CHALMERS
Department of Computer Science and Engineering

2006-12-11

Embedded systems Communication interfaces with the empasis on serial protocols

Introduction

With communication we mean the transport of information from a transmitter to at least one receiver.
The transport might be in only one direction or in both, it might be *uni- or bidirectional*. If we have communication in only one direction we talk about a *simplex* channel. If the communication is 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 we have (*full*) *duplex*.
The communication might be between only two units, one transmitter and one receiver, or if the communication is bidirectional two tranceivers (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.
Another situation arises if we have a number of units connected together trough a common network, we talk about a *bus topology*. In this case in its basic form we have one unit transmitting a message containing some kind of address and the other units are listening and if the listening unit is having the same address as the one transmitted it will receive the message while the other units 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 telling what kind of data that will be transmitted and then all units that have interest in this kind of data will listen. Obviously all units can not talk at the same time on a bus network so the access to the bus neads 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 that might be a command or a request for an answer from some unit. In this case the other units are called *slaves*. In other protocols more than one unit can initialize a conversation and in that case we talk about a *multi master* protocol.

In all types of bus communication we need a way to deside which 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 quit simple but it is inefficient if the needed transmission rate differs for the different units on the bus. In this case we would vaste a lot of time slots passing the token 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 deside 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 (Controller area network) 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 byte length 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 might seem like the parallel choice would always be better since we have all bits at the same time and it would be faster than serial communication. This might be true but we have some problems with parallel protocols.

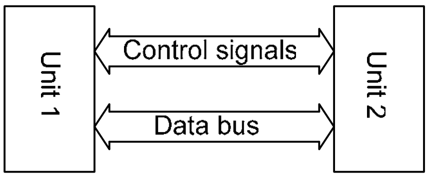

Figure 1 parallel communication

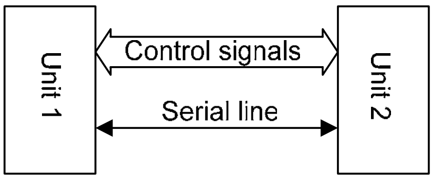

Figure 2 Serial communication

Modern equipment become more and more complicated and it gets harder and harder to find room in the integrated circuits and on the printed circuit boards for the parallel lines. Things are being even more complicated by our constant efford 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 byte will reach the receiver at different times and we have a great risk of reading false data, *Figure 3*.

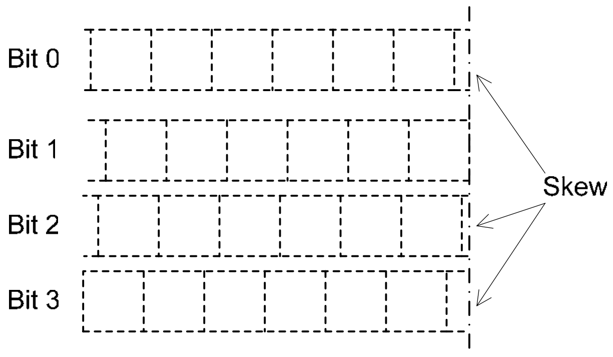

Figure 3 Missreading caused by skew

If we use serial communication the routing of wires will be simpler and we don´t have any skew between bits. For this reason almost all modern fast communicaton protocols are serial.

In the serial case we still have a risk of missreading 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, on a separate wire, but then we need this extra wire. An other 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. This would be a somewhat more complicated protocol. In both cases we talk about *syncronious communication* where both transmitter and receiver use the same synchronization signal. Later we will have a look at the SPI protocol (Serial periph-

