week	lecture	topics

7	Micro-controller programming III	- Buffered serial communication
		  - Ring buffers
		  - Interrupt driven communication
		- Other communication protocols:
		  - CAN
		  - IIC/I²C
		  - Ethernet-TCP/IP
		  - USB

Buffered Serial Communication

- The *serial interface* of a microcontroller is probably the most commonly used means for host-target communication
- An efficient implementation of the routines for serial communication is therefore an important aspect of embedded systems engineering
- This lecture will develop code fragments for interrupt driven buffered serial communication; the buffers will be implemented efficiently as ring-buffers

Buffered Serial Communication

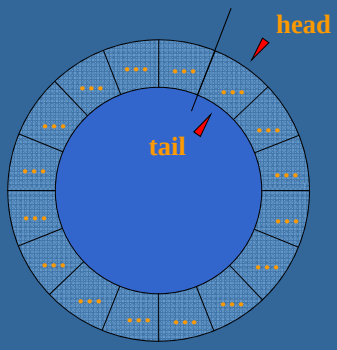
- In *buffered serial communication* the program simply places the data to be sent in a *transmission buffer* and carries on with whatever it has been designed to do
- A background process (usually interrupt driven) then takes care of the actual mechanics of shifting the data out on the serial line
- Reception works in a similar way: Incoming data is received by a background process and placed in an *reception buffer*; the program (foreground task) can access this data whenever convenient

Buffered Serial Communication

- Both *transmission buffer* as well as *reception buffer* will have to be safeguarded against *buffer overflow*
- A suitable implementation of such a communication buffer is the so-called *ring buffer*
- *Ring buffers* can be interpreted as circular memories, where the *tail* (end of buffer) joins the *head* (start of buffer); data is written to the buffer at its *head* and read from the *tail*, thereby implementing a *First-In-First-Out (FIFO)* buffer
- A pair of *pointers* or *indices* is used to address both the head as well as the tail of the buffer

Buffered Serial Communication

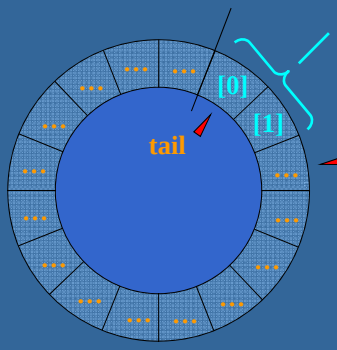
- Example: Ring buffer (16 elements, buffer empty)



The buffer is empty when:
tail = head

Buffered Serial Communication

- Example: Ring buffer (after two write operations)



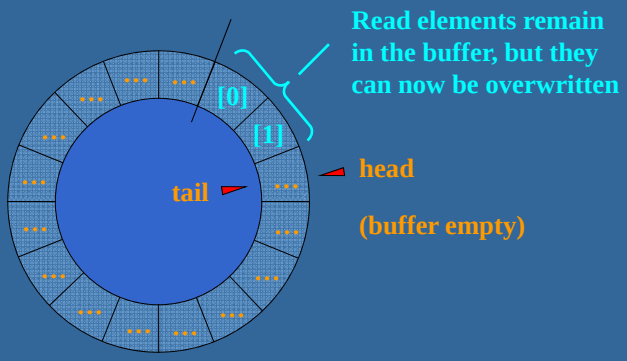
2 elements have been stored in the buffer

head

in this implementation write operations are *followed* by the advancing of the head pointer (if possible) *('post-increment')*

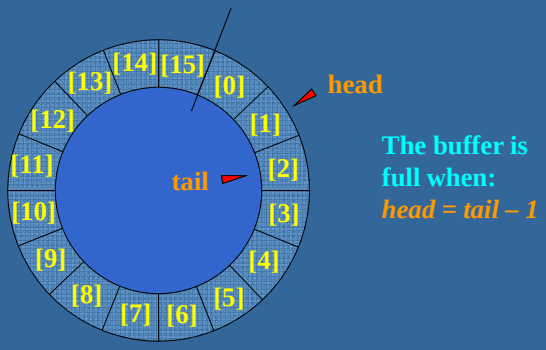
Buffered Serial Communication

- Example: Ring buffer (after two read operations)



Buffered Serial Communication

- Example: Ring buffer (after 16 write operations)



