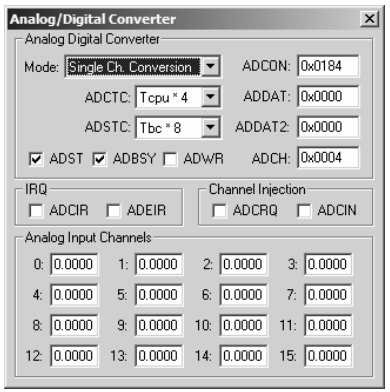
The Analog-to-Digital converter unit (ADC)

The KEIL simulator can display all registers associated with any of its hardware units (ADC, parallel ports, etc.)



The Analog-to-Digital converter unit (ADC)

- The ADC control window allows the monitoring as well as the modification of all aspects of the unit
- Setting bit 2 and clearing 4 and 5 of ADCON configures the ADC for a single conversion on channel 4; to start the unit, the ADST bit (bit 7) needs to be set – once started, the ADBSY bit comes on



The Analog-to-Digital converter unit (ADC)

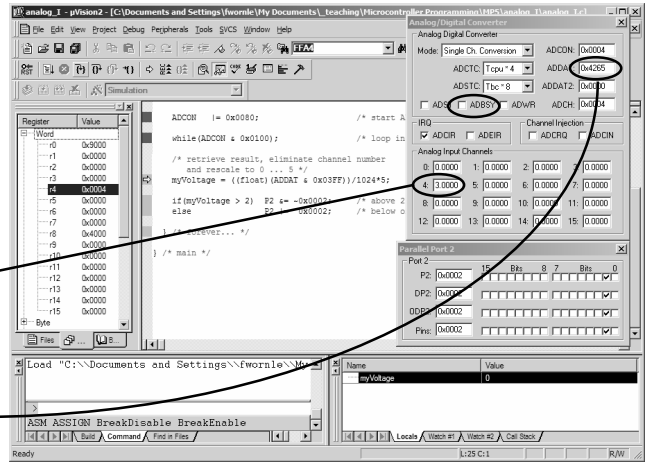
- Following the start of the conversion, the program will have to wait for its completion; this can be done by checking the ADBSY flag in ADCON – this flag is on throughout the conversion and is cleared at its end

```
while(ADCON & 0x0100);
```

- Note that the simulator waits the exact number of CPU cycles which would elapse on a real board before the blocking while statement is overcome

The Analog-to-Digital converter unit (ADC)

After the conversion, the converted voltage (entered at channel 4) can be found in ADDAT



The Analog-to-Digital converter unit (ADC)

- The converted value needs to be rescaled. This starts with the elimination of the channel information from the top 4 bits of ADDAT (ADDAT & 0x03FF); the result of this operation needs to be cast into a *float* and then normalized by the full-scale value (1023 = 0x3FF); finally the normalized value needs to be rescaled to the required output range (0 … 5 V)

```
   and rescale to 0 ... 5 */
myVoltage = ((float)(ADDAT & 0x03FF))/1023*5;
```


The Asynchronous Serial Communication interface

- Debugging embedded controller software can be a difficult task, in particular in the absence of professional development tools such as background debug equipment; in this case, the programmer's best friend is the (asynchronous) serial interface ASC. It is therefore very important to be able to set up simple communications through the serial interface to a terminal program on a personal computer (PC)

- The following explanation is just the most basic way information can be communicated through the ASC

The Asynchronous Serial Communication interface

- Register S0CON controls the operating mode of the interface (number of data bits, number of stop bits, error checking, baud rate generation, start/stop, etc.)

ASC0 Control Register SFR ($FFB0_H/D8_H$) Reset value: 0000_H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S0R	S0 LB	S0 BRS	S0 ODD	-	S0 OE	S0 FE	S0 PE	S0 OEN	S0 FEN	PEN	S0 REN	S0 STP	S0M		
rw	rw	rw	rw	-	rwh	rwh	rwh	rw	rw	rw	rwh	rw	rw		

- The default settings of the most commonly used terminal programs (e.g. Hyperterm) are:
  9600 bps, 8 data bits, 1 stop bit, no parity checking

The Asynchronous Serial Communication interface

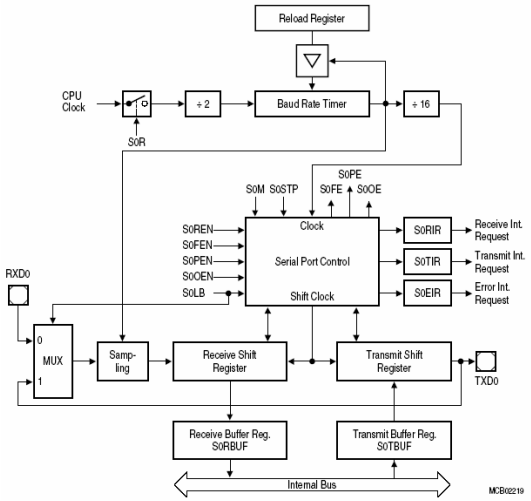
- On the C167CR, a serial transmission is started by writing the data to be sent into the transmission buffer S0TBUF; this register is double-buffered, i. e. new data can be written to the shadow register while a transmission is ongoing, thereby allowing for back-to-back transmissions without gaps
- Serial reception is enabled by setting the Receiver Enable bit (S0REN) in S0CON; this register is also double-buffered: the next character can be received (into the shadow register) while the previous one is still being read from the Reception Buffer (S0RBUF)