eral interface) and the Inter-integrated circuit protocol I2C. Both protocols use a separate clock line.
An other way is to let transmitter and receiver use their own internal clocks to decide the transmission speed and there by decide when to transmit a bit and when to read the received bit respectively. We have an *asyncronious protocol*. Since we can not be sure that the two clocks run at exactly the same speed we have to keep the transmission rate lower than in the syncronious case so the two clocks don´t drift that far apart and we have to have some syncronisation between the transmitter and the receiver to set the timing of the transfer. One common way to do this is to add extra bits to the transmission for syncronisation. 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.
When we talk about serial communication we need to seperate unbalanced and balanced wire links. In the *unbalanced* case we use only one wire and the transmitted bit, '1' or '0' is a voltage referenced to ground, *Figure 4*.
In the *balanced* case we use two wires and the bits 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 a unbalanced signal is disturbed by interference, noise, this will be an extra voltage that will be added to the transmitted voltage that represents the bit and we have the risk of missreading the received bit, *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 are only slightly affected by the disturbance and the received bit will still be correct, *Figure 7*.
One thing that is used here and there in communication are *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 simpliest 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* shows the frequencies used in the modem standard V.21.

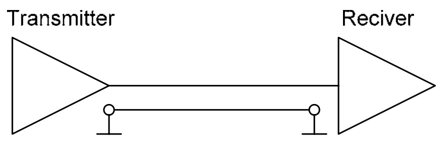

Figure 4 Unbalanced link

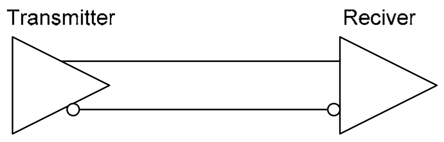

Figure 5 Balanced link

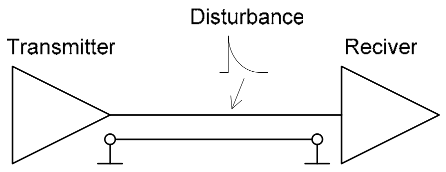

Figure 6 Unbalansed link with disturbance

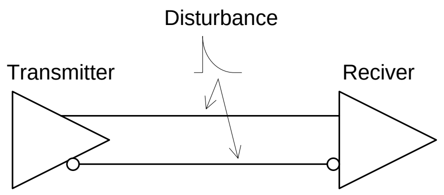

Figure 7 Balanced link with disturbance

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 fibre) or radio waves. In these medias we normally modulate our information on a carrier wave and this will give a serial channel even if we 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 there by 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 no move in to some examples of communication protocols starting with a parallel protocol and then moving on to serial solutions.

Example 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 synchronious serial interface (**SPI**) or a **I2C** interface. We will get back to these interfaces later on.
For the moment we will have a look at the parallel case. As the description implies we use a parallel approach, that is the data is presented at the same time, at separate wires on 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 finaly we will need some control wires.
Let´s use the way **HC12** addresses external memory as an our 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 is 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 he 16 address lines come out of **PORTB** (**A0**-**A7**) and **PORTA** (**A8**-**A15**) in the processor while the data bus comes out of the same two ports in wide mode and out of **PORTA** in narrow mode. As we can see the

data and address bus shares the same port(s) and the address respectively 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 but when we get to the read/write phase this address still needs to be active so we need to use some external logic to remember the address during this second phase.
Let us as an example see how we can use the emulation expanded narrow mode to address a 8K big static RAM memory called **6264**. The memory has a 13 bit wide address bus a 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 from the memory
- One active low write enable sinal **/WE** that we use when we want to write 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**. High level indicates read phase
- The E clock **ECLK** which is low during the address phase and high during the data phase

From these two signals we will use some logic 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 internal). Since our 8K memory will only fill half of one of these external slots we will use **A15**-**A13** and a 3/8 decoder to place the memory in the address space. Let´s place it at the start address `0xC000`. If we like we can use the decoder to place other memories in other slots. Now let us create the control signals. Let us thart with the **/WE** signal. We can realize that this signal should be active (low) under the data phase (**ECLK** high) if the **R/W** signal is low. We will get the truth table in *Table 3* and the logical expression

$$/WE = NOT\left[ECLK\ \ AND\ \ NOT\left(R/W\right)\right]$$

The output enable signal **/OE** should be active (low) under the data phase and when the **R/W** signal is high. We will get the truth table in *Table 2*.
We are now ready to draw a full schematic of the memory interface, *Figure 8*.