Buffered Serial Communication

- A *ring buffer administration structure* can be used to administer the buffer:

```
struct {
          char  *rb_start;    /* beginning of the buffer */
          char  *rb_end;      /* end of the buffer */
          char  *rb_in;       /* write position */
          char  *rb_out;      /* read position */
       }
```

- Other implementations may also store the number of elements on the buffer or they may refer to head and tail by their relative indices with respect to a base pointer (more efficient for small buffers); also note that the elements do not have to be of type *char*

Buffered Serial Communication

- It is good programming style to implement a number of *buffer access macros;* these are replaced by their definitions during the *pre-compilation* stage

- The use of macros optimizes a program for fast execution; macros differ from function calls in that they do not introduce any unnecessary overhead (call and return instructions) and, therefore, the program runs as fast as possible – the cost of this is a possibly slightly bigger program (especially if a macro is large and used frequently throughout the program)

Buffered Serial Communication

- Buffer access macro *RB_CREATE(rb, type)*

```
#define RB_CREATE(rb, type) \
   struct { \
     type *rb_start; \
     type *rb_end; \
     type *rb_in; \
     type *rb_out; \
   } rb
```

Macro RB_CREATE(rb, type) defines a *ring buffer administration structure;* the name of this structure is to be specified through the first macro parameter (rb), the type of element to be stored in this buffer is to be specified through the second macro parameter (type)

Example: ***RB_CREATE(inbuf, char)***
… defines a ring buffer admin structure *inbuf* for a buffer storing elements of type *char*

Buffered Serial Communication

- Buffer access macro *RB_INIT(rb, start, number)*

```
#define RB_INIT(rb, start, number) \
        ( (rb)->rb_in = (rb)->rb_out= (rb)->rb_start= start, \
          (rb)->rb_end = &(rb)->rb_start[number] )
```

Macro RB_INIT(rb, start, number) initializes the ring buffer administration structure; head and tail pointer (rb_in and rb_out, respectively) point to the beginning of the empty buffer (rb_start); the end-of-buffer pointer is set to the end element of the buffer

Example: ***RB_INIT(inbuf, &myBuf, 100)***
… initializes the previously defined buffer admin structure for buffer myBuf (to be defined statically or dynamically), specifying a total size of 100 buffer elements (e.g. char, int, long, float, struct, …)

Buffered Serial Communication

- Buffer access macro *RB_SLOT(rb, slot)*

```
#define RB_SLOT(rb, slot) \
        ( (slot)==(rb)->rb_end? (rb)->rb_start: (slot) )
```

Macro RB_SLOT(rb, slot) is a support macro which makes the buffer circular; all access to the elements of the buffer (read and write operations) needs to be done through this macro

The macro returns the physical address of the element to be accessed; this is either parameter *slot* itself (access to a regular buffer element → no wrapping required) or the first address of the buffer memory (tempted access to element *rb->end+1* → buffer needs to be wrapped)

Example: ***RB_SLOT(inbuf, inbuf->rb_in+1)***
… returns either the address *inbuf->rb_in+1* (not wrapped) or *inbuf->rb_start* (buffer wrapped)

Buffered Serial Communication

- Buffer access macros *RB_EMPTY(rb), RB_FULL(rb)*

```
#define RB_EMPTY(rb) ( (rb)->rb_in==(rb)->rb_out )

#define RB_FULL(rb)  ( RB_SLOT(rb, (rb)->rb_in+1)==(rb)->rb_out )
```

Macro RB_EMPTY(rb) tests if the buffer is completely empty; in this implementation, the ring buffer is empty when head and tail pointer address the same buffer element (rb_in = rb_out)

Macro RB_FULL(rb) tests if the buffer is full; this is the case when *tail = head + 1*. Note that support macro RB_SLOT is used to ensure that the buffer is wrapped around (if required)

Example: ***if(!RB_EMPYT(inbuf)) { /* buffer not empty */ … }***
if(!RB_FULL(inbuf)) { /* buffer not full */ … }

Buffered Serial Communication

- Buffer access macros *RB_PUSHSLOT(rb)* and *RB_POPSLOT(rb)*

```
#define RB_PUSHSLOT(rb) ( (rb)->rb_in )

#define RB_POPSLOT(rb)  ( (rb)->rb_out )
```

