MCPros
MicroController Pros Corporation

Microcontroller Interfaces – Part 1

Synchronous Microcontroller Communication Interfaces: SPI and Microwire versus I^2C

By Volker Soffel
MicroController Pros Corporation
May 1, 2003

This paper is the first in a series about various microcontroller communication interfaces, their application, features, benefits and disadvantages. You can find about a dozen different communication interfaces integrated in microcontrollers on the market today to tackle almost any conceivable communication task. With this increasing interface variety it becomes more complicated to pick the "right" interface for the task at hand. Each of the available interfaces has its advantages and disadvantages and is better suited at some tasks than others.

In this article series, I will make the attempt to describe in the allotted space the various interface types, highlight their typical uses, summarize the key electrical and physical parameters, and point out the features and benefits of each. My goal is to provide you with some key information in a concise format that hopefully will make your task of selecting interfaces for your next application easier.

Synchronous Interfaces

The types of interfaces available to you can be classified into two basic categories: Synchronous interfaces and asynchronous interfaces. In this article we'll focus on synchronous interfaces.

Synchronous interfaces are characterized by the presence of a dedicated receive/transmit clock signal. A "Master" device usually outputs a clock signal that is received by all "Slave" devices to receive and transmit data in synch. The advantage: Each device works with the transmit/receive clock of the master independent of any oscillator variations of each individual device; so these

interfaces are very suitable for use with cheap oscillators that have large frequency variations

Examples of synchronous interfaces are: SPI (Serial Peripheral Interface), developed by Motorola, MICROWIRE developed by National Semiconductor, I^2C (Inter Integrated Circuit) developed by Philips/Signetics, and USART (Universal Synchronous & Asynchronous Receiver Transmitter) - as the name suggests a USART can either be used in a synchronous or asynchronous mode, so it falls into both categories.

Synchronous interfaces were designed mainly to connect peripheral devices on the same circuit board, like external EEPROMS, A/D converters, display drivers and sensors to microcontrollers. They are only suitable to bridge relatively short distances (< 1 meter).

Microwire and SPI versus I^2C

SPI is a close cousin of the older Microwire. Both interfaces are very simple and basically consist only of an 8-bit serial shift register and (for master devices) a programmable shift clock. There is no means of addressing devices. Typical applications consist of one master device (usually a microcontroller) and one or multiple slave devices (usually peripheral functions, like A/D, EEPROM, display drivers, etc.).

I^2C is quite a bit more complex than SPI and Microwire, which results in a larger silicon area and therefore slightly more expensive devices. In addition Philips is collecting licensing fees for I^2C implementations from competitors, adding to the cost of I^2C devices.

Connecting External Peripherals

There is a minimum of 3 connections for SPI and Microwire: serial clock, serial data out and serial data in. Therefore you'll see those interfaces sometimes referred to as 3-wire interfaces. The interconnected devices need to also share the same Vcc and GND of course and in the case of multiple connected devices you need one chip select for each connected slave device (for just one slave, the slave's chip select can be enabled all the time – not recommended, but possible).

If you want to connect N devices to your microcontroller with Microwire or SPI you need to sacrifice 3+N pins to do the job. This is an area where I^2C has an advantage. I^2C features a 7-bit address as part of the protocol. As such I^2C can address up to 128 devices on the bus without the need for dedicated chip select signals.

The Need For Speed

Microwire and SPI shine when it comes to speed. I^2C was initially specified at a maximum speed of 100kbits/sec. This was later increased to 400kbits/sec and lately some devices started to show up that boast 1Mbits/sec. This still pales in comparison to Microwire and SPI speeds. SPI has the edge over Microwire, due to the availability of higher speed peripheral devices. Today's serial EEPROM for example support up to 3MBits/s for Microwire and up to 10Mbits/sec for SPI. But even the slowest Microwire and SPI peripherals still beat the typical 100 or 400kbit/s I^2C speeds.

Increasing the speed gap is the fact that SPI and Microwire have full–duplex capability (can receive and send data at the same time), while I^2C, due to its two-wire nature (one clock, one data) can only communicate half-duplex.