The Asynchronous Serial Communication interface

- Receive buffer overrun (i. e. S0RBUF is overwritten before it has been read by the program) can be detected and is announced through the Overflow Error flag (S0OE) in S0CON; this also sets the Error Interrupt Request flag (S0EIR) and, if enabled, may trigger an interrupt
- Data is transmitted on pin P3.10 (TxD) and received on pin P3.11 (RxD); these bits of port 3 have to be set up as output and input, respectively

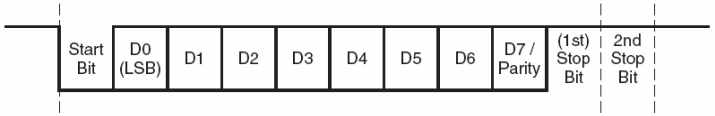
The Asynchronous Serial Communication interface

The serial communication interface on the C167 has two independent and double-buffered shift registers for back-to-back (gap-free) full duplex serial communication



The Asynchronous Serial Communication interface

- Data is organised in frames consisting of a start bit (always zero), followed by 7 or 8 data bits, an optional parity bit (7-bit transmissions only) and up to 2 stop bits



- The effective data throughput of the interface depends on the baud rate, the amount of overhead of each character (parity checking, stop bits) and the length of the gaps between successive characters

The Asynchronous Serial Communication interface

- The baud rate is derived from the CPU clock signal as follows:

$$B_{Async} = \frac{f_{CPU}}{16 \times (2 + \langle S0BRS \rangle) \times (\langle S0BRL \rangle + 1)}$$

- The 13-bit constant S0BRL provides the main divisor in reducing the commonly very large frequency of the CPU (20 MHz); setting bit S0BRS increases the divisor by an additional factor 3 – this allows the programming of baud rates which are *odd-valued* fractions of f_{CPU}

The Asynchronous Serial Communication interface

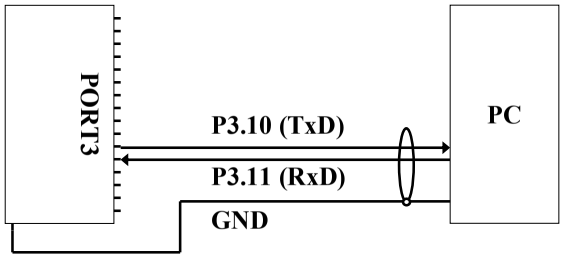
- Baud rates as generated from a 20 MHz clock signal:

Baud Rate	S0BRS = '0'		S0BRS = '1'	
	Deviation Error	Reload Value	Deviation Error	Reload Value
625 KBaud	± 0.0%	0000_H	–	–
19.2 KBaud	+ 1.7%/ – 1.4%	$001F_H/0020_H$	+ 3.3%/ – 1.4%	$0014_H/0015_H$
9600 Baud	+ 0.2%/ – 1.4%	$0040_H/0041_H$	+ 1.0%/ – 1.4%	$002A_H/002B_H$
4800 Baud	+ 0.2%/ – 0.6%	$0081_H/0082_H$	+ 1.0%/ – 0.2%	$0055_H/0056_H$
2400 Baud	+ 0.2%/ – 0.2%	$0103_H/0104_H$	+ 0.4%/ – 0.2%	$00AC_H/00AD_H$
1200 Baud	+ 0.2%/ – 0.4%	$0207_H/0208_H$	+ 0.1%/ – 0.2%	$015A_H/015B_H$
600 Baud	+ 0.1%/ – 0.0%	$0410_H/0411_H$	+ 0.1%/ – 0.1%	$02B5_H/02B6_H$
75 Baud	+ 1.7%	$1FFF_H$	+ 0.0%/ – 0.0%	$15B2_H/15B3_H$
50 Baud	–	–	+ 1.7%	$1FFF_H$

75 bps is an odd-valued fraction of 20 MHz…

The Asynchronous Serial Communication interface

- Example:
  The character sequence 'Hello World\n' is to be sent to the host (PC), communicating at 9600 bps with 8 data bits, 1 stop bit and no parity checking



