**CHALMERS**
Department of Computer Science and Engineering
2011-11-07

# Microcomputer systems Examples of communication interfaces

## Introduction

With the word communication we mean the transport of information from a transmitter to at least one receiver and often transport in the other direction too, sometimes at the same time.
Traditionally communication interfaces have been about interconnections without that much intelligence which means that many of the standards describe how to physically interconnect devices and does not handle the format of the information transmitted. These physical characteristics might involve changes in voltage levels or transformation from balanced to unbalanced signal or the other way around. The protocol of the transfer is not part of the interface but we use our application programs to generate this by setting the bit rate, define how bits and words should look and so on maybe with the use of prewritten standard modules. This means that it is not that hard to get at least some simple communication going even if we don´t live up to the full standard of the transfer.
In modern interfaces much of the intelligence of the transfer has moved from the application program to the interfaces and they have become much more complicated. In this cases there is normally no way to establish a communication channel without following the standards to the point and letting the interfaces establish their communication link before the transfer can take place. These new interface standards like **USB** and **Bluetooth** have increased the capabilities of the interfaces tremendously but it has at the same time taken away some of the control of the transfer from the programmer and we have to rely on the circuits and firmware in the interfaces.
In this paper we will focus on the older, less intelligent, types of communications interfaces. We shall see that the interfaces are quite simple and if we like we can use them for communication channels that don´t follow any standard protocol at all, the exceptions to this are **GPIB**, the **I2C** and one wire interfaces at the end of the paper.
The transport in a communication channel might be in only one direction or in both directions, that is it might be *uni- or bidirectional*. If we have communication in only one direction we talk about a *simplex* channel. If the communication can go in both directions but not at the

same time we have *half duplex* and if the communication can take place in both directions at the same time then we have (*full*) *duplex*.
The communication might take place between only two units, one transmitter and one receiver, or if the communication is bidirectional two *transceivers* (short for transmitter/receiver). In this case we talk about *peer to peer* communication. If the transmission is from one transmitter to several receivers we talk about *multicast*. In this case the communication is in most cases only in one direction, which means that we have one sender and many receivers. A peer to peer communication could involve more than two devices but then only two devices are active at any one time.
Another situation arises if we have a number of units connected together trough a common network, we talk about a *bus topology*. In its basic form of bus communication one unit transmits a message containing some kind of address and the other units are listening on the bus and if the listening unit have the same address as the transmitted address it will accept the reception of the message while the other units will ignore it. We can have a number of variations to this. The address does not have to be the address of a specific unit but can instead be a header indicating what kind of data that will be transmitted and then all units that have interest in this kind of data will accept and receive it. We will see an example of this when we talk about the **CAN** (Controller area network) interface later on.
Obviously only one unit can be talking at any given time on a bus network so the access to the bus needs to be controlled in some way. In some protocols we have a *master* unit which will always initiate the communication by sending the start message and this might be a command to other units or a request for an answer from some unit. In this case the other units are called *slaves* and they will only respond to requests from the master and not start any conversation on their own. In some cases they can request attention from the host though. In other protocols more than one unit can initialize a conversation and in those cases we talk about *multi master* protocols.
In all types of bus communication we need a way to decide which of the unit that is allowed to talk at any given time. This can be done in several ways. One way is to let all units talk in turn, we pass a *token* around and when a unit has the token it may speak on the bus. We have a *token ring*. This is quite simple but it is inefficient if the needed transmission rate differs between the different units on the bus. In this case we would vast a lot of time slots passing the token around to units that have nothing to transmit.
In other bus topologies we don´t allocate separate time slots to the different units but we let them talk when they have something to say. This calls for some way to decide which unit that will be allowed to speak while the other unit(s) back off. We have what is called an *arbitration* process. This is for example typical for **CAN** networks that are quit common in automotive applications. We will have a brief look at this later.
Another distinction we need to do when we talk about communication is how the physical transportation layer looks. If we base or communication on data of a given length, for example a byte, we could transmit all the bits at the same time through different, parallel wires or bit by bit on one single wire. In the first case we talk about *parallel communication*, *Figure 1*, while we in the other case have *serial communication*, *Figure 2*. It

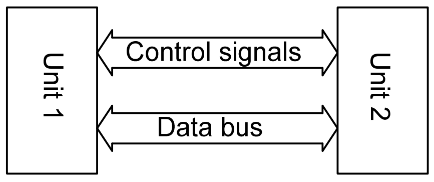

*Figure 1 Parallel communication*

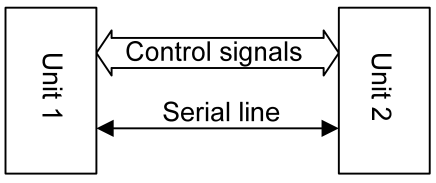

*Figure 2 Serial communication*

might seem like the parallel connection would always be a better choice since we can deliver all the bits at the same time and this would be faster than a serial communication channel. This might be true but we have some problems associated with parallel protocols.

Modern equipment becomes more and more complicated and it gets harder and harder to find room within the integrated circuits and on the printed circuit boards for the parallel data lines and since the data words are getting larger the number of lines increase. Things are being even more complicated by our constant effort to increase the speed of the data transfer. We have now reached speeds where we have to take in account the time it will take for the information to pass though a wire from one unit to another and if the parallel wires are not of exactly the same length the bits in the word will reach the receiver at slightly different times and we have a great risk of reading false data, *Figure 3*. This uncertainty in the arrival time of different bits is called *skew*.

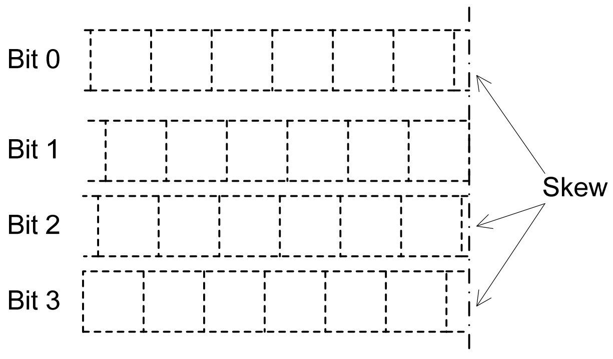

*Figure 3 Misreading caused by skew*

If we use serial communication the routing of wires will be simpler and we don´t have any skew between bits, they come one by one. For this reasons almost all modern fast communication protocols are serial.

In the serial case we still have a risk of misreading the received data if we read it at the wrong time. We need a way to synchronize the transmitter and the receiver. One way to do this is to send a synchronization signal, a *clock* signal, on a separate wire, but then we need this extra wire. Another way is to code the information bits in a way that will give a pulse edge in every bit that the receiver can trigger on, see **Synchronous serial transfer** later on. This would be a somewhat more complicated protocol. In both cases we talk about *synchronous communication* where both transmitter and receiver use the same synchronization signal. Later we will have a look at the **SPI** protocol (Serial peripheral interface) and the Inter-integrated circuit protocol **I2C**. Both protocols use a separate clock line, although I2C is a bus interface.

Another way is to let transmitter and receiver use their own internal clocks to decide the transmission speed and thereby decide when to transmit a bit and when to read the received bit respectively. We have an *asynchronous protocol*. Since we cannot be sure that the clocks in the two units run at exactly the same speed and that the phasing is the same we have to keep the transmission rate lower than in the synchronous case so the two clocks don´t drift that far apart and we have to have some kind of synchronization between the transmitter and the receiver to set the timing of the transfer but this synchronization will not take place on every bit in the transfer. One common way to do this is to add extra bits to the transmission of every message for synchronisation. In the **SCI** protocol (Serial communications interface), that we will look at later, the synchronization is done by adding a *start bit* at the beginning of every byte that is transmitted to trigger and synchronize the start of the transfer.

When we talk about serial communication we need to separate unbalanced and balanced wire links. This holds true for both analog and digital signals.

In the *unbalanced* case we use only one wire and the transmitted signal is a voltage referenced to ground, *Figure 4*. The voltages for both signal levels can be

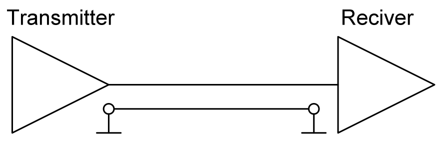

*Figure 4 Unbalanced link*

separated from 0 Volts.
In the *balanced* case we use two wires and the signals are represented by the voltage difference between these two wires and we have no reference to ground, *Figure 5*.
In noisy environments or when the distance between the transmitter and the receiver is long the balanced, differential approach is to prefer. Let´s explain why.
If an unbalanced signal is disturbed by interference, noise, this will be an extra voltage that will be added to the transmitted signal voltage and we have the risk of misreading the received signal, *Figure 6*. On the other hand if we use balanced transmission the disturbance will most likely affect both wires in the same way if they are placed close together and the voltage difference between the wires, which is the important voltage in this case, not the voltage levels as such, are only slightly affected by the disturbance and the received bit will still be read correctly, *Figure 7*.