Why can speed be important? Current consumption for one - many microcontroller applications spend most of their time in power save modes and only short periods of time in "normal" operating mode. The faster a read or write operation to a peripheral can be completed, the shorter the time the controller needs to be active. This is especially true for access to large external EEPROM memories.

Multi-Master Systems

I^2C offers better support for multi-master systems. The interface has built in arbitration to detect multiple devices sending on the bus at the same time and to give priority to the one that first sends a "0". Microwire would require some software implemented handshaking via a standard I/O pin to allow for multiple master devices on the bus. SPI has a crude way to support multi-master systems via its built in "fault logic". It can detect requests of devices to become the master via the dedicated SS (slave select) pin.

Noise Immunity

One possible disadvantage of I^2C should not go unmentioned: Higher noise sensitivity and along with it lower data integrity. I^2C uses a read/write bit which follows the initial 7 address bits to tell a peripheral whether data should be read or written. In addition I^2C is level sensitive - in contrast to Microwire and SPI, which are edge sensitive. This means that I^2C samples data during the high or low phase of a bit and you can easily envision that noise could flip the read/write bit. So if you wanted to read data from your external EEPROM, but noise turned your read bit into a write bit, your memory might get corrupted. Microwire and SPI peripherals on the other hand implement read and write operations via explicit commands send over the bus, making selecting the "wrong" operation less likely.

So which synchronous interface should you give the preference?

If you have many devices to connect and in addition have multiple microcontrollers in your system that can act as masters, then I^2C is the interface of choice. The same holds true if you need to keep the number of interconnects, board routing and pins required for the interface to an absolute minimum. The I^2C interface is very popular in video and audio applications, due to Philips' (microcontroller & application specific peripheral) dominance in those applications. If you develop such applications you might not find your desired peripheral function with any other interface.

If your main concerns are low cost, high speed or noise immunity, either Microwire or SPI are preferable. An added advantage is that MICROWIRE/PLUS microcontrollers can talk to SPI peripherals and SPI microcontrollers can talk to Microwire peripherals with minimum additional software overhead, which gives you a large selection of available peripherals to choose from for most applications.

There's still more to cover on Microwire, SPI and I^{2}C, so in next month's article, I'll cover SPI's, Microwire's and I^{2}C's protocol format and electrical specs in some more detail before moving on to the asynchronous interfaces.

Microcontroller Interfaces – Part 2

Synchronous Microcontroller Communication Interfaces: SPI, Microwire and I^{2}C Protocol Formats

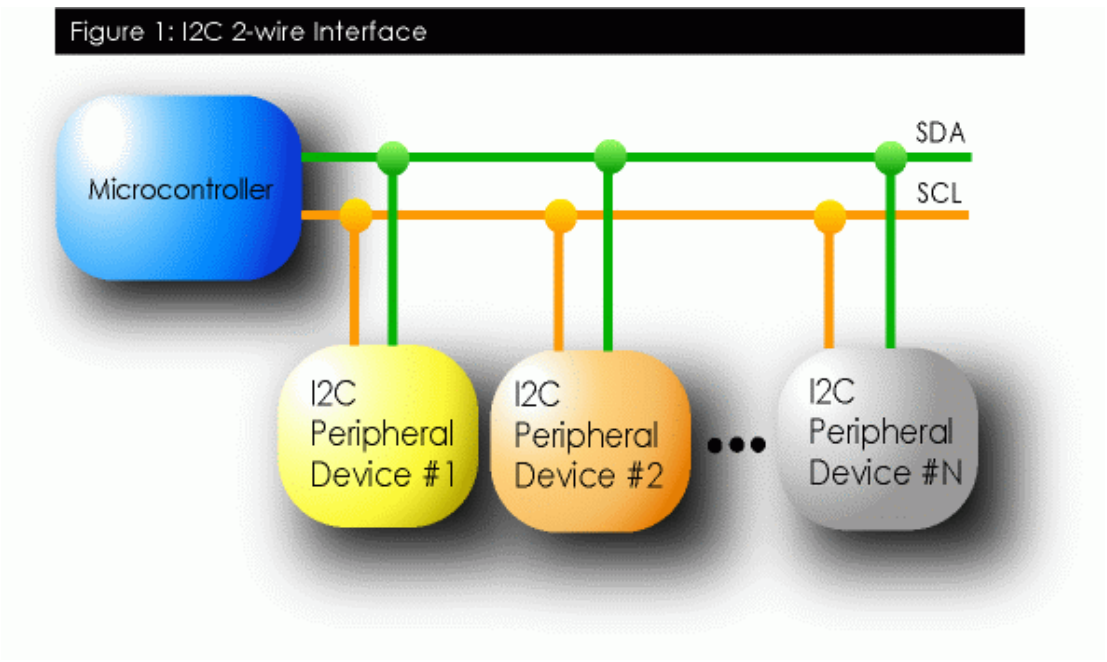
By Volker Soffel
MicroController Pros Corporation