$$/OE = NOT\left[ECLK\ \ AND\ \ R/W\right]$$

ECLK	R/W	/WE
0	0	1
0	1	1
1	0	0
1	1	1

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

ECLK	R/W	/OE
0	0	1
0	1	1
1	0	1
1	1	0

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

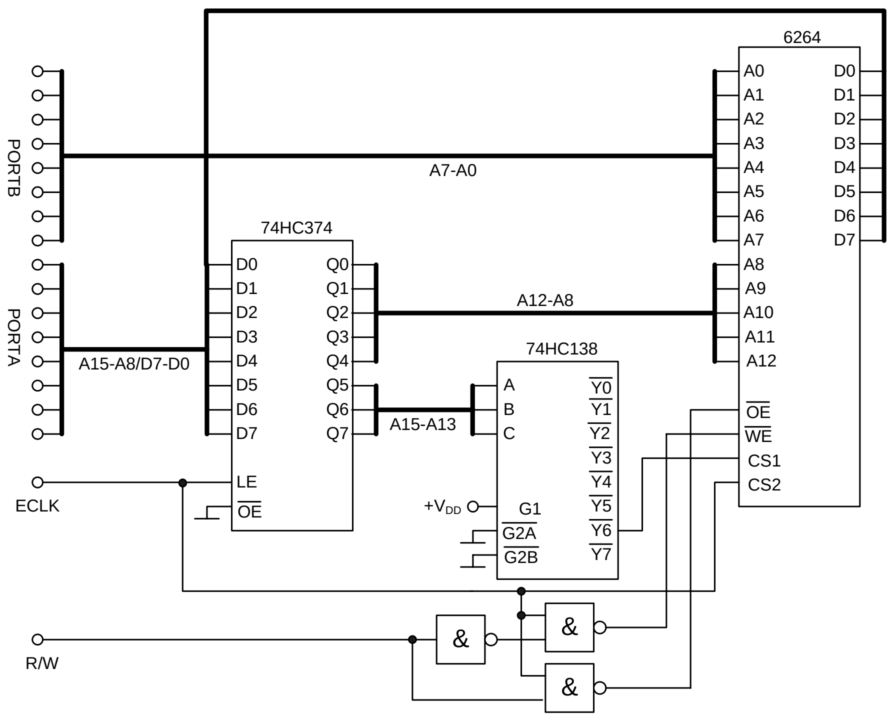

Figure 8 Parallel external memory interface

GPIB or IEEE-488

GPIB (General Purpose Instrumentation Bus) is a 8 bit parallel communication bus developed for the connection of programmable measurement instruments. It was developed by Hewlett-Packard who named it **HP-IB** (Hewlett-Packard Instrument bus) but when it got standarilized it got it´s present name. The bus has been standardized by the US organisation **IEEE** (Institute of Electrical and Electronics Engineers) as standard **IEEE-488**. The standard has later evolved to standard **ANSI/IEEE488.1.** In Europe it has been standardized by **IEC** (International Electrotechnical Commission) as standard **IEC-625**. Later on 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 programming command set that is used by any **SCPI** instrument.

Three types of devices can be connected to the bus: controllers, talkers and listners. 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 it´s 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 have the head role of system, controller. A *listener* is a device that receives data from the bus when instructed by the controller. A *talker* transmits data one bus when so instructed by the controller.

The data transfer rate in standard **GPIB** can be up to 1.8 MbByte/second. There is a new high speed standard, **HS488**, that can use data transfers up to 8 MByte/second.

The physical interface consists of 16 signal lines and 8 ground lines, *Table 4*. The signal lines are divided into three groups: 8 data lines (each of these 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 indicat 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 contol byte has been placed on the data lines and can now safely be accepted by other devices

Pin	*Abbriviation*	*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 five interface management lines manage the flow of data and control bytes across the interface.