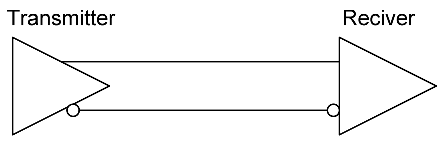

*Figure 5 Balanced link*

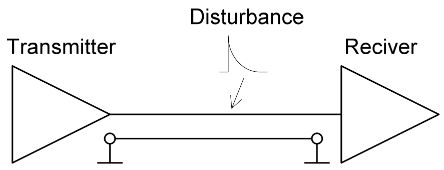

*Figure 6 Unbalanced link with disturbance*

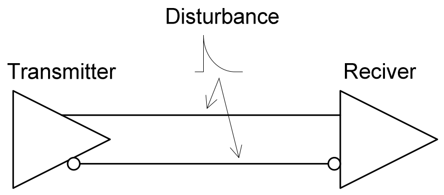

*Figure 7 Balanced link with disturbance*

One thing that is used here and there in communication is *modems*. Modem is short for modulator/demodulator and is a device that is used to transform our binary bits into a suitable form for the transmission channel and then back again. An example is the telephone modem. These are used for digital communication over telephone lines. An ordinary telephone line can transfer signals with frequencies in the band 200 Hz to 3.3 kHz and in the simplest form of telephone modem the '1':s and '0':s are converted into two different tones that fit into this frequency range. To make duplex communication possible we use a total of four frequencies, two for the communication in one direction and the other two for the communication in the other direction.
*Table 1* show the frequencies used in the modem standard V.21. By using more frequencies and/or using phase information we can transmit more than one data bit with each transmitted symbol and thereby increase the transmission speed or we can communicate over several channels at the same time.

| *Channel* | *One [Hz ]* | *Zero [Hz]* |
|---|---|---|
| 1 | 980 | 1180 |
| 2 | 1650 | 1850 |

*Table 1 Frequencies in the modem standard V.21*

So far we have mostly talked about wires but there is nothing stopping us from using other transmission media like light (optical fiber) or radio waves. In these media we normally modulate our information on a carrier wave, a fixed high frequency signal and this will give a serial channel even if we as stated earlier can use methods to embed more than one bit of information into each transmitted character. If we use so called spread spectrum for the transfer we can use a whole set of carrier waves and thereby transmit more than one bit at the same time, we send one bit per carrier wave. This is for example used in digital radio (**DAB**, Digital Audio Broadcast).
We will now move on to some examples of communication protocols starting with two parallel protocols and then focus on serial solutions.

# Examples of  parallel communication protocols

We will have a look at two examples of parallel interfaces. The first one is an example of a parallel interface between a microcontroller and external memory chip while the other describes the **GPIB** bus, a bus widely used to interconnect intelligent measuring instruments in a network together with a controller.

## Parallel memory interface

A memory interface on a microcontroller is used to expand the available amount of memory by adding external memory devices. This could be done through a parallel or a serial interface. In the serial case we will in most cases use a synchronous serial interface (**SPI**) or an **I2C** interface. We will get back to these interfaces later on.
For the moment we will have a look at the parallel interface. As the description implies we use a parallel approach, that is the data bits are presented at the same time, on separate wires in a data bus. This is not enough though because we need to select the address in the external memory to read from or write to, that is we need a parallel address bus too and finally we will need some control wires.
Let´s look at the way the microcontroller **HC12** from **Freescale** addresses external memory as an example. The **HC12** has a number of different addressing methods when it comes to addressing external memory, we will just mention two of these. Both of these methods can be used when the processor is in emulation mode where some of the internal operations of the processor are emulated externally. In this mode the external bus is configured out of reset with the bus control signals enabled. We have two different emulation modes

- *Emulation expanded wide* where we have a 16 bit wide address bus and a 16 bit wide data bus
- *Emulation expanded narrow* where we have 16 bit wide address bus and a 8 bit wide data bus

We will use the latter in our example. In both cases the 16 address lines are connected through **PORTB** (**A0**-**A7**) and **PORTA** (**A8**-**A15**) in the processor while the data bus uses the same two ports in wide mode and just **PORTA** in narrow mode. As we can see the data and address bus shares the same port(s) and to make this possible the address and data lines are active during separate parts of a cycle of reading data from or writing data to external memory. During the first part of a read/write cycle the address bus is active and during the latter part the data bus is active. We say that the bus is *multiplexed*. Now the selected address in the memory chip will be addressed during the first phase of the clock cycle but when we get to the read/write phase this address still needs to be active and address the external unit so we need to use some external logic to hold, remember, the address during this second phase, the data phase.
Let´s as an example see how we can use the emulation expanded narrow mode to address an 8K big static RAM memory called **6264**. The memory has a 13 bit wide address bus for 8K of data and an 8 bit wide data bus. There are four more control signals to the memory

- One active low output enable signal **/OE** that we use when we want to read data from the memory
- One active low write enable signal **/WE** that we use when we want to write data to the memory
- Two active high chip select signals **CS1** and **CS2**

From the processor we will use two signals besides the data and the address bus

- The read/write signal **R/W**, where high level indicates read phase
- The E clock **ECLK** which is low during the address phase and high during the data phase

We will use some logic together with these two signals to create the necessary control signals for the memory. In the emulation mode we use the lower 16K of the address space to address internal memory while we have 48K left for external memory. The address lines **A15**-**A14** will split the total address space into four slots (value `00` for the internal memory). Since our 8K memory will only fill half of one of these external slots we will use **A15**-**A13** and at first some logic and then a 3/8 decoder to place the memory in the address space. Let´s place it at the start address `0xC000`. This means that **A15**-**A13** should have the value 110 to activate the memory (high signal) and using NAND-logic we get

$$CS1 = A_{15}\cdot A_{14}\cdot\overline{A_{13}} = \overline{\overline{A_{15}\cdot A_{14}}}\cdot\overline{A_{13}}$$

If we like we can use the decoder to place other memories in other slots. Now let us create the control signals. Let us start with the **/WE** signal. We can realize that this signal should be active (low) in the data phase (**ECLK** high) if the **R/W** signal is low. We will get the truth table in *Table 2* and the logical expression

| *ECLK* | *R/W* | */WE* |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*Table 2 Truth table for the **/WE** signal*

$$\overline{WE} = \overline{ECLK\cdot\overline{R/W}}$$

The output enable signal **/OE** should be active (low) in the data phase and when the **R/W** signal is high. We will get the truth table in *Table 3* and the logical expression

| *ECLK* | *R/W* | */OE* |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*Table 3 Truth table for the **/OE** signal*

$$\overline{OE} = \overline{ECLK\cdot R/W}$$

We can draw a full schematic of the memory interface, *Figure 8* and *Figure 9.*

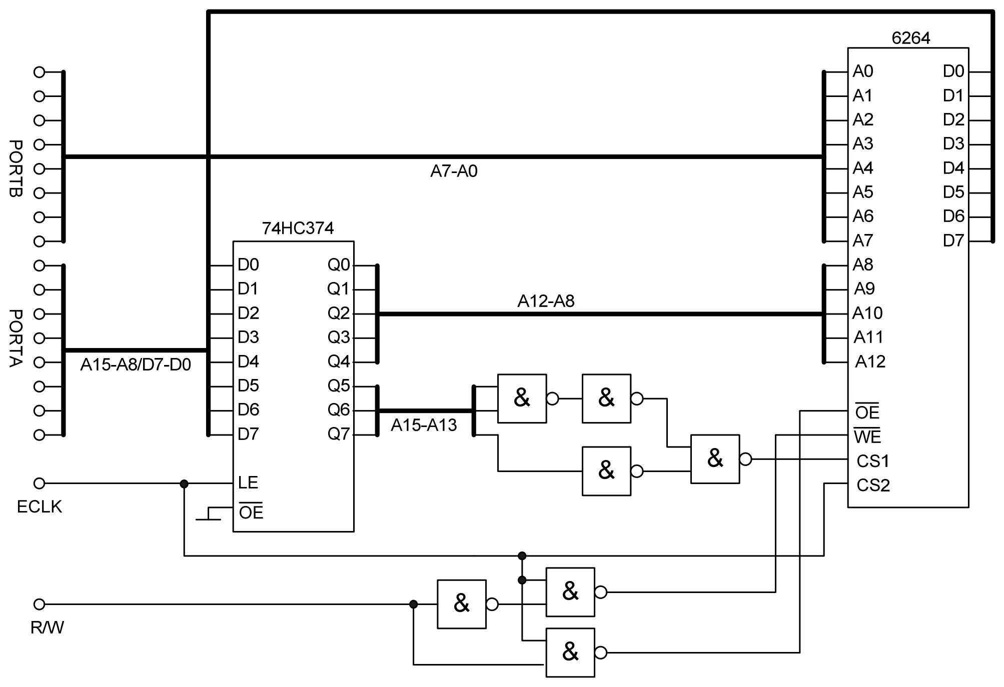

*Figure 8 Parallel external memory interface using discrete logic for the address decoding*