This article is part 2 in a series about various microcontroller communication interfaces. In part 1 of the series, I discussed some typical applications and implementations of the SPI, Microwire and I2C interfaces. I looked at their general differences, basic features, advantages and disadvantages. In this part, I'll cover the specifics of their protocol formats. In Part 3 we will start to take a closer look at asynchronous interfaces (UART based RS232C/RS485/LIN, CAN, USB and Ethernet).

I^{2}C Bus

I^{2}C is a 2-wire, half-duplex, serial bus, as shown in Figure 1. The two I^{2}C signals are serial clock (SCL) and serial data (SDA). Both lines are bidirectional and must be connected to Vcc via pull-up resistors.

Figure 1: I2C 2-wire Interface



The SCL and SDA pins need to be implemented as an open drain or open collector type to allow for a wired AND function on the bus ("0" wins over "1") .

The device initiating data transfers and providing the clock signal on the bus is called a "master". A device being addressed by the master is called a "slave".

The addressed I²C slave can slow down or stop the master by keeping the SCL line pulled low (clock stretching) until it is ready to continue. This way a slow slave device can still keep up with a fast master.

I²C supports multiple masters on the bus through its built in bus arbitration. In case of multiple masters trying to send data at the same time, priority is given to the master that first sends a "0" bit.

I²C Communication Protocol

Data can be transferred at rates up to 100kbps in Standard mode, up to 400kbps in Fast mode and up to 3.4Mbps in High-Speed mode (finding peripherals that are that fast though is a challenge). In Standard mode 7-bit addressing is used. In the other modes, slaves can either have 7- or 10-bit addresses. The total capacitive bus load of all devices connected to the bus can not exceed 400pF.

Figure 2: I2C Protocol

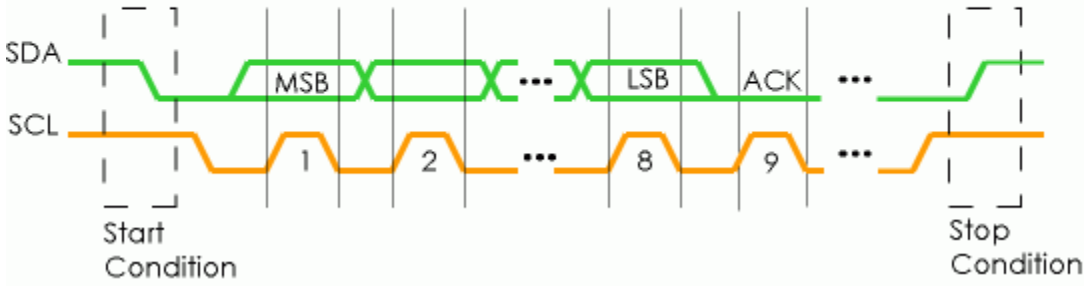


Figure 2 shows the I²C Communication Protocol. The data transmission format is most significant bit (MSB) first. I²C is level sensitive; therefore SDA must be stable while SCL is high. The SDA line can only change when SCL is low, with two exceptions:

- **Start Condition**: The master signals the beginning of a transfer by sending a 1-to-0 transition while SCL is high.
- **Stop Condition**: A 0-to-1 transition issued by the master while SCL is high, signals the end of a transfer.