- The **ATN** (Attention) sinal is asserted by the controller to indicate that it is placing a address or control byte on the data bus
- The **EOI** (End or Idertify) signal has two uses. 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

The devices are connected in either a linear or a star configuration or a combination of the two using a shielded 24 conductor cable, *Figure 9*, *Figure 10* and *Figure 11* respectively. The maximal separation between two devices is 4 meters while the maximal total cable length is 20 meters.

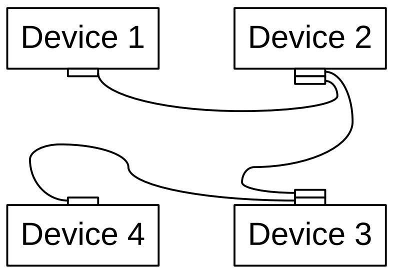

Figure 9 Linear GPIB configuration

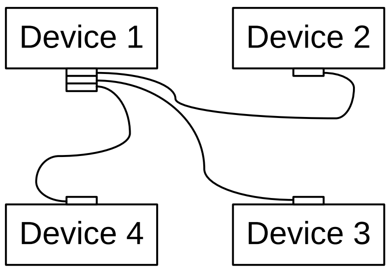

Figure 10 Star GPIB configuration

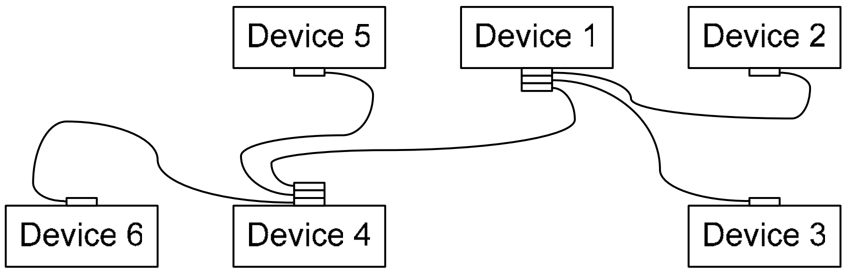

Figure 11 Combination of linear and star GPIB configuration

The interconnection use a chuncy connector with both a male and a female side which means that connectors can be stacked on top of each other, *Figure 12*.
The bus uses standard TTL logic levels with a negative logic meaning that a ‘1’ is a low TTL level while a ‘0’ is a high TTL level.

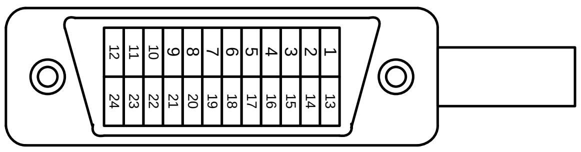

Figure 12 GPIB connector

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 a abbreviation of Electric Industries Association representing many manufacturers in the U.S electronics industry. **RS-232-C** is a standard that defines how ‘1’:s and ‘0’:s should be electrically transmitted, including the voltage levels needed as well as the other signals necessary for computer communication.

Since it only gives the electrical characteristics it can be used for different protocols, both synchronious and asynchronius, even if we mostly associate it with asynchronious communication. **RS-232-C** is an unbalanced protocol where ones and zeros are transmitted using negative and positive voltages. A one, a mark, is represented by an electrical signal between -3 and -15 Volts (often -12 Volts). A zero, 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 levels seen inside computers the com-munication in-terface must contain means to convert from 0 and +5 Volts to the **RS-232-C** levels and back again. The maximal distance between transmitter and receiver is 15 meter. The protocol could 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 transmission pin (**TD**) as output and the data communication equipment (**DCE**) which uses this pin for input. The **DTE** should have male connectors while the **TCE** should have female connecors. 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*).
We are not going to go through all of the signals but we can see from *Table 1* that the full implementation includes pins for clock transfer (pin 15, 17 and 24) which means that it can be used for synchronious communication. These signals are missing in *Table 2* which means that this implementation can only be used for asynchronious transfer.
If we are to connect two interfaces of the same kind (two **DTE** units) some of the signals has to be twisted. The most important of these are the **TD** and **RD** signals.
Let us look at the most common use of **RS-232-C**, asynchronious communication, often called **SCI** communication.