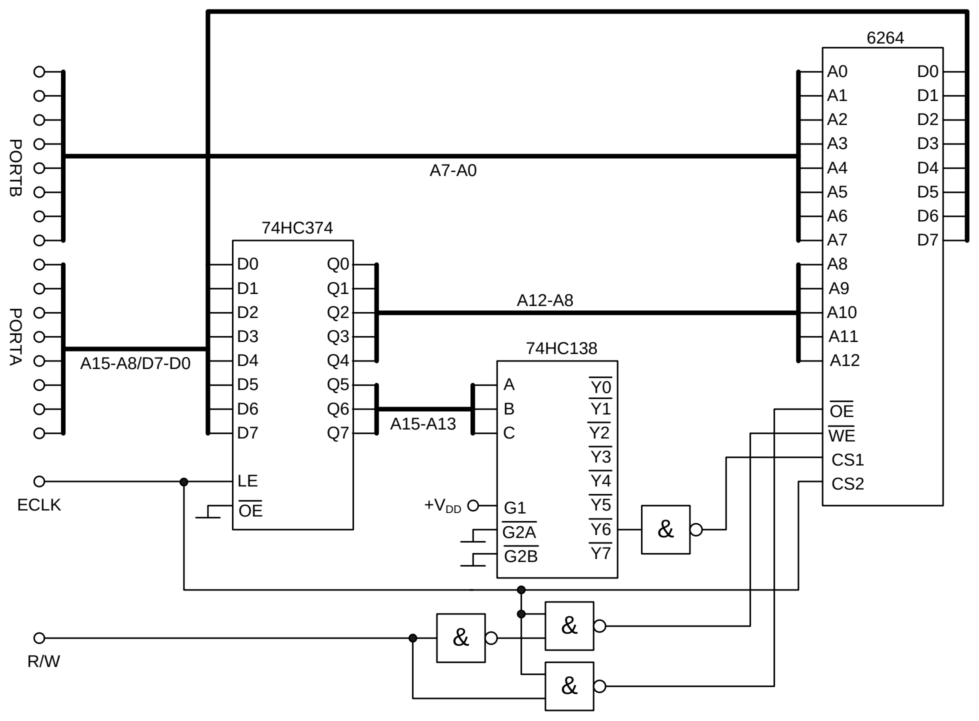

*Figure 9 Parallel external memory interface using an 3/8 decoder for the address decoding*

In *Figure 8* we have used discrete logic for the address decoding and we get a somewhat simpler schematic. In *Figure 9* we have used a 3/8 decoder for the address decoding. Using this circuit we can use the same decoding to incorporate more memory circuits into the schematic. We just activate the other memories using some of the other outputs from the decoder.
In the expanded wide mode where we have 16 data lines multiplexed with the 16 address lines the main difference from the description above will be that we need to remember the values of all 16 address lines during the data phase.

# GPIB or IEEE-488

**GPIB** (General Purpose Instrumentation Bus) is an 8 bit wide parallel communication bus developed primarily for the connection of programmable measurement instruments. It was originally developed by Hewlett-Packard who named it **HP-IB** (Hewlett-Packard Instrument bus) but when it got standardized it was given its present name. The bus has been standardized by the US organization **IEEE** (Institute of Electrical and Electronics Engineers) as standard **IEEE-488**. The standard has later evolved to the standard **ANSI/IEEE488.1.** In Europe the bus has been standardized by **IEC** (International Electrotechnical Commission) as standard **IEC-625**. Later on the standard **ANSI/IEEE488.2** defined how controllers and interfaces communicate. **SCPI** (Standard Commands for Programmable Instruments) took the command structure from **ANSI/IEEE488.2** to create a comprehensive command set that can be used to program any **SCPI** instrument no matter the brand.
Three types of devices can be connected to the bus: *controllers*, *talkers* and *listeners*. Some devices may have more than one of these functions. Up to 15 devices can be connected to the bus. Each device is assigned a unique primary address ranging from 0-30. A secondary address may also be specified in the same range.
A control system can in its minimum configuration consist of one controller and one talker or listener. The *controller* controls the traffic on the bus. There may be more than one controller connected to the bus but only one of them can be active at any one time. One of the controllers has the head role of system controller. A *listener* is a device that receives data from the bus when so instructed by the controller. A *talker* puts data one to bus when so instructed by the controller.
The data transfer rate in standard **GPIB** can be up to 1.8 Mbyte/second. There is a newer high speed standard, **HS488**, that can use data transfers up to 8 Mbyte/second.
The physical interface in **GPIB** consists of 16 signal lines and 8 ground lines, *Table 4*. The signal lines are divided into three groups: 8 data lines (each of which can be shielded by one ground line), three handshake lines and five interface management lines.
The data lines **DIO1** – **DIO8** can transfer addresses, data and control information. **DIO1** is the least significant bit.
The three handshake lines control the transfer over the bus and are used to acknowledge the transfer of data.

- The **NRFD** (Not Ready for Data) line is asserted by a listener to indicate that it is not yet ready for the next data or control byte
- The **NDAC** (Not Data Accepted) line is asserted by a listener to indicate that it has not yet accepted the data or control byte on the data lines

- The **DAV** (Data Valid) line is asserted by the talker to indicate that a data or control byte has been placed on the data lines and can now safely be accepted by other devices

The five interface management lines manage the flow of data and control bytes across the interface.

- The **ATN** (Attention) signal is asserted by the controller to indicate that it is placing a address or control byte on the data bus
- The **EOI** (End or Identify) signal has two functions. A talker may assert the line simultaneously with the last data byte to indicate end of data. The controller may assert **EOI** along with **ATN** to indicate a parallel poll
- The **IFC** (Interface Clear) signal is asserted by the system controller to initialize all device interfaces to a known state. After releasing **IFC** the system controller is the active controller
- The **REN** (Remote Enable) signal is asserted by the system controller. **REN** enables a device to go into remote mode when addressed to listen. In remote mode the device should ignore its local front panel controls
- The **SRQ** (Service Request) signal is an interrupt signal. It may be asserted by any device to request the controller to take some kind of action

| *Pin* | *Abbreviation* | *Name* |
|---|---|---|
| 1 | DIO1 | Data input/output bit 1 |
| 2 | DIO2 | Data input/output bit 2 |
| 3 | DIO3 | Data input/output bit 3 |
| 4 | DIO4 | Data input/output bit 4 |
| 5 | EIO | End or Identify |
| 6 | DAV | Data Valid |
| 7 | NRFD | Not Ready for Data |
| 8 | NDAC | Not Data Accepted |
| 9 | IFC | Interface Clear |
| 10 | SRQ | Service Request |
| 11 | ATN | Attention |
| 12 | | Shield |
| 13 | DIO5 | Data input/output bit 5 |
| 14 | DIO6 | Data input/output bit 6 |
| 15 | DIO7 | Data input/output bit 7 |
| 16 | DIO8 | Data input/output bit 8 |
| 17 | REN | Remote Enable |
| 18 | | Shield |
| 19 | | Shield |
| 20 | | Shield |
| 21 | | Shield |
| 22 | | Shield |
| 23 | | Shield |
| 24 | | Single GND |

*Table 4 24 pin connector used by GPIB*

The devices can be connected in either a linear configuration, *Figure 10*, in a star configuration, *Figure 11* or in a combination of the two, *Figure 12*, using a shielded 24 conductor cable. The maximal separation between two devices is 4 meters while the maximal total cable length is 20 meters.

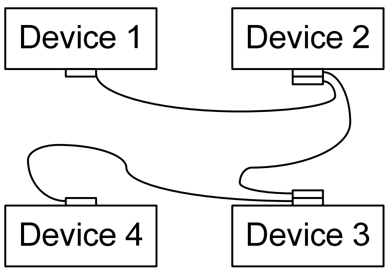

*Figure 10 Linear GPIB configuration*

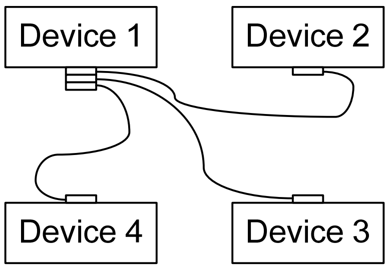

*Figure 11 Star GPIB configuration*

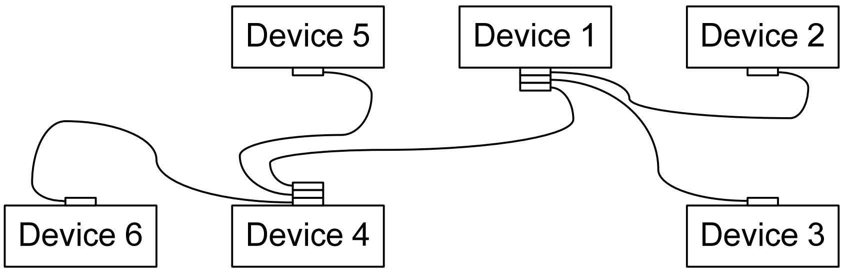

*Figure 12 Combination of linear and star GPIB configuration*

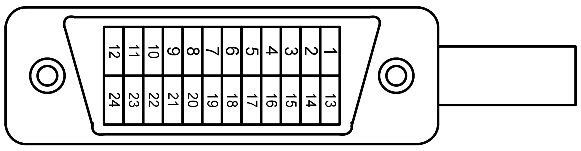

*Figure 13 GPIB connector*

The interconnection use a chunky connector with both a male and a female side which means that connectors can be stacked on top of each other, *Figure 13*.
The bus uses standard TTL logic levels with a negative logic meaning that a one (1) will give a low TTL level while a zero (0) will give a high TTL level.
The protocol and its connectors might seem old and bulky but is still used to a wide extent since there are many instruments out there equipped witch GPIB interfaces. There is a trend away from using GPIB instruments towards using instruments equipped with USB, Firewire or Ethernet interfaces. There are adapters that convert between the GPIB interface and the more modern interfaces so that you can keep your older instruments with GPIB and control them using the newer interfaces.