Macro RB_PUSHSLOT(rb) returns the address of the element the head pointer currently points to

Macro RB_POPSLOT(rb) returns the address of the element the tail pointer currently points to

Example: ***RB_PUSHSLOT(outbuf) = myData;***
… the '*' operator accesses the element the head pointer of *outbuf* points to; myData is stored there
myData = *RB_POPSLOT(inbuf);
… fetches the next element from buffer *inbuf*

Buffered Serial Communication

- Buffer access macros *RB_PUSHADVANCE(rb)* and *RB_POPADVANCE(rb)*

```
#define RB_PUSHADVANCE(rb) ( (rb)->rb_in= RB_SLOT((rb), (rb)->rb_in+1) )

#define RB_POPADVANCE(rb)  ( (rb)->rb_out= RB_SLOT((rb), (rb)->rb_out+1) )
```

Macro RB_PUSHADVANCE(rb) advances the head pointer of buffer 'rb' by one, taking into account possibly required buffer wrapping

Macro RB_POPADVANCE(rb) advances the tail pointer of buffer 'rb' by one, taking into account possibly required buffer wrapping

Example: ***RB_PUSHADVANCE(outbuf);***
RB_POPADVANCE(inbuf);

Buffered Serial Communication

- All access to the ring buffers is done through these access macros; there are many more such macros that could have been defined, thus providing for ring buffers that are managed slightly differently (e.g. the *buffer full condition* could be *'head = tail – 1'*, etc.)
- A hierarchical system of communication function is built, ranging from the low-level Interrupt Service Routines (ISR, access to hardware), through simple functions for the sending/receiving of one character up to functions which can print a formatted string and/or read formatted data from the interface

Buffered Serial Communication

- Interrupt driven communication is commonly based on a pair of *Interrupt Service Routines (ISR)* which are responsible for transmitting and receiving individual bytes via the serial interface
- The ISR for reception is triggered whenever a full byte has been received; the ISR fetches the byte from *S0RBUF* and places it in an input buffer
- The ISR for transmission is first triggered by setting the *S0TIR* flag in register *S0TIC* and then after every completed transmission; once the buffer is empty, the interface becomes dormant again

Buffered Serial Communication

- ISR *SerOutchar_ISR*

```
void SerOutchar_ISR(void) interrupt 0x2a {

  if(!RB_EMPTY(&out)) {

       S0TBUF = *RB_POPSLOT(&out);       /* start transmission of next byte */
       RB_POPADVANCE(&out);              /* remove the sent byte from buffer */

  }
  else TXactive = 0;                     /* TX finished, interface inactive */

}
```

The ISR first checks if the transmission buffer (out) is empty; if this is the case, static global variable *TXactive* is reset to '0' – *TXactive* is used to indicate to higher level functions that a transmission is currently underway. The ISR then fetches the next data element from the buffer (RB_POPSLOT(&out)*) and places it in S0TBUF, thereby initiating the transmission. Finally, the read pointer (tail) is advanced by one element (*RB_POPADVANCE(&out)*)**

Buffered Serial Communication

- ISR *SerInchar_ISR*

```
void SerInchar_ISR(void) interrupt 0x2b {

     if(!RB_FULL(&in)) {

     *RB_PUSHSLOT(&in) = S0RBUF;        /* store new data in the buffer */

     RB_PUSHADVANCE(&in);               /* next write location */

  }
}
```

The ISR first checks if there is a free slot on the reception buffer (*!RB_FULL(&in)* → *buffer 'in' not full);* it then fetches the received value from register S0RBUF and places it in the current write slot of the reception buffer (tail pointer, *RB_PUSHSLOT(&in)*). Finally, the write position is increased by the size of one element of the buffer – here: 'one byte' (*RB_PUSHADVANCE(&in)*)

Initialization of the serial interface

```
/* initialise interface S0 (1-N-57600) */
void serInit(void) {

  RB_INIT(&out, outbuf, 255);           /* set up TX ring buffer */
  RB_INIT(&in, inbuf, 255);             /* set up RX ring buffer */

  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  = 0x0A;                         /* program baud rate to 57600 bps */
  S0CON = 0x8011;                       /* configure serial interface */

  S0TIC = 0x49;                         /* ILVL: 2, GLVL: 1, enabled */
  S0RIC = 0x4a;                         /* ILVL: 2, GLVL: 2, enabled */
}
```