Asynchronious serial communication, SCI

The transfer will have to have a clock. 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 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 asynchronious communication no common clock is transfered between the two interfaces which means 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 clocks. We do this by starting each transmitted word with a *start bit* that will retrigger the receivers 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 bits may be from five to eigth bits.
After the data bits 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 simpliest 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 we set the parity bit to one (1). In *Figure 13* 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*.

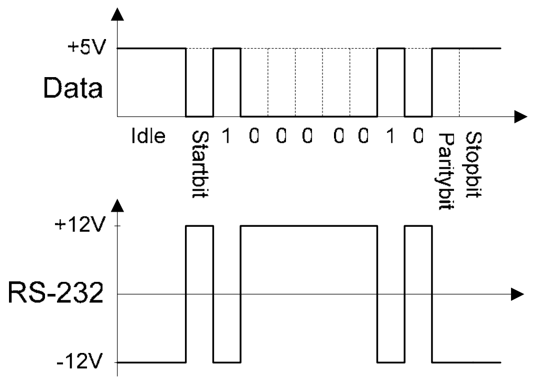

Figure 13 Coding of the letter 'A with odd parity' in computer and on RS-232 link

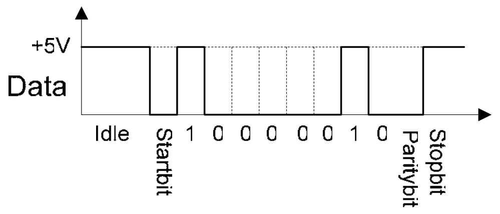

Figure 14 Coding of the letter 'A' with even parity

Even parity works 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 14* we see the same example as in *Figure 13* 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. We call this *space parity, Figure 15*. 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. We call this *mark parity, Figure 16*.

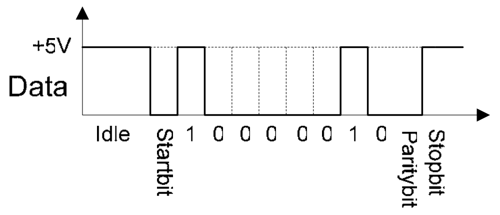

Figure 15 Coding of the letter 'A' with space parity

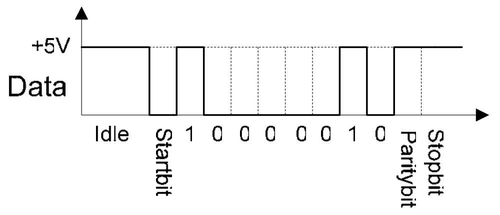

Figure 16 Coding of the leter 'A' with mark parity

If we are not using any parity bit and this bit is omitted from the transfer we say that we have *no parity, Figure 17*.
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 is called *Hamming coding,* see below.
In the asynchronious 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

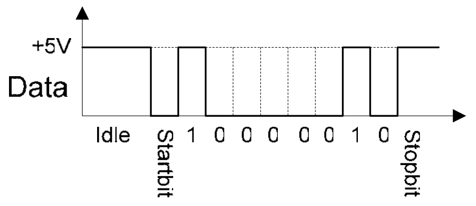

Figure 17 Coding of the letter 'A' with no parity

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.

Typical transmission sequence

A typical asynchronous transmitter might look like *Figure 18*. 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 into the transmission shift register. At the same time it sets the signal **Transmission Data Register Empty** so the processor can write new data to the data register whenever it wants. When the data is transferred to the shift register it is completed with start, stop and parity bits so we generate the complete word that shall be transmitted and then the data is shifted out through the **TX** pin.