# Examples of serial communication protocols

## RS-232

One of the most common interface standards for data communication is **EIA**´s Recommended Standard 232C (**RS-232-C**). EIA is an abbreviation of *Electric Industries Association* representing many manufacturers in the US electronics industry. **RS-232-C** is a standard that defines the electrical characteristics of signals for serial computer communication. It defines how ‘1’:s and ‘0’:s should be electrically transmitted, including the voltage levels needed as well as the other signal characteristics necessary in the communication but it doesn´t define the communication protocol used in the transfer. Since it only gives the electrical characteristics it can be used for different protocols, both synchronous and asynchronies, even if we mostly associate it with asynchronous communication.

**RS-232-C** is an unbalanced protocol where ones (1) and zeros (0) are transmitted using negative and positive voltages. A one (1), a *mark*, is represented by an electrical signal between -3 and -15 Volts (often -12 Volts). A zero (0), a *space*, is represented by an electrical signal between +3 and +15 Volts (often +12 Volts). Signals outside these ranges are considered undefined and are ignored. Since these voltages differ from the usual 0 and +5 Volts or 3.3 Volts levels seen inside computers the communication interface must contain means to convert from 0 and +5 Volts (or +3.3 Volts) to the **RS-232-C** levels and back again. The maximal distance between transmitter and receiver is 15 meter. The protocol can use half or full duplex.

| *Pin* | *Abbriviation* | *Name* | *Direction* |
|---|---|---|---|
| 1 | GND | Protective ground | Both ways |
| 2 | TD | Transmitted data | TDE to DCE |
| 3 | RD | Received data | TCE to DTE |
| 4 | RTS | Request to send | TDE to DCE |
| 5 | CTS | Clear to send | TCE to DTE |
| 6 | DSR | Data set ready | TCE to DTE |
| 7 | SG | Signal ground | Both ways |
| 8 | DCD | Data carrier detect | TCE to DTE |
| 9 |  | Positive test voltage | TCE to DTE |
| 10 |  | Negative test voltage | TCE to DTE |
| 11 |  | Unassigned |  |
| 12 | SDCD | Secondary data carrier detect | TCE to DTE |
| 13 | SCTS | Secondary clear to send | TCE to DTE |
| 14 | STD | Secondary transmitted data | TDE to DCE |
| 15 | TC | Transmit clock | TCE to DTE |
| 16 | SRD | Secondary received data | TCE to DTE |
| 17 | RC | Receive clock | TCE to DTE |
| 18 |  | Unassigned |  |
| 19 | SRTS | Secondary request to send | TDE to DCE |
| 20 | DTR | Data terminal ready | TDE to DCE |
| 21 | SQ | Signal quality detect | TCE to DTE |
| 22 | RI | Ring indicator | TCE to DTE |
| 23 | DRS | Data rate select | Either way |
| 24 | XTC | External transmit clock | TDE to DCE |
| 25 |  | Unassigned |  |

*Table 5 25 pin DSUB connector for RS-232-C*

| *Pin* | *Abbriviation* | *Name* | *Direction* |
|---|---|---|---|
| 1 | DCD | Data carrier detect | TCE to DTE |
| 2 | RD | Received data | TCE to DTE |
| 3 | TD | Transmitted data | TDE to DCE |
| 4 | DTR | Data terminal ready | TDE to DCE |
| 5 | SG | Signal ground | Both ways |
| 6 | DSR | Data set ready | TCE to DTE |
| 7 | RTS | Request to send | TDE to DCE |
| 8 | CTS | Clear to send | TCE to DTE |
| 9 | RI | Ring indicator | TCE to DTE |

*Table 6 9 pin DSUB connector for RS-232-C*

In **RS-232-C** we define two types of interfaces, the data terminal equipment (**DTE**) which uses the reception pin (**RD**) as input and the transmission pin (**TD**) as output and the data communication equipment (**DCE**) which uses the pin the other way around. The **DTE**

should have male connectors while the **TCE** should have female connectors. The **RS-232-C** definition does not specify the type of connector to be used but in many cases 25 pin DSUB connectors are used for the full implementation (*Table 5*) while 9 pin DSUB connectors could be used if we leave out some of the rarely used signals (*Table 6*) and the latter is the most common case. Unfortunately the interface doesn´t contain any power line so we cannot supply power to external devices through the interface.
We are not going to go through all of the signals in the interface but we can see from *Table 5* that the full implementation includes pins for clock transfer (pin 15, 17 and 24) which means that it can be used for synchronous communication. These signals are missing from the 9 pin version in *Table 6* which means that this implementation can only be used for asynchronous transfer.
In the simplest form of SCI communication where we don´t use any handshaking signals we only need three lines **RD**, **TD** and ground (**SG**).
If we are to connect two interfaces of the same kind (two **DTE** units) some of the signals has to be *twisted*, that is go between different pins in the two connectors, *Figure 14* and *Table 7*. The other signals are connected to the other DTE in the same way as in *Table 6,* that is RD and TD are twisted.

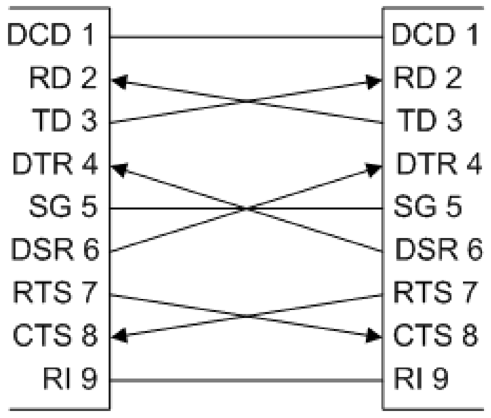

*Figure 14 Connecting two DTE units to each other*

| *DTE1* | | *DTE2* | |
|---|---|---|---|
| *Pin* | *Abbriviation* | *Pin* | *Abbriviation* |
| 1 | DCD | 1 | DCD |
| 2 | RD | 3 | TD |
| 3 | TD | 2 | RD |
| 4 | DTR | 6 | DSR |
| 5 | SG | 5 | SG |
| 6 | DSR | 4 | DTR |
| 7 | RTS | 8 | CTS |
| 8 | CTS | 7 | RTS |
| 9 | RI | | RI |

*Table 7 Connecting two DTE units to each other*

In a *zero modem* two DTE units are connected to each other and the signals are connected in a way that makes the two unit think that they get response from another unit while all communication are done through software.. To do this we need to connect some of the outgoing signals from a unit back into inputs. We need to this at both ends of the communication line, *Figure 15* and *Table 8.*

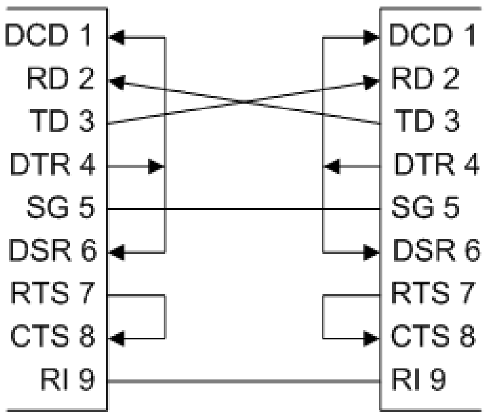

*Figure 15 Zero modem connection*

| *Pin* | *Output* | *Pin* | *Input* |
|---|---|---|---|
| 4 | DTR | 1 | DCD |
| | | 6 | DSR |
| 7 | RTS | 8 | CTS |

*Table 8 Zero modem connection*

Let´s look at the most common use of **RS-232-C**, asynchronous communication, often called **SCI** (Serial Communication Interface).