***serInit* sets up the *ring buffer admin structures (RB_INIT)* before initialising the serial interface ASC0 of the C167CR for communication at 57600 bps (8 data bits / no parity, 1 stop bit); both interrupts (transmission as well as reception) are *enabled*. Note that the priority of receptions is defined larger than that of transmissions**

Initialization of the serial interface

```
#include <string.h>                     /* strlen() */
#include "serial.h"                     /* declarations of comms routines */
#include "rb.h"                         /* ring buffer macros */

#define       MAX_BUFLEN 128
static char   outbuf[2*MAX_BUFLEN];     /* memory for ring buffer #1 (TXD) */
static char   inbuf [2*MAX_BUFLEN];     /* memory for ring buffer #2 (RXD) */
static int    TXactive = 0;             /* transmission status flag (off) */

/* define o/p and i/p ring buffer control structures */
static RB_CREATE(out, char);            /* static struct { ... } out; */
static RB_CREATE(in,  char);            /* static struct { ... } in; */
```

Two *ring buffers* have been defined, *inbuf* and *outbuf* (defined as *'static' char* – they are therefore *'global at file level'* → only the functions in file 'serial.c' know about these buffers!). The corresponding buffer admin structures, *in* and *out*, are initialized using macro *RB_CREATE*; again, the keyword *static* limts their scope to the functions defined in this file (serial.c). The data type of both buffers is *'char'* – it may increase performance to change this declaration to *'near char'* (fast 16-bit access)

Transmission of a single character

```
/* O/P : send single character */
int serOutchar(char c) {

  while(RB_FULL(&out));                 /* wait until there's space */
  *RB_PUSHSLOT(&out) = c;               /* store data in the buffer */
  RB_PUSHADVANCE(&out);                 /* adjust write position */

  if(!TXactive) {

     TXactive = 1;                      /* indicate ongoing transmission */
     S0TIC |= 0x80;                     /* start transmission
                                           - only required once */
  }

  return 0;
}
```

serOutchar **is a blocking implementation of a low-level transmission routine – a permanently full buffer will make the microcontroller 'hang' on the first line (*while(RB_FULL(&out))*; this may or may not be desirable (depends on the application). Whenever possible, the routine stores the character to be sent in the transmission buffer (out) and – if required – initiates its transmission**

Reception of a single character

```
/* I/P : get single character */
char serInchar(void) {
char c;

  while(RB_EMPTY(&in));                 /* wait for data */

  c = *RB_POPSLOT(&in);                 /* get character off the buffer */
  RB_POPADVANCE(&in);                   /* adjust write position */

  return c;
}
```

serInchar **is a blocking implementation of a low-level reception routine – a permanently empty buffer will make the microcontroller 'hang' on the first line (*while(RB_EMPTY(&in))*; this may or may not be desirable (depends on the application). Whenever possible, the routine fetches one character from the reception buffer (in) and returns it to the calling program. The write position of the reception buffer is increased by the size of one element of the buffer**

Transmission of an entire string (unformatted)

```
/* O/P : send entire string */
int serOutstring(char *buf) {
unsigned char len;

  for(len = 0; len < strlen(buf); len++)
     serOutchar(buf[len]);

  return len;
}
```

serOutstring **calls upon** ***serOutchar*** **to send the 0-terminated string stored in** ***buf;*** **the number of characters sent is returned to the calling program**

- Many other functions could be defined to establish a convenient high-level communication interface; these often include functions which emulate the behaviour of *printf* and *scanf* (formatted I/O)

Test program for the buffered serial communication

```
include <stdio.h>                       /* sprintf() */
#include <reg167.h>
#include "serial.h"                     /* S0 communication routines */


#define MY_BUF 80                       /* size of local string buffer */
static char  userbuf[MY_BUF];

static char  *error = "\n\rError: Buffer overflow.\n\r";


void main(void) {
long  x, y, z;
int   len, c, rc = 0;

  serInit();                             /* init. ASC0: 57600, 8N-1 */
  IEN = 1;                               /* allow interrupts to happen */
     (...)
```