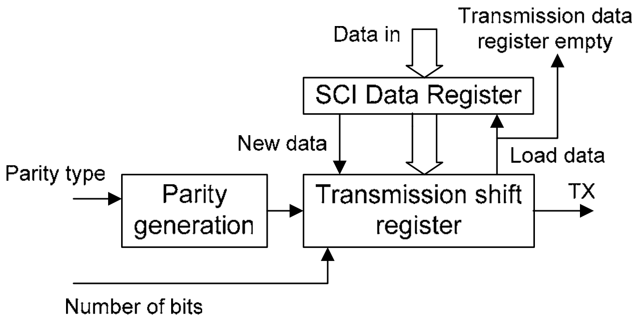

Figure 18 Asynchronous transmitter

Typical reception sequence

A typical asynchronous receiver might look like *Figure 19*. The data is shifted in to the shift register through the **RX** pin. When the register is full it will be loaded into the data register. At the same time the interface will set the signal **Reception Data Register Full**. The interface will also do a check of the received parity bit if it is used and signal if the parity bit is wrong.

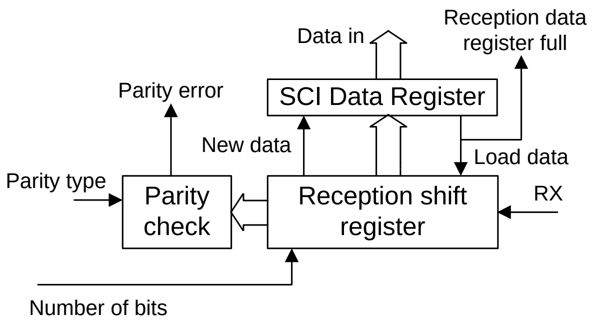

Figure 19 Asynchronous receiver

Hamming Coding

Hamming Coding is a set of algorithms that can be used to detect errors in a transfer and even correct these errors. The code can have different levels of complexity handeling different word lengths and different number of errors. The simpliest form is called Hamming (7:4) Code and consists 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 mololo-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, the error is. 000 indicates no error. 101 indicates 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, that is in 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.

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 compatable 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 others 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 os well known interface standards, including **DMX**, **Profibus** and **Modbus**.

Synchronious serial communication

In synchronious serial communication the transmitter and the receiver use the same synchronization source, the same clock. The clock could be embedded in the data stream or be

distributed over a separate line. In the latter case the 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 synchronious protocols. We will have a look at two common synchronious 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 protocol the transmitted signal will always return to zero (0) level in every bit period. We will give two examples. In *Figure 20* 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.
In *Figure 21* we use three levels positive, negative and zero. A bit period always starts with the signal leaving the zero state and 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.
In both these examples a bit period start with a positive or negative flank and we can use this flank to synchronize the receiver on every transfered bit, that is we have a clock embedded into the data stream.

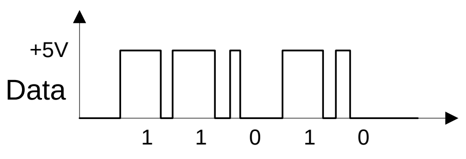

Figure 20 Return to zero protocol

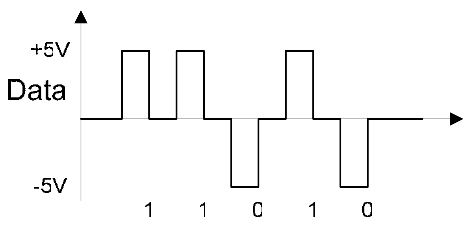

Figure 21 Return to zero protocol with positive and negative signal level

Serial peripheral interface, SPI

The serial peripheral interface, **SPI**, was developed by Motorola and have received a broad acceptance in the industri 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 flach memories), real time clocks and sensors.
Since it is a synchronious protocol for short distances the transfer rate can be high, up to tens on Mbps.
The system consists of one master unit while the other units are slaves. Although we can connect more than one slave unit only one of these can be active at any one time.
The physical connection consists of four wires, *Table 7* and *Figure 22*.
We can see from *Figure 22* that the master generates the communication clock 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**

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

Table 7 Signal lines in the SPI protocol

(Master In, Slave Out). 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 deside if it wants to read the data or not. The slave select signal, **SS**, is generated by the master and is used to activate the slave that it wants to speak to. 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, *Figure 23*.
A SPI interface could be working in four different modes controlling when the data is read relative to the phase of the serial clock, *Table 8*.
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. Some units can be configured for different modes while others only can work in one mode.