# Asynchronous serial communication, SCI

The transfer will need a clock signal. The clock frequency will give the period for each digit in the transfer. This rate is measured in digits per second or *Baud*. Since we can use clever coding to transmit more than one bit of information in each digit the actual number of bits transferred each second, the symbol rate, may be greater than the Baud rate. Typical Baud rates are 9600, 38400 and 115200 Baud although the RS-232-C standard sets the speed limit to 20 kbps.
In asynchronous communication no common clock signal is transferred between the two interfaces which mean that each interface has to have its own clock. Since these two clocks are not absolutely stable but may drift somewhat in frequency and may have different phases we need some way to synchronize the two units. We do this by starting each transmitted word with a *start bit* that will retrigger the receiver’s clock.
In rest when there is no transmission the level on the transmission line is high (‘1’ typically -12 Volts) so the start bit consist of one clock interval of low level (‘0’ typically +12 Volts). After that we send the data bits starting with the least significant bit (LSB). The number of data bits may be from five to eight bits.
When all these data bits have been sent there might come a parity bit which we will get back to soon.
Finally we transmit *stop bits* which in reality is a return to the high, idle level. We can specify the number of stop bit to be 1, 1.5 or 2 bits. This means that we have to wait this number of clock cycles before we start sending the next word my sending a new start bit.
There is always a risk of errors in the transfer so there would be a good idea to have a system to detect, or even better correct errors. The simplest way to detect errors is to use a *parity bit*. We can have four different types of parity bits. With *odd parity* we use this bit to make sure that the number of ones (1) in the data word, including the parity bit, is odd. That is if the number of ones (1) in the data word is odd we set the parity bit to zero (0) and if the number of ones in the data word is even then we set the parity bit to

one (1). In *Figure 16* we see the transfer of the **ASCII** code 65 (0x41), which is the letter ‘A’ using an eight bit word with odd parity. **ASCII** stands for *American Standard Code for Information Interchange*.
*Even parity* works in the same way but we use the parity bit to make sure that the number of ones (1) in the data word, including the parity bit, is even. In *Figure 17* we see the same example as in *Figure 16* but using even parity.
With odd or even parity we can detect, but not correct an odd number of errors in the transfer.
Sometimes a parity bit is used but it is always set to zero (0). We call this *space parity, Figure 18*. In this case an error, a one in this bit would indicate an error in the transmission.
In the same way we sometimes use a parity bit that is always set to one (1). We call this *mark parity, Figure 19*.
If we are not using any parity bit and this bit is omitted from the transfer we say that we have *no parity, Figure 20*.

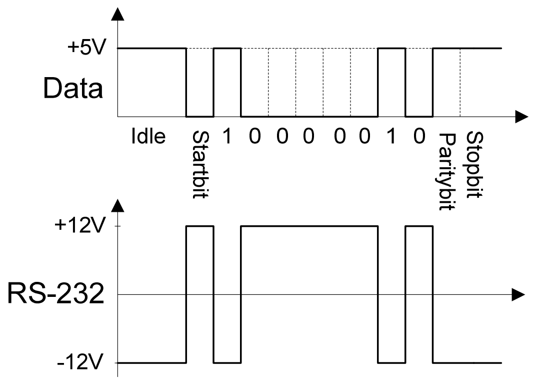

*Figure 16 Coding of the letter ’A with odd parity’ in computer and on RS-232 link*

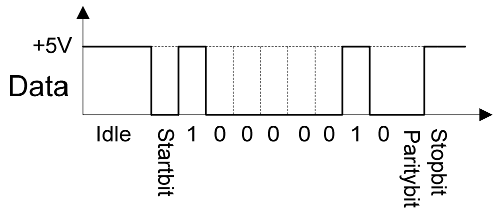

*Figure 17 Coding of the letter ’A’ with even parity*

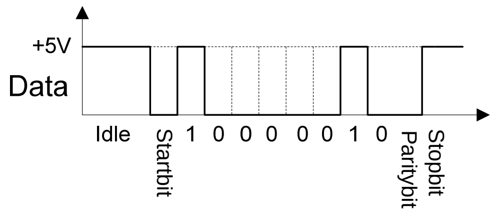

*Figure 18 Coding of the letter ’A’ with space parity*

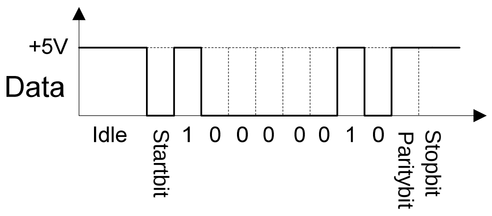

*Figure 19 Coding of the leter ’A’ with mark parity*

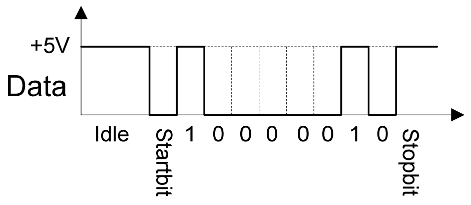

*Figure 20 Coding of the letter ’A’ with no parity*

There are more elaborate ways to detect and even correct errors in data transfer but these are not part of **RS-232-C**. One well known method is called *Hamming coding*, see below.

In the asynchronous transfer we can use the other signals in the protocol, besides the data pins, to control the transfer. A receiving device could for example use **DTR** (Data Terminal Ready) to signal that it is ready to receive data and later to signal that it wants to suspend the transfer. We use what is called *hardware handshaking*.
There is also *software handshaking* using **XON** and **XOFF** signals. The receiver sends the **XOFF** code (decimal 19, hexadecimal 0x13) to tell the transmitter to stop the transfer and then it uses **XON** (decimal 17, hexadecimal 0x11) to tell the transmitter to resume the transfer.

## Baud rate and symbol rate

As stated above the Baud rate is the rate of the transfer over the serial interface, that is the rate of the bits. In modern communication protocols using serial interfaces there are clever methods to transfer more than one symbol within each transferred bit using some kind of modulation. This means that the symbol rate in the transfer can be higher than the bit rate.

## Simplified RS-232

In some simple cases we rely totally on the information buried in the serial bits and do away with all the control and handshaking signals leaving just the receive and transmit lines, **Rx** and **Tx** respectively, and a ground signal of course. The connection could be done in two different ways depending on the devices involved.
If we have one **DTE** and one **DCE** the connections to the **DCE** could be reversed so we can use a straight cable with pin 2 in one end connected to pin 2 in the other end and pin 3 in one end connected to pin 3 in the other end, *Figure 21*.
On the other hand if we are connecting one **DTE** to another **DTE** then **Rx** in one end needs to be connected to **Tx** in the other end and this means that pin 2 in one end needs to be connected to pin three in the other and for this we need a crossover cable, *Figure 22*.
Since both types of cables are quit common we need to make sure that we are using the correct one.

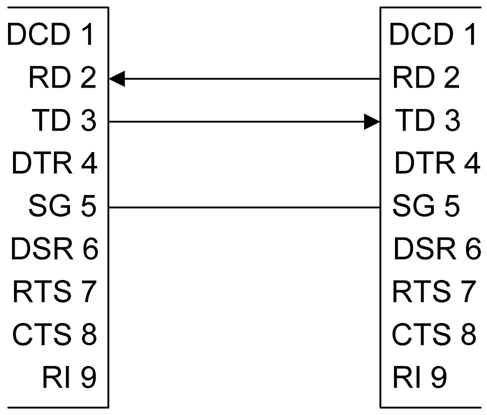

*Figure 21 Straight SCI connection*

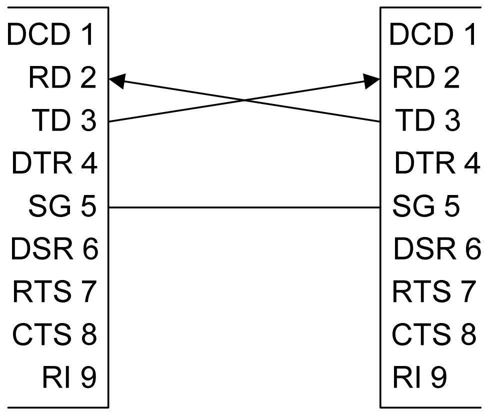

*Figure 22 Crossover SCI connection*

## Typical transmission sequence

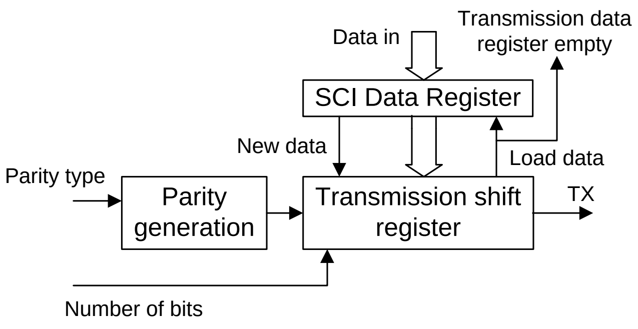

*Figure 23 Asynchronous transmitter*

A typical asynchronous transmitter might look like *Figure 23*. The processor writes data to a data register. When the serial interface is ready to transmit this data it reads the data register and loads the data in parallel into the transmission shift register. At the same time the interface sets the signal **Transmission Data Register Empty** so the processor can write new data to the data register whenever it wants. If enabled a transmission interrupt request will also be triggered at this time. When the data is transferred to the shift register we add start, stop and parity bits to complete the word that is to be transmitted and then the data is shifted out through the **TX** pin bit by bit.

## Typical reception sequence

A typical asynchronous receiver might look like *Figure 24*. The data is shifted in to the shift register through the **RX** pin. When the register is full it will in parallel be loaded into the data register. At the same time the interface will set the signal **Reception Data Register Full** and if enabled a reception interrupt request will be flagged. The interface will also do a check of the received parity bit if it is used and signal if the parity bit is wrong.