The Asynchronous Serial Communication interface

```
#include <reg167.h>

/* the following text is stored in ROM, last byte: 0 */
static const char  myText[] = "Hello World\n";

void main (void) {

const char  *nextC = myText;

#ifndef MONITOR                    /* do not initialize when monitor is used */
  DP3 |=  0x0400;                  /* Port DP3.10 : output (TxD) */
  P3  |=  0x0400;                  /* Port P3.10 : high (inactive) */
  DP3 &= ~0x0800;                  /* Port DP3.11 : input (RxD) */
  S0TIC = 0x80;                    /* set transmit interrupt flag */
  S0RIC = 0x00;                    /* clear receive interrupt flag */
  S0BG  = 0x40;                    /* program baud rate to 9600 bps */
  S0CON = 0x8011;                  /* configure serial interface */
#endif /* MONITOR */
```

myText is defined as *static* (global at file level) *const* (placed in ROM); the array length is determined automatically by the compiler; a zero-byte is appended to the end of this string

The Asynchronous Serial Communication interface

```
#include <reg167.h>

/* the following text is stored in ROM, last byte: 0 */
static const char  myText[] = "Hello World\n";

void main (void) {

const char  *nextC = myText;       /* note: myText = &myText[0] */

#ifndef MONITOR                    /* do not initialize when monitor is used */
  DP3 |=  0x0400;                  /* Port DP3.10 : output (TxD) */
  P3  |=  0x0400;                  /* Port P3.10 : high (inactive) */
  DP3 &= ~0x0800;                  /* Port DP3.11 : input (RxD) */
  S0TIC = 0x80;                    /* set transmit interrupt flag */
  S0RIC = 0x00;                    /* clear receive interrupt flag */
  S0BG  = 0x40;                    /* program baud rate to 9600 bps */
  S0CON = 0x8011;                  /* configure serial interface */
#endif /* MONITOR */
```

Automatic variable *nextC* is defined as a *pointer to a const char*; as we are compiling in the SMALL memory model, this pointer will be implemented as *near pointer* (access via the DPP registers); *nextC* is initialized to point the beginning of the string myText

The Asynchronous Serial Communication interface

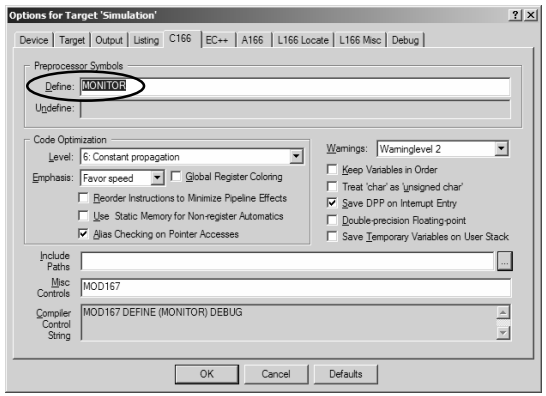
```
#include <reg167.h>

/* the following text is stored in ROM, last byte: 0 */
static const char  myText[] = "Hello World\n";

void main (void) {

const char  *nextC = myText;       /* note: myText = &myText[0] */

#ifndef MONITOR                    /* do not initialize when monitor is used */
  DP3 |=  0x0400;                  /* Port DP3.10 : output (TxD) */
  P3  |=  0x0400;                  /* Port P3.10 : high (inactive) */
  DP3 &= ~0x0800;                  /* Port DP3.11 : input (RxD) */
  S0TIC = 0x80;                    /* set transmit interrupt flag */
  S0RIC = 0x00;                    /* clear receive interrupt flag */
  S0BG  = 0x40;                    /* program baud rate to 9600 bps */
  S0CON = 0x8011;                  /* configure serial interface */
#endif /* MONITOR */
```

This section is only compiled into the code if macro MONITOR has not been defined; this allows the same source code file to be used with the simulator (requires initialization of the serial interface) as well as with the on-chip monitor program (ASC0 is already initialized)

The Asynchronous Serial Communication interface

- Macro MONITOR is used to control the initialization of ASC0 (simulation: yes, monitor target: no)

To use the simulation target, this macro should be deleted from the *Define* box on the compiler options page