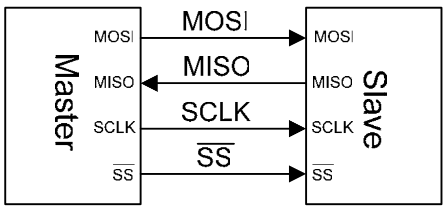

Figure 22 SPI communication with one slave

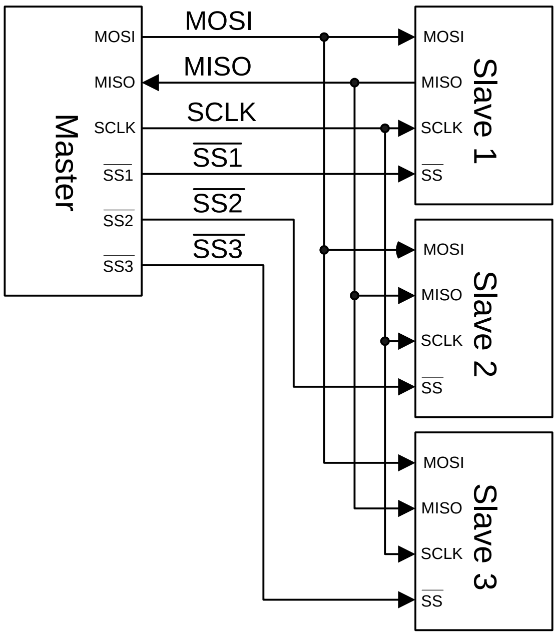

Figure 23 SPI communication with three slaves

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

Table 8 Clocking modes in SPI

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 furfil the specifications the distance between the units should be no more than 3 meters.

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 are a multi-master bus. No chip select signals or arbitration logic is required, *Figure 24*. Electrically the bus connection looks like *Figure 25*.
In a communication the sequence would be as follows

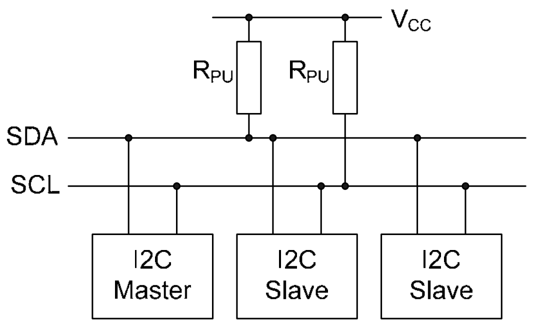

Figure 24 The I2C bus

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 ro 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)