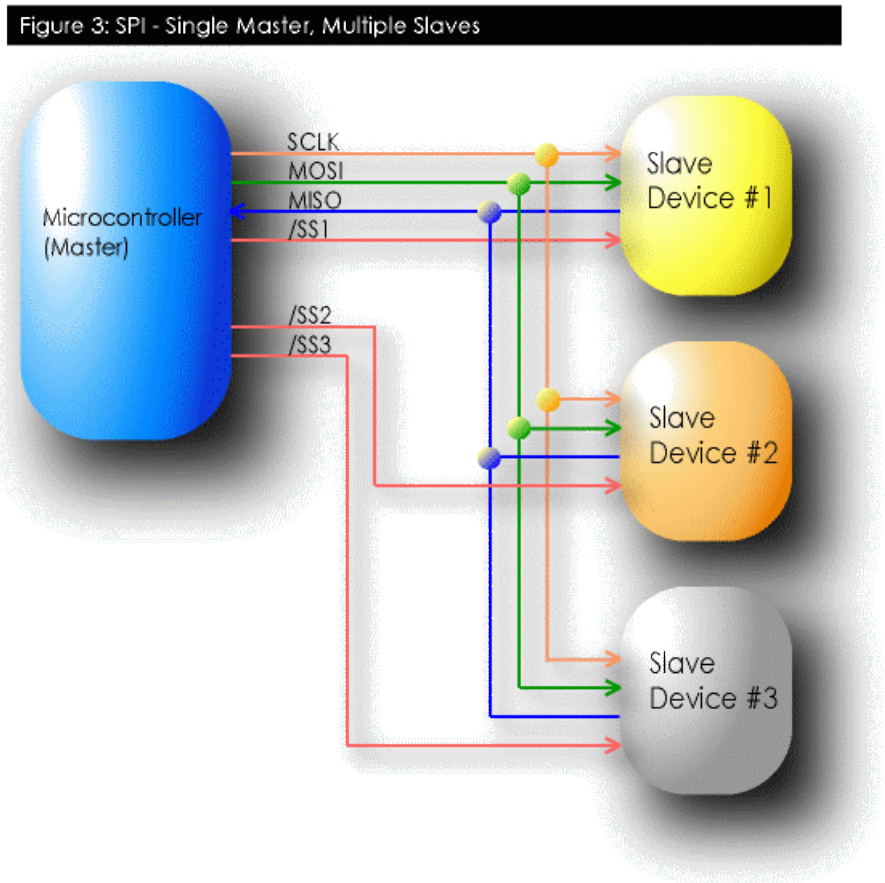
The start condition is followed by 8-bits of data with the first 7 bits being the unique slave address in standard mode. 10-bit addressing is indicated by transmitting the bit pattern 11110 followed by address bits A9 and A8. The 7-bit address field is followed by the read/not-write (R - /W) bit, which tells the slave whether to receive (0) or transmit (1) data.

This pattern of 8-data bits and 1 acknowledge bit is repeated if more bytes need to be transmitted. The device that is receiving data asserts the acknowledge signal. In case of the master transmitting, it monitors the slave's acknowledge after the last byte to be transmitted and then issues the stop condition. In case of the slave transmitting, the master acknowledges all bytes but the last one received, to indicate to the slave that transmission is to be stopped. The master then issues the stop condition.

SPI and Microwire

Both SPI and Microwire are full-duplex, (3+n)-wire serial busses. (n= # of slaves). In single master, multiple slave configurations, as shown in Figure 3, each slave device requires a dedicated Slave Select (SS) signal, which is created by the master using standard I/O pins.

Figure 3: SPI - Single Master, Multiple Slaves



A Microwire multiple slave configuration looks similar, with the difference that the master's SO pin is connected to the slaves' SI pins and the slaves' SO pins are connected to the master's SI pin.