The main program includes the header file ***"serial.h"*** **to be aware of the call-up parameters of the new communication routines; a (static) global buffer is defined for subsequent string manipulations – note that this buffer is different from the communication ring buffers!**

Test program for the buffered serial communication

```
    (...)
while(1) {

    /* display message */
    sprintf(userbuf, "\r\nPlease enter 3 values (long long long): ");
    serOutstring(userbuf);

    /* read from the terminal... */
    len = 0;
    do {

       c = serInchar();                /* get character */
       serOutchar(c);                  /* echo */
       userbuf[len++] = c;

    } while((c!='\r') && (len<MY_BUF));
    (...)
```

Standard I/O library (stdio.h) function *sprintf* is used to set-up variable *userbuf* with an welcome message which is then sent using *serOutstring*. Input function *serInchar* is used to read up to MY_BUF characters from the interface and store them in *userbuf*

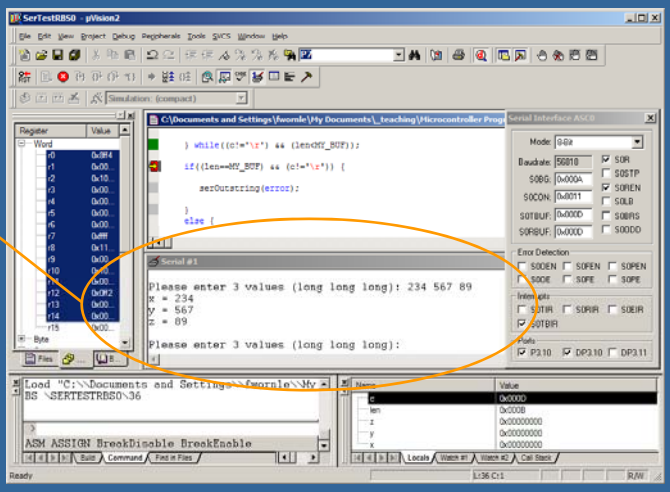
Test program for the buffered serial communication

```
    (...)
   if((len==MY_BUF) && (c!='\r')) {

      serOutstring(error);

   }
   else {

      rc = sscanf(userbuf, "%ld %ld %ld", &x, &y, &z);
      sprintf(userbuf, "\n\rx = %ld\n\ry = %ld\n\rz = %ld\r\n", x, y, z);
      serOutstring(userbuf);

   } /* if-else */
  } /* while */
} /* main */
```

An error message is printed, in case too many characters were received; otherwise, the received string *(userbuf)* is scanned to extract three *signed long integer* numbers *(x, y, z)* which are then echoed back to the terminal using *serOutstring*.
The entire process is repeated in an endless loop.

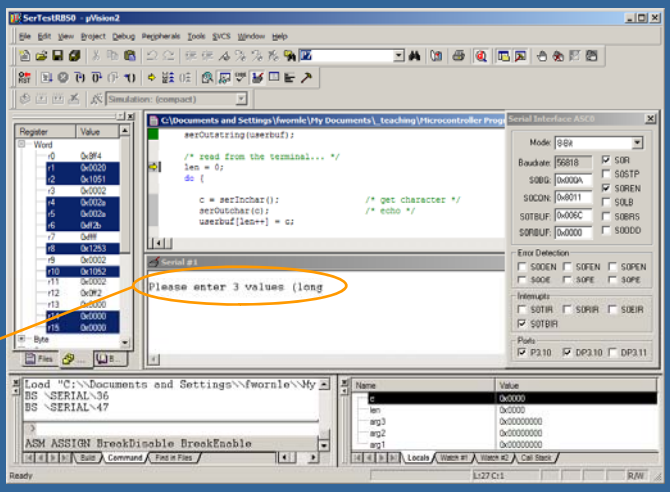
Test program for the buffered serial communication

The simulator provides a *serial coms terminal* which allows the program to be tested



Test program for the buffered serial communication

The display is handled by a backgrnd task *(note: only half of the text has been displayed so far)*



Other communication protocols: CAN [1],[2]

- The *CAN protocol* is an ISO standard (ISO 11898) for serial data communication. The protocol was originally developed by BOSCH (1986), aiming at automotive applications. Today CAN has gained widespread use and is used in industrial automation as well as in automotives and mobile machines.
- The CAN protocol has been around for more than 15 years (since 1986). There are now a lot of CAN products and tools available on the open market.
- CAN supports transmission rates of up to 1 MBit/s

Other communication protocols: CAN [1],[2]

- The error handling of CAN is one of the really strong advantages of the protocol. The error detection mechanisms are extensive, and the fault confinement algorithms are well developed.
- A faulty node within a system can ruin the transmission of a whole system, e.g. by occupying all the available band width. The CAN protocol has a built-in feature that prevents a faulty node from blocking the system. A faulty node is eventually excluded from further sending on the CAN bus.