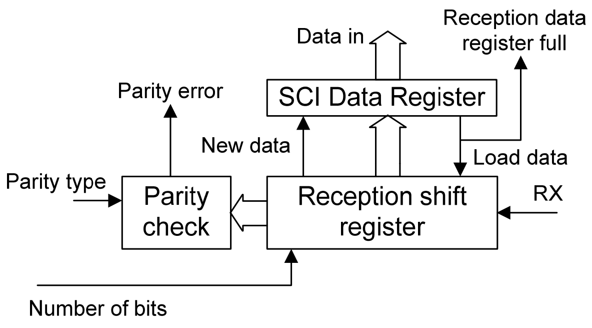

*Figure 24 Asynchronous receiver*

## Hamming Coding

*Hamming Coding* is a set of algorithms that can be used to detect errors in a transfer and even correct some of these errors. The code can have different levels of complexity handling different word lengths and different number of errors. The simplest form is called Hamming (7:4) Code and consists of groups of four data bits ($D_3$, $D_2$, $D_1$ and $D_0$) and three parity bits ($P_2$, $P_1$ and $P_0$). The total word that we send has the structure $D_3D_2D_1P_2D_0P_1P_0$. The parity bits are calculated using the following equations

$$\begin{cases} P_2 = D_3 \oplus D_2 \oplus D_1 \\ P_1 = D_3 \oplus D_2 \oplus D_0 \\ P_0 = D_2 \oplus D_1 \oplus D_0 \end{cases}$$

Where $\oplus$ represents exclusive-OR which is the same as addition modulo-2.
At the receiver we calculate the equations

$$\begin{cases} S_2 = D_3 \oplus D_2 \oplus D_1 \oplus P_2 \\ S_1 = D_3 \oplus D_2 \oplus D_0 \oplus P_1 \\ S_0 = D_2 \oplus D_1 \oplus D_0 \oplus P_0 \end{cases}$$

And the binary 3 bit word $S_2S_1S_0$ tells us in which position in the word, if any, there is an error. `000` indicates no error. `101` indicate error in bit 5 counting from the end of the word, that is error in bit $D_1$. `001` indicates error in bit one, that is the last bit in the word $P_0$. As we can see the coding can detect one error, no matter if it is in a data bit or in a parity bit and we can also indicate in which bit the error is so we can correct it. More than one error in the transmission will give us problem though.
*Example*:
We are to send the four data bits

$$D_3D_2D_1D_0 = 1011$$

This will give the parity bits

$$\begin{cases} P_2 = D_3 \oplus D_2 \oplus D_1 = 1 \oplus 0 \oplus 1 = 0 \\ P_1 = D_3 \oplus D_2 \oplus D_0 = 1 \oplus 0 \oplus 1 = 0 \\ P_0 = D_2 \oplus D_1 \oplus D_0 = 0 \oplus 1 \oplus 1 = 0 \end{cases}$$

and we will send the sequence

$$D_3D_2D_1P_2D_0P_1P_0 = 1010100$$

At a correct reception the receiver will calculate its check bits

$$\begin{cases} S_2 = D_3 \oplus D_2 \oplus D_1 \oplus P_2 = 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\ S_1 = D_3 \oplus D_2 \oplus D_0 \oplus P_1 = 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\ S_0 = D_2 \oplus D_1 \oplus D_0 \oplus P_0 = 0 \oplus 1 \oplus 1 \oplus 0 = 0 \end{cases}$$

The result

$$S_2 S_1 S_0 = 000$$

indicates that the result is correct.
Let´s say that we get a transmission error in bit $D_1$ so that the received sequence is

$$D_3 D_2 D_1 P_2 D_0 P_1 P_0 = 1000100$$

The receiver will calculate its check signals

$$\begin{cases} S_2 = D_3 \oplus D_2 \oplus D_1 \oplus P_2 = 1 \oplus 0 \oplus 0 \oplus 0 = 1 \\ S_1 = D_3 \oplus D_2 \oplus D_0 \oplus P_1 = 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\ S_0 = D_2 \oplus D_1 \oplus D_0 \oplus P_0 = 0 \oplus 0 \oplus 1 \oplus 0 = 1 \end{cases}$$

The result

$$S_2 S_1 S_0 = 101 = 5_{10}$$

indicates an error in bit five if we set the rightmost bit to bit one and we can see that bit five is $D_1$ so we have not only detected an error but also detected in what bit the error occurred.

## RS422, RS423 and RS485

In some more modern equipment **RS-232-C** has been replaced by other standards that could be used at longer distances and at higher speed. We will briefly mention three of these. All three can only use half duplex and can work up to 1200 meter.
**RS-423** is an unbalanced standard that allows one transmitter and ten receivers at a maximal speed of 100 Kbps at a distance of 12 meter while the maximal speed is 1 Kbps at a distance of 1200 meter. The voltage levels are compatible with **RS-232-C**.
**RS-422** is a balanced variation of **RS-423** with the same number of transmitters and receivers. The maximal speed is 10 Mbps at a distance of 12 meter and 100 Kbps at a distance of 1200 meter.
**RS-485** is a balanced bus protocol that allows 32 transmitters and 32 receivers. The maximal speed is 35 Mbps at a distance of 12 meter and 100 Kbps at a distance of 1200 meter.

In most cases we use one unit as master and the other units as slaves but there are also implementations where all units can act as masters and initialize a data session. **RS-485** is used as the electrical layer for a number of well known interface standards, including **DMX**, **Profibus** and **Modbus**.

# Synchronous serial communication

In *synchronous serial communication* the transmitter and the receiver use the same synchronization source, the same clock signal. The clock could be embedded in the data stream or be distributed over a separate line. In the latter case this clock is in most cases generated by a master unit in the system.
We will briefly mention how a protocol with the clock embedded in the data stream might look and then we will move on to synchronous protocols. We will have a look at two common synchronous protocols, **SPI** and **I2C**. These are both intended for short distance communication between a central unit, a processor, and peripheral units like memories and A/D converters.

## Return to zero protocols

In a return to zero (**RTZ**) protocol the transmitted signal will always return to zero (0) level in every bit period. We will give two examples. In *Figure 25* a bit period always starts with the signal going high (1) and ends with the signal going low (0) but depending on if the signal is a zero (0) or a one (1) we will change the duty cycle of the signal. A high duty cycle (long pulse) indicates a one (1) while a low duty cycle (short pulse) indicates a zero (0).
In *Figure 26* we use three levels: positive, negative and zero. A bit period always starts with the signal leaving the zero state and it is going positive for a logic one (1) while it goes negative for a logic zero (0). After half the bit period the signal will in both cases return to zero level.

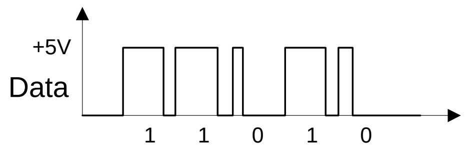

*Figure 25 Return to zero protocol*

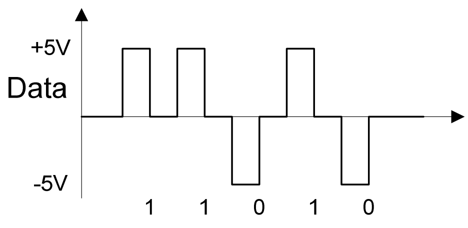

*Figure 26 Return to zero protocol with positive and negative signal level*

In both these examples a bit period starts with a positive or negative flank generated with a fixed frequency and we can use this flank to synchronize the receiver on every transferred bit, that is we have a clock embedded into the data stream.

## Serial peripheral interface, SPI

*The serial peripheral interface*, **SPI**, was developed by Motorola and has received a broad acceptance in the industry and we can find a lot of units using this interface. Examples of units using this interface are A/D and D/A converters, memories (mostly EEPROM and flash memories), real time clocks and sensors. It is an expansion of an older interface from National Semiconductor called **Microwire**. Later on the SPI inter-

face have evolved into new incarnations as **Queued SPI** (**QSPI**) and **MicrowirePLUS**.
Since it is a synchronous protocol for short distances the transfer rate can be high, up to tens of Mbps.
The system consists of one master unit while the other units act as slaves. Although we can connect more than one slave unit to the master only one of these slaves can be active at any one time. As a result the interface gives a fast and reliable communication channel for short distance transfers between two units (peer-to-peer).
The physical connection consists of four wires, *Figure 27* and *Table 9*.

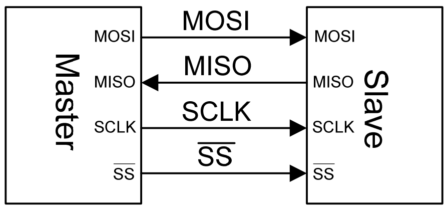

*Figure 27 SPI communication with one slave*

| *Symbol* | *Name* |
|---|---|
| MOSI | Master out, slave in |
| MISO | Master in, slave out |
| SCLK | Serial clock |
| /SS | Slave select |

*Table 9 Signal lines in the SPI protocol*

We can see from *Figure 27* that the master generates the communication clock, **SCLK**, and we have separate lines for communication from the master to the slave, **MOSI** (Master Out, Slave In) and for communication from the slave to the master, **MISO** (Master In, Slave Out). In reality the two registers act as one long shift register and when the master is pushing a bit out on the MOSI line and into the slave it will also push out a bit from the slaves register and on to the MISO line. This means that we have communication in both directions at the same time, we have full duplex. The two lines are actually always sending and it is up to the receiver to decide if it wants to read the data or not. If a unit only transmits data it can just discard the received data. If the unit is only supposed to receive data however it has to produce some dummy data for transfer.
The slave select signal, **SS**, is generated by the master and is used to activate the slave that it wants to speak to and it is also indicating the start and stop of each word in the transmission. Since the signal is active low it will often be named **/SS**. If we have more than one slave in the system the master must be able to generate one **/SS** signal for each slave and these slave signals needs to be mutually excluding each other, *Figure 28*.
The interface in its basic form supports the transfer of 8 bit data but it can be made to work with different word length and the **/SS** signal will then control when the transfer is completed.