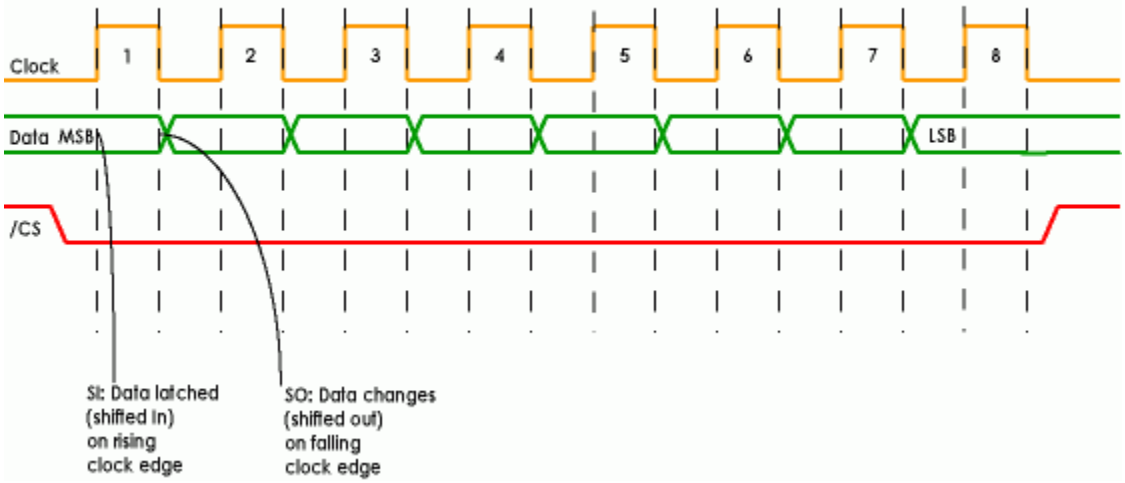
SPI Signals	Microwire Signals
SCLK- Serial Shift Clock	SK – Serial Shift Clock
MOSI - Master Out Slave In	SO – Serial Out (both master & slave)
MISO – Master In Slave Out	SI – Serial In (both master & slave)
/SS – Slave Select	/CS – Chip Select

Most SPI interfaces have two configuration bits, called clock polarity (CPOL) and clock phase (CPHA). CPOL determines whether the shift clock's idle state is low (CPOL=0) or high (CPOL=1). CPHA determines on which clock edges data is shifted in and out (CPHA=0 MOSI data is shifted out on falling edge, MISO data is shifted in on rising edge). As each bit has two states, this allows for four different combinations, all of which are incompatible with each other. For two SPI devices to talk to each other, they need to be set to use the same clock polarity and phase settings.

Why would you want 4 different settings? Two of the four settings allow the SPI interface to talk to different flavors of Microwire devices and vice versa.

Figure 4 shows the original Microwire protocol (which is older than SPI) that has fixed clock polarity and clock phase: SI is latched (data shifted in) on the rising edge of the SK clock and SO changes (data shifted out) on the falling edge. SK is always low if no data is transmitted.

Figure 4: Microwire Protocol



Many early SPI devices implemented only SPI mode 0 that does the opposite of Microwire, namely shift data out on the rising edge and in on the falling. Therefore Microwire/Plus was created that allows selecting an alternate shift clock via the SKSEL (shift clock select) bit. Nowadays Microwire/Plus with alternate shift clock is compatible to SPI mode 0 and Microwire with standard shift clock is compatible with SPI mode 1.

Both Microwire and SPI do not explicitly define any maximum data rates. Different peripherals on the market each have their own maximum speed limit - most of them in the several Mbit range. There is no means of slaves to slow the master down and also no acknowledgement on received data, like with I2C. To accommodate a wide range of SPI/Microwire speeds, microcontrollers usually feature a programmable shift clock divider.

Conclusions

SPI's and Microwire's full duplex capability and fast data rates make those interfaces very efficient and simple for single master - single slave applications. In practical applications, the requirement for dedicated slave select signals severely limits the number of slave devices that can be connected to a microcontroller. Multi-master systems significantly increase complexity and are very rarely used with those two interfaces.

I2C's lower speed and more complex protocol put it at a disadvantage in single master-single slave applications. Its weakness turns into strength if a larger number of slave devices needs to be connected or a multi-master system is needed.

All three interfaces have the advantage of being tolerant to large oscillator variations, as all data transfers are synchronized to the master's shift clock. As synchronous interfaces they are, however, limited to bridging short distances on a single PCB or between PCBs within a smaller system. When it comes to bridging larger distances or connecting external devices, asynchronous interfaces play a dominant role and we will start looking at some of them in part 3 of this series.