Other communication protocols: CAN [1],[2]

- The CAN protocol is a good basis when designing *Distributed Control Systems*. The CAN arbitration method ensures that each CAN node just has to deal with messages that are relevant for that node.
- A Distributed Control System can be described as a system where the processor capacity is distributed among all nodes in a system (e.g. implemented as embedded microcontrollers). The opposite can be described as a system with a central processor and local I/O-units.

Other communication protocols: CAN [1],[2]

The CAN protocol is defined by the ISO 11898-1 standard and can be summarized as follows:

- The physical layer uses differential transmission on a twisted pair wire
- A *non-destructive bit-wise arbitration* is used to control access to the bus
- There is *no explicit address in the messages*, instead, each message carries *a numeric value which controls its priority on the bus*, and may also serve as an identification of the *contents* of the message

Other communication protocols: CAN [1],[2]

The CAN protocol is defined by the ISO 11898-1 standard and can be summarized as follows:

- The messages are small *(at most eight data bytes)* and are protected by a checksum
- An *elaborate error handling scheme* that results in retransmitted messages when they are not properly received
- There are effective means for isolating faults and removing faulty nodes from the bus

Other communication protocols: CAN [1],[2]

- The CAN protocol itself just *specifies how small packets of data safely may be transported from point A to point B* using a shared communications medium. It (quite naturally) contains nothing on topics such as flow control, transportation of data larger than can fit in a 8-byte message, node addresses, establishment of communication, etc. These topics are covered by a *Higher Layer Protocol (HLP)*. The term HLP is derived from the *Open Systems Interconnection model (OSI)* and its seven layers. Popular HLP for CAN are *CANopen*, *DeviceNet*, *SAE J1939*, etc.

Other communication protocols: CAN [1],[2]

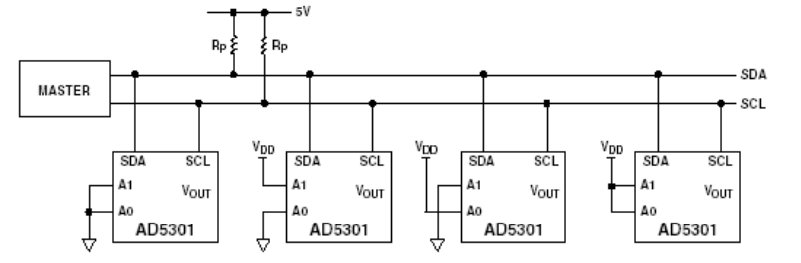
- CAN technology is by far more complex than simple serial communication through the RS-232 (V24) interface of the microcontroller
- However, once implemented, CAN provides an application with a by far more reliable means of communication
- Most modern microcontrollers support CAN; the setting-up of the interface is often controlled by memory mapped Special Function Registers (SFR)
- For more information about CAN: see reference [2]

Other communication protocols: IIC, I^2C

- The *Inter-IC bus* (IIC, I^2C) was originally developed (early 1980s) by Philips Semiconductors
- The I^2C bus is widely used in embedded systems to interface the microcontroller with peripherals such as external A/D or D/A converters, memory chips, intelligent sensors and/or other microcontrollers
- Technically, it is a bidirectional 2-wire bus system, designed for simple but efficient control applications; all devices on the bus can be configured as *master* and/or as *slave;* the two lines are *SCL (serial clock)* and SDA (serial data)

Other communication protocols: IIC, I²C

- Example: Analog Device D/A converter AD5311



- The AD5311 is a slave device which can be accessed through an I²C bus interface; up to 4 AD5311s can be operated in parallel on the same I²C bus segment