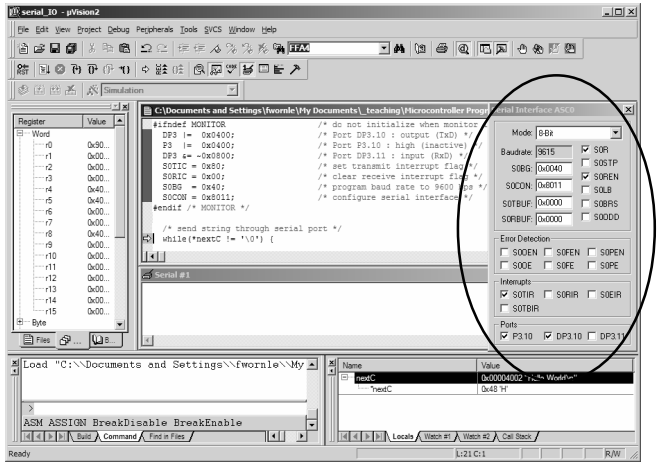
The Asynchronous Serial Communication interface

```
#include <reg167.h>

/* the following text is stored in ROM, last byte: 0 */
static const char  myText[] = "Hello World\n";

void main (void) {

const char  *nextC = myText;       /* note: myText = &myText[0] */

#ifndef MONITOR                    /* do not initialize when monitor is used */
  DP3  |=  0x0400;                 /* Port DP3.10 : output (TxD) */
  P3   |=  0x0400;                 /* Port P3.10 : high (inactive) */
  DP3  &= ~0x0800;                 /* Port DP3.11 : input (RxD) */
  S0TIC = 0x80;                    /* set transmit interrupt flag */
  S0RIC = 0x00;                    /* clear receive interrupt flag */
  S0BG  = 0x40;                    /* program baud rate to 9600 bps */
  S0CON = 0x8011;                  /* configure serial interface */
#endif /* MONITOR */
```

Initialization of the serial interface (ASC0): P3.10 is output (TxD), P3.11 is input (RxD), S0TIC indicates 'transmission finished', S0RIC indicates 'reception buffer clear', B0BG selects a baud rate of 9600 bps and S0CON is set to 8 data bits / no parity, 1 stop bit and *active*

The Asynchronous Serial Communication interface

All parameters of the serial interface (ASC0) can be monitored / modified in its control window



The Asynchronous Serial Communication interface

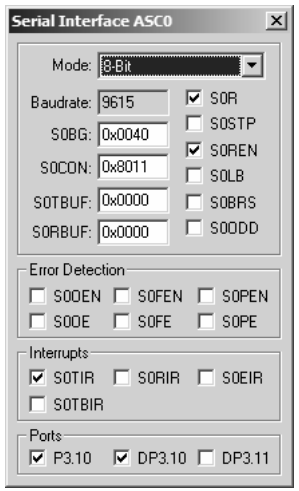
- The baud rate register (S0BG) has been loaded with 0x0040; the corresponding baud rate is therefore:

$$f_{BR} = \frac{f_{CPU}}{16 \cdot (2+0) \cdot (0x0040+1)}$$

$$= \frac{20 \cdot 10^6}{16 \cdot (2+0) \cdot (64+1)}$$

$$\approx 9615.385$$

$$= 9600 + 0.16\%$$



The Asynchronous Serial Communication interface

```
  /* send string through serial port */
  while(*nextC != '\0') {

    while(S0TIR == 0);              /* wait for end of ongoing transmission */
    S0TIR = 0;                      /* clear interrupt request flag */
    S0TBUF = *nextC++;              /* start transmission of next character */

  }

  while(1);                         /* forever... */

} /* main */
```

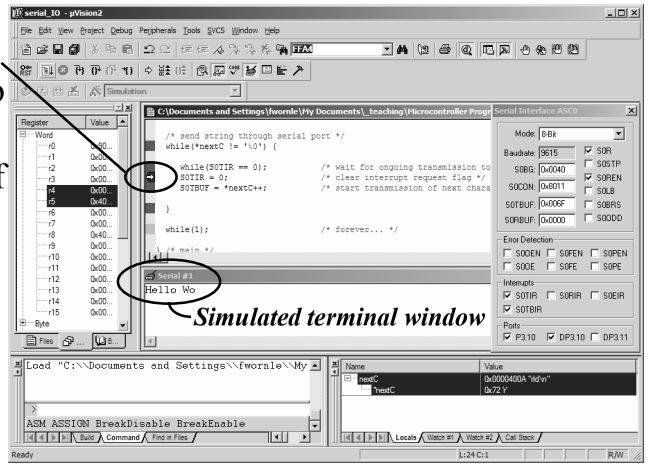
First check, if the end of string (character '\0' = a zero-byte) has been reached (note: **nextC* fetches the contents of the memory location *nextC* points to)

Character-by-character transmission:

(1) Ensure that a possibly ongoing transmission is finished
(2) Clear Transmission Interrupt Request flag
(3) Send next character *(*nextC)* and increment pointer *nextC* by 1

The Asynchronous Serial Communication interface

Breakpoints can be set to observe execution of the code in an efficient way (e.g. after the *while* statement)



Further reading:

[1]
Jonathan W. Valvano, *Developing Embedded Software in C Using ICC11/ICC12/Hiware,* www.ece.utexas.edu/~valvano/embed/toc1.htm, accessed: January 2005

[2]
Tom Swan, *Tom Swan's GNU C++ FOR LINUX,* Que Corporation, 2000, ISBN: 0-7897-2153-8