Microcontroller Interfaces – Part 1

Asynchronous Interfaces Overview: UART and LIN Bus

By Volker Soffel

MicroController Pros Corporation

May 12, 2003

This article is part three in a series about various microcontroller communication interfaces. In part 1 and 2 of the series, I covered three of the most prevalent synchronous interfaces: SPI, Microwire and I2C. In part 3 it's time to take a closer look at asynchronous interfaces. I will start

out with a general overview and then look in more detail at some UART based interfaces. In part 4 of this series we will the CAN interface and the move on to Ethernet and USB.

Asynchronous Interfaces

While synchronous interfaces transmit and receive data in sync with a dedicated receive/transmit clock signal, asynchronous interfaces embed the clock information into the data stream. Therefore they are characterized by the absence of a dedicated receive/transmit clock signal. For devices to communicate in sync with each other, they need to agree on the same transmission speed (kbits/sec), the same protocol (number of data bits, stop bits, parity, etc), and they need to constantly re-sync to the clock embedded into the data stream. Re-syncing is usually achieved through start and stop bits (or frames) at defined positions in the data stream. To keep in synch, it is also required that the devices' system clock is stable within a few percent - simple R/C oscillators with +-25% tolerances or more will not work.

While synchronous interfaces were designed to allow for easy inter-device connections on the same printed circuit board, asynchronous interfaces were designed for connections via cables and to bridge larger distances. Distances can range from a few meters to up to a kilometer (with copper cable) depending on the interface. Applications range from simple point-to-point connections to complex bus networks with hundreds of devices in industrial control or automotive applications. Asynchronous interfaces vary greatly in key parameters that determine their suitability for a particular application, such as maximum bus length, maximum data transmission speed, multi-master capability, maximum number of devices on the bus, network topology, fail safety, data integrity and noise immunity.

The table below summarizes some key parameters for the most popular asynchronous interfaces.

Technology	*Multi-Master Support*	*Typical Data Rates*	*Network Topology*	*Cable*	*Max. # of devices/ segment*	*Max. Cable Segment Length*
Ethernet	yes	10/100/1000 Mbps	Star (Hub or Switch)	UTP3/UTP5/ UTP5e	2 (point-to-point)	100m (300'). Max 5 segments (4 repeaters): 500m (1500')
Ethernet	yes	10 Mbps	Daisy Chain with terminating resistors	RG58 Coax	30	185m (555'). Max. 5 segments (4 repeaters): 925m (2775')
RS485 (UART based)	no	0.3 kbps …1 Mbps	Daisy Chain with terminating resistors	STP, 24AWG	32 (2-wire bus), 64 (4-wire bus)	1330m (4000') @64kbps, much more with repeaters

Technology	Multi-Master Support	Typical Data Rates	Network Topology	Cable	Max. # of devices/ segment	Max. Cable Segment Length
RS422 (UART based)	no	0.3 kbps …1 Mbps	Daisy Chain with terminating resistors	STP, 24AWG	10	1330m (4000')@64kbps, much more with repeaters
RS232C (UART based)	no	0.3 kbps …128 kbps	Point-to-Point	Various (ribbon, STP, UTP)	2	15m (45') @ 19.6 kbps, ~1.5m (4.5') @ 128 kbps
LIN (UART based)	no	20 kbps	Daisy Chain	single wire	16	40m (120'), more with repeater
CAN	yes	20 kbps …1 Mbps	Daisy Chain with terminating resistors	STP/UTP	128	1000m (3000') @40kbps, 40m (120') @1Mbps, much more with repeaters
USB	no	1.5/12/480 Mbps	Star	STP	2 (point-to-point), 127 per root host controller via USB hub	5m (15') Max. 7 segments (6 repeaters/hubs) 35m (105')

The OSI Layer Model

Before I continue covering asynchronous interface in more detail, let's take a short look at the OSI (Open Systems Interconnect) Layer Model, so that we have a common definition of terminology. The OSI layer model, developed by the ISO ((International Organization for Standardization), defines a framework for communications which has seven layers: 1-the physical layer, 2-the data link layer, 3-the network layer, 4-the transport layer, 5-the session layer, 6-the presentation layer, 7-the application layer. Control is passed from one layer to the next. A communication begins with the application layer on one end (for example, a sensor reading changes). The communication is passed through each of the seven layers down to the physical layer (which is the actual transmission of bits). Most interfaces that are integrated on microcontrollers only cover the data link layer, with the physical layer being implemented externally (with some exceptions). In some cases one and the same data link layer might be combined with different physical layer implementations to create different interface standards. In microcontroller applications layers 3 to 7 are typically implemented in software (with some hardware support).