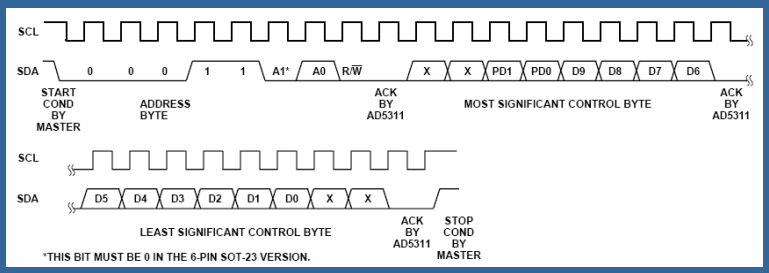
Other communication protocols: IIC, I²C

- Up to 128 devices can be connected to a single I²C bus segment; prior to transmitting the actual data, one of these devices has to be selected – this is done by sending a 7-bit address
- The AD5311 is a 10-bit D/A converter (DAC); its small package only hosts two address lines, i. e. only 2 of the 7 address bits are specified by the designer – the others *5 bits have been fixed* by Analog Devices
- Example:

```
00011----0------0-------0 = 0001.1000 = 0x18
(fixed)  (A1)   (A0)   (R/W)
```

Other communication protocols: IIC, I²C

- Following the address byte is a short pause during which the selected device is expected to send an acknowledgement bit
- Finally, the actual data word is sent; the number of bits depends on the connected device (here: 16)



Other communication protocols: IIC, I²C

- I²C communication is supported by many micro-controllers; frequently, it suffices to set-up a number of Special Function Registers to get communications going
- The maximum line speed of the I²C bus is 400 kBit/s
- Compared to the *Universal Asynchronous Receiver-Transmitter (UART)* interface, I²C does not allow for *full-duplex* communication (there is only one data line, i. e. bidirectional transmissions can not happen simultaneously)

Other communication protocols: Ethernet

- Today, an increasing number of embedded systems are *web-enabled*, i. e. they are compliant with the *IEEE Standard for LAN/MAN (IEEE 802.3)* [3]
- These systems connect to a regular *10/100Base-T Ethernet* backbone network using an on-chip *Ethernet Media Access Controller (EMAC)* or similar external transceiver ICs
- Web-enabled software implements a small *TCP/IP protocol stack* (*Transmission Control Protocol / Internet Protocol)* allowing an application to communicate with servers and/or clients on the net

Other communication protocols: Ethernet

- Industrial automation has evolved in stages:
  - Isolated PLCs
  - Interconnected PLCs (digital data lines)
  - PLCs and PCs are connected in simple point-to-point networks (RS-232, RS-422, RS-485, etc.)
  - After 1984: bus hierarchies, *field bus systems* (BITBUS, PROFIBUS, I^2C, CAN, TCP/IP, …)
  - Most recent trend: *Middleware* – component based software architecture which isolates an application from all platform specific details and thus provides manufacturer independent services

Other communication protocols: Ethernet

- Ethernet-TCP/IP has played an important/dominant role in industrial automation as it represents a world-wide accepted standard for data communication networks based on the concept of client/server applications
- The widespread use of Ethernet-TCP/IP will carry over to embedded systems engineering; many more systems and applications will come to the market which exploit the ever-increasing performance of Ethernet-TCP/IP (high speed, wireless, security)

Other communication protocols: USB

- The advent of the *Universal Serial Bus (USB)* has had an enormous impact on peripheral devices in the computer industry (printers, mice, keyboards, hard-disk drives, cameras, etc.); in microcontroller based embedded systems USB is only beginning to appear
- An increasing number of microcontrollers features a *USB device interface*; this is required to communicate with a *USB host* such as a personal computer (PC)
- Today, only a few microcontrollers include a full *USB host and device interface*, e.g. the *Cypress CY7C6200/300* or the *Freescale MPC850 (PowerPC)*

Further reading:

[1]
KVASER, Advanced CAN Solutions,
www.kvaser.com/can, accessed: January 2005

[2]
Konrad Etschberger, *Controller Area Network*,
IXXAT Automation GmbH, 2001
ISBN: 3000073760

[3]
IEEE 802.3 CSMA/CD (ETHERNET),
www.ieee802.org/3, accessed: January 2005