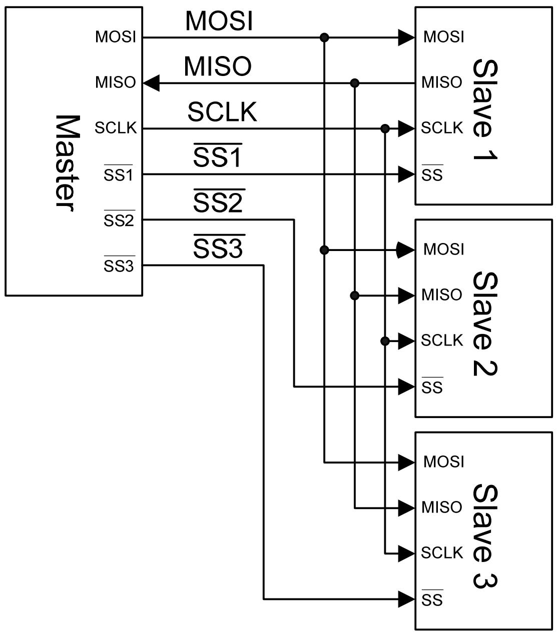

*Figure 28 SPI communication with three slaves*

The SPI interface has no predefined transfer protocol but it can work in four different modes controlling when the data is read relative to the phase of the serial clock, *Table 10*.

| *SPI mode* | *CPOL* | *CPHA* | *Active edge* |
|---|---|---|---|
| *0* | *0* | *0* | *Rising* |
| *1* | *0* | *1* | *Falling* |
| *2* | *1* | *0* | *Falling* |
| *3* | *1* | *1* | *Rising* |

*Table 10 Clocking modes in SPI*

If CPOL=0 then the signal is low when idle. If CPOL=1 then the signal is high in idle mode.

When CPHA=0 data is latched at the rising edge of the serial clock if CPOL=0 and on the falling edge if CPOL=1. If CPHA=1 the polarities are reversed, *Figure 29* and *Figure 30*. Some units can be configured for more than one mode while others only can work in one of the modes. The successor Microwire only supported CPOL=0 and latched incoming data on the rising edge of the serial clock while outgoing data was latched on the falling edge of **SCLK**.

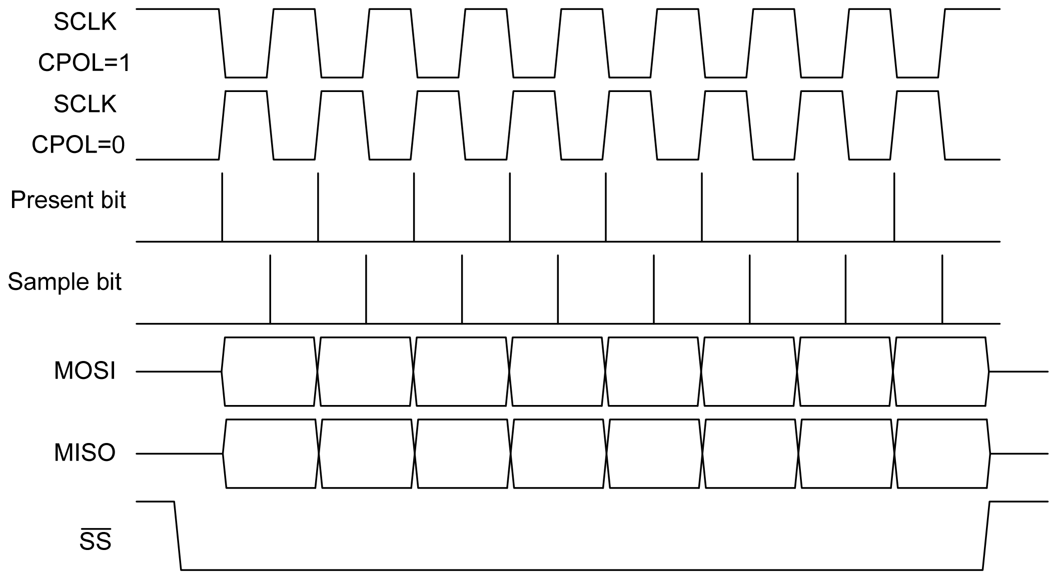

*Figure 29 SPI signals when CPHA=1*

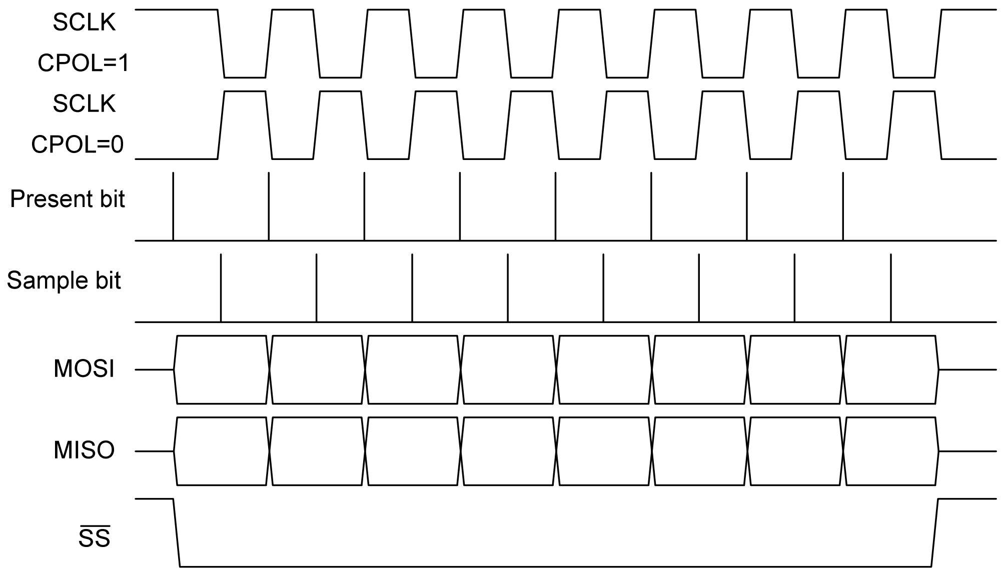

*Figure 30 SPI signals when CPHA=0*

# Inter-integrated circuit, I2C

*The inter-integrated circuit bus*, **I2C**, **I$^2$C** or **I$^2$** was developed by Philips to control the separate units in their stereo and TV equipment but have since moved into the same type of short distance applications as the SPI interface.

The **I2C** has three speed grades, *slow* (under 100 Kbps), *fast* (400 Kbps) and *high-speed* (3.4 Mbps). To fulfill the specifications the distance between the units should be no longer than 3 meters. The short distance makes the bus most suited for use in communication between circuits inside a unit and not between units.

The **I2C** bus is a two wire bus with one line, **SDA**, for the serial data and one line, **SCL**, for the serial clock. The bus can use half duplex and is a multi-master bus. No chip select signals or arbitration logic is required, *Figure 31*.

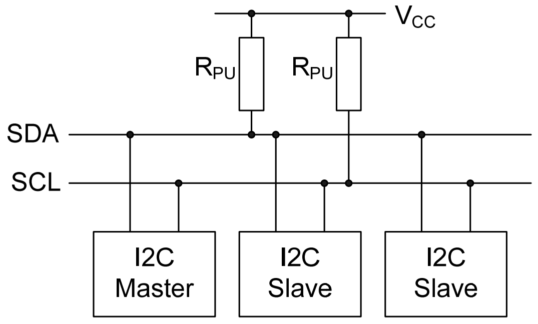

*Figure 31 The I2C bus*

Electrically the bus connection looks like *Figure 32*. We can see that the node connections use open drain devices and that the clock and data busses use pull up resistors to give the bus high level in rest.