U(S)ART

The oldest and still most predominant asynchronous interface is the UART (Universal Asynchronous Receiver Transmitter). Some implementations support both a synchronous and asynchronous mode - then it is a USART (Universal Synchronous Asynchronous Receiver Transmitter). The same thing with just a different name is the Serial Communication Interface (SCI). With reference to the OSI model, a UART implements the data link layer (layer 2). The physical layer (layer1) is covered by several driver standards that all utilize the UART data link layer, among the most popular are RS232C, RS485 and RS422.

USARTs that are integrated on microcontrollers support data rates ranging from a few hundred bits per second (bps) up to 1.5Mbps. UART systems are typically either pure point-to-point connections of 2 devices (RS232C), or single master- multiple slave bus systems (RS422, RS485). It is possible to built multi-master RS422 and RS485 systems, but it requires the development of your own software protocol to handle bus arbitration (in case multiple masters want to send at the same time). Other interfaces, like CAN and Ethernet, handle bus arbitration in hardware and are therefore better suited for multi-master systems.

The basic UART data layer is very simple. A data packet consists of one start bit, 7,8 or 9 data bits, an optional parity bit (7 and 8 data bit modes only) and 1or 2 stop bits (as a curiosity some UARTs also support $7/8^{th}$ for the last stop bit).

All the UART data link layer has to offer in terms of error checking is a simple (optional) parity bit. There is no acknowledgement from the receiving side, unless you implement some software protocol with acknowledgement or hardware handshaking mechanisms via standard microcontroller I/O pins. Better error checking also requires you to do it in software, which for CRC checksums can be pretty resource intensive.

The advantage of UART's simplicity is that it can be implemented in a relatively small silicon area and therefore commands a much smaller price premium than CAN, USB or Ethernet.

The distances that can be bridged with a UART interface depend on the physical layer selected.

It is possible to connect two UART devices using the chips' CMOS logic signals. In that case a UART is not much different from the synchronous interfaces and only very short distances can be bridged (on the same board or within a closed system).

RS232C allows for cable connections from 1.5m up to 20m, depending on the data rate selected, the quality of the cable and the output voltage swing of the RS232C transmitter (the RS232C spec provides much leeway on this).

RS485 and RS422 can bridge up to 1200m (4000'), again dependent on data rate, cable quality and physical layer driver/receiver characteristics (not all are created equal). To achieve such long distances and to increase noise immunity both RS422 and RS485 use differential data transmission.

LIN (Local Interconnect Network)

The newer LIN standard is a close cousin of the UART. LIN was specified by a consortium of automotive and semiconductor companies as a lower cost supplement to CAN with an emphasis on data integrity, error recovery and fail safety. The LIN spec defines both physical layer (layer1) and data link layer (layer 2).

LIN is a single master – multiple slave system. The "bus" is a single wire with reference to ground. The maximum bus length is therefore limited to 40m (120') with a maximum of 16 devices sharing the same bus.

LIN's data link layer provides improved control and error checking over a UART interface. It should be noted that all of LIN enhancements over a standard UART data link layer can be implemented in software, thus allowing you to do LIN with many of the standard integrated UART implementations found on microcontrollers - at the expense of quite some software and resource overhead.

Data is transmitted in fixed format message frames of selectable length. A message frame starts with a sync break signal, followed by synchronization and ID byte. Next follow 2, 4 or 8 data bytes and the CRC checksum byte. Sync break, Sync byte and ID Byte are always send by the master. The identifier describes the meaning of data. Slaves receive or transmit data based on the ID send by the master.

The Sync byte is used by the slave microcontrollers to automatically determine the bit rate and compensate for any clock tolerances. Through this trick it is possible to use high tolerance, cheap R/C or integrated oscillators for the slaves.

The LIN Bus specification 1.2 defines five different frames: data frame, sleep frame, wake-up frame, extended identifier frame and command identifier frame.

A LIN Master controller must be able to detect the following error conditions: bit error, identifier parity error