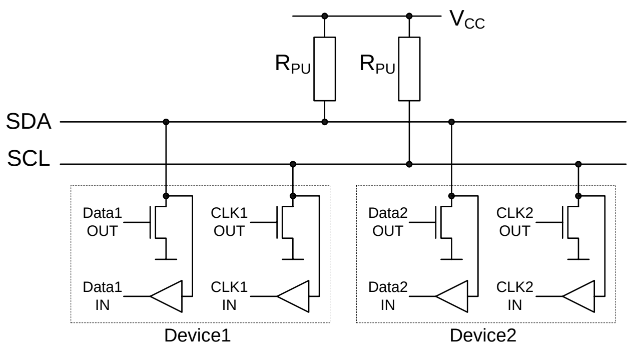

Figure 25 The I2C bus electrical connection

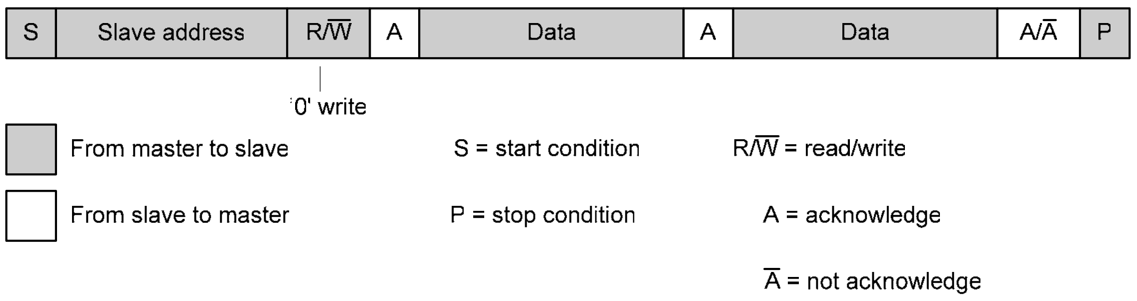

Figure 26 I2C a master addressing a slave receiver and transfering two bytes of data to the slave

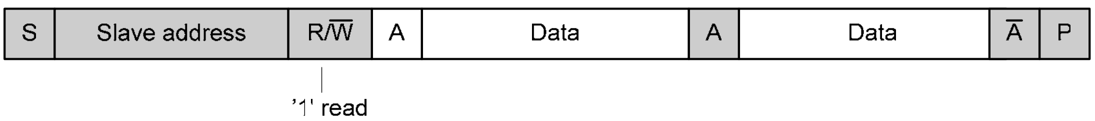

Figure 27 I2C a master addressing a slave receiver and receiving two bytes of data from the slave

The start condition (S) is a high-to-low transaction on the SDA line while the SCL line is high, *Figure 28*.
The stop condition (P) is a low-to-high transaction on the SDA line while the SCL line is high, *Figure 29*.
The ACK signal is generated when the receiver pulls SDA low, *Figure 30* while the transmitter allows it to float high (NACK), *Figure 31*.
A data bit transaction takes place while SCL is low and the data gets valid when the SCL goes high, *Figure 30*.

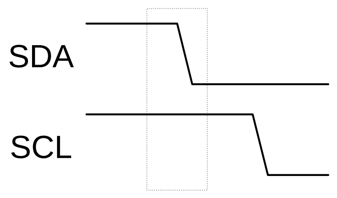

Figure 28 I2C start condition

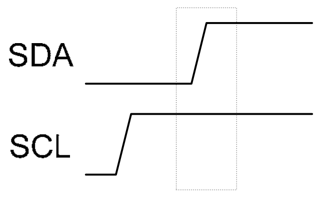

Figure 29 I2C stop condition

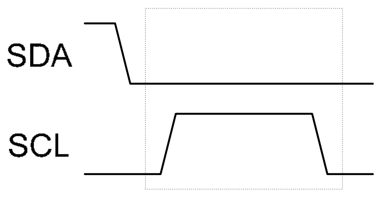

Figure 30 I2C ACK condition

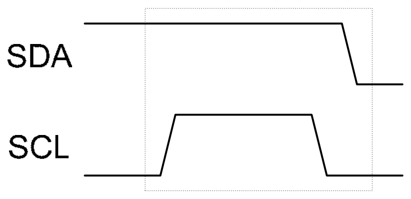

Figure 31 I2C NACK condition

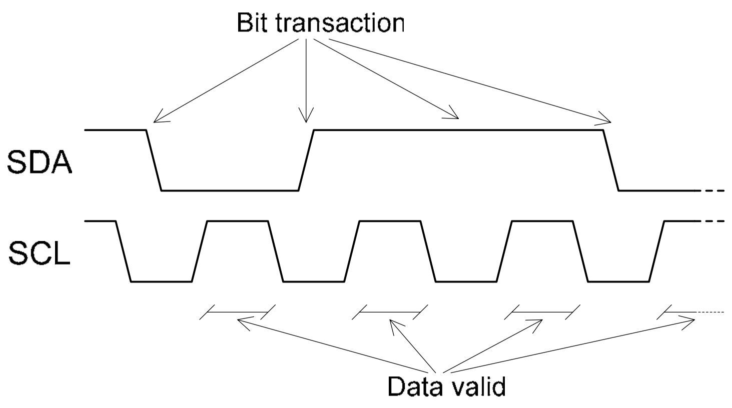

Figure 32 I2C data transaction