In a communication transfer the sequence would be as follows

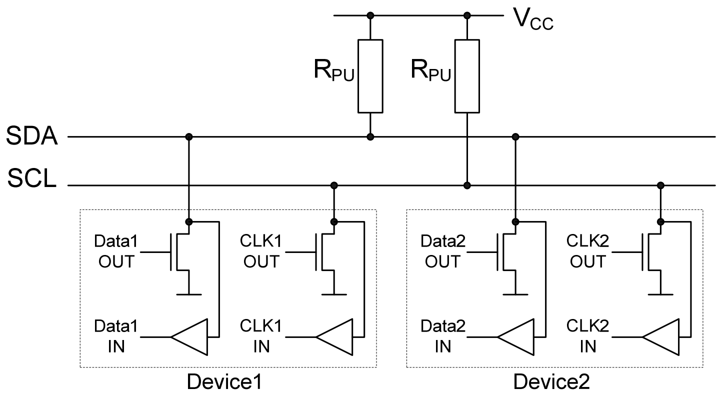

*Figure 32 The I2C bus electrical connection*

1. The master sends a start condition signal (S) and controls the clock signal
2. The master sends a unique 7-bit address addressing the slave that the master wants to talk to
3. The master sends a read/write bit. If the master wants to send (write) data to the slave the bit is ‘0’ and if the master wants to receive (read) data from the slave the bit is set to ‘1’
4. The receiver sends acknowledge bit (**ACK**) confirming that it has received the address and the read/write bit
5. The transmitter (master or slave) transmits one byte of data
6. The receiver sends an ACK bit to acknowledge that it has received the data byte
7. If more data are to be sent phase 5 and 6 are repeated
8. For a write transaction (master transmitting) the master issues a stop condition (P) after the last byte of data
   For a read transaction (master receiving) the master does not acknowledge the final byte but just issues a stop condition (P)

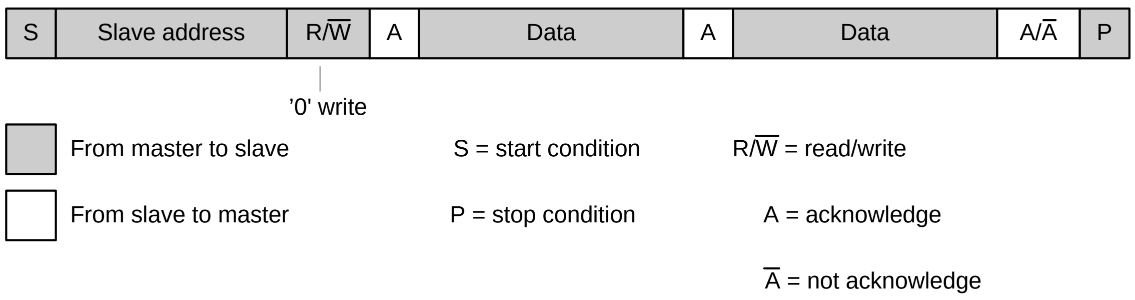

*Figure 33 I2C a master addressing a slave receiver and transfering two bytes of data to the slave*

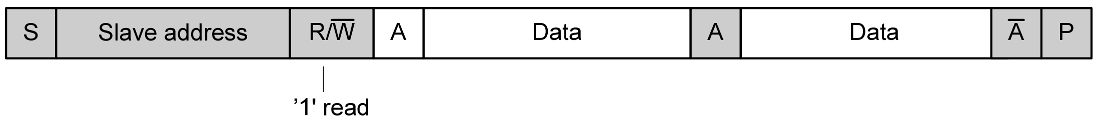

*Figure 34 I2C a master addressing a slave receiver and receiving two bytes of data from the slave*

The start condition (S) is a high-to-low transaction on the SDA line while the SCL line is high, *Figure 35*.
The STOP condition (P) is a low-to-high transaction on the SDA line while the SCL line is high, *Figure 36*.

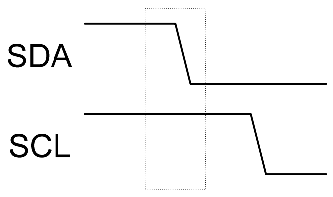

*Figure 35 I2C start condition*

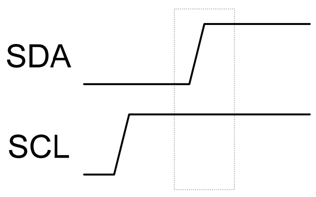

*Figure 36 I2C stop condition*

The ACK signal is generated when the receiver pulls SDA low, *Figure 37*, while the transmitter allows it to float high (**NACK**), *Figure 38*. If the receiver returns a low ACK signal it indicates that it has received the data and is ready for a new transfer. If it returns a high ACK signal it indicates that the unit cannot accept any further data and that the master should terminate the transfer by sending a STOP condition. If the slave has problems keeping up with the speed of the transfer it can slow down the transfer by holding the clock line, **SCL**, low and thereby stopping the clock.
A data bit transaction takes place while SCL is low and the data gets valid when the SCL goes high, *Figure 39*.

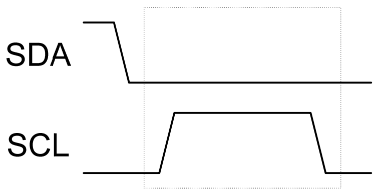

*Figure 37 I2C ACK condition*

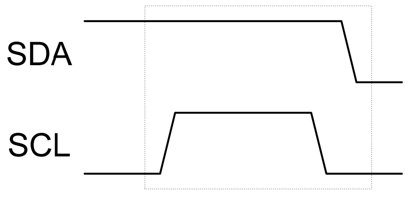

*Figure 38 I2C NACK condition*

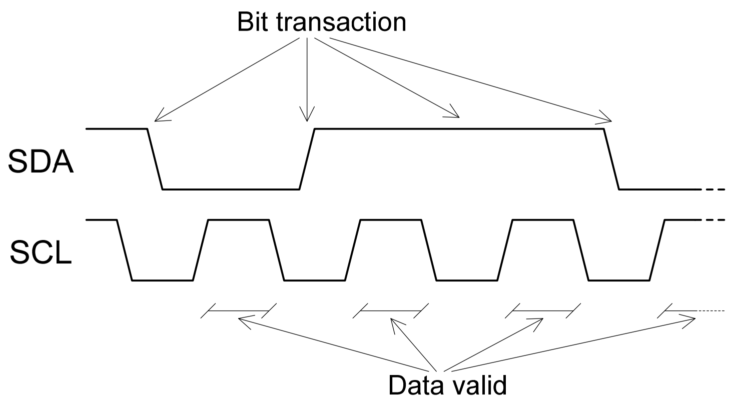

*Figure 39 I2C data transaction*

## 1-wire bus

An example of a very simple bus is the *1-wire bus* from Dallas Semiconductor. It uses a twisted-pair for transfer. In the pair one line is ground while the other carries the data signal and this line can at the same time supply power to the connected devices.

The bus can have one master device and a number of slaves. The connections to the bus are through open drain circuits with a pull up resistor to the power line at the master unit closing the circuit. This means that the bus is at high level when idle. The bus accepts supply voltages in the range 2.8 to 6 Volts and uses standard CMOS/TTL levels for the data transfer *Figure 40*.

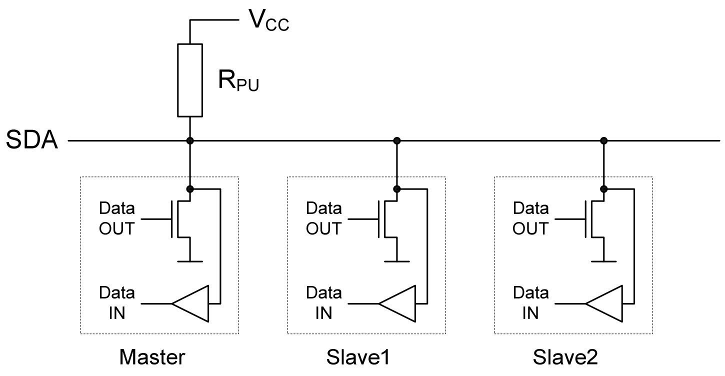

*Figure 40 1-wire bus*

All communication on the bus is started by the master and no slave can talk on the bus if not requested by the master. Communication between slaves can only take place through the master.

The master starts a transfer by resetting the bus by pulling it low for more than 480 µs. Within 60 µs after the end of the reset all slaves that recognize the reset pulse will be pulling the bus low for at least 60 µs. After that each transfer over the bus is initialized by the master pulling the bus low for a short moment. All the units on the bus are synchronized by this falling edge. If the sending unit, be it the master or a slave, wants to send a ‘1’ it continues this initialization pulse by keeping the bus low for at least 60 µs. To send a ‘0’ the unit pulls the bus low for less than 15 µs. The transfer is taking place with LSB first.

All units on a 1-wire bus have a unique 64 bit serial number. Starting from LSB the number begins with 8 bits giving a family code to identify the device type. This is followed by 48 bits giving a unique individual address for each device. The last 8 bits are a **CRC** (Cyclic Redundancy Check) checksum.

The slaves on the bus can be powered over the bus, *Figure 41*. To do this each unit has a rectifying diode and a capacitor built in. When the bus is at high level the bus charges the capacitor through the diode. When the bus is at low level the diode is reverse biased and isolates the charged capacitor. The slave unit then takes its power from the charge in the capacitor.

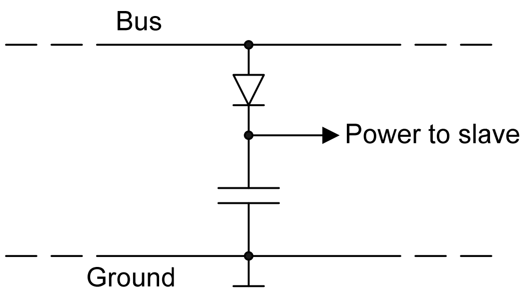

*Figure 41 Power over 1-wire bus*

The bus has developed into a more modern version, **1-wire Extended**, that increases the noise immunity on the bus.

Typical units developed for this bus include memories, A/D-converters, clocks and temperature sensors. A special type of units is the so called **iButtons**. These small buttons communicate wirelessly through induction with a unit on the bus and can be used for security identification, for example to gain access to garages end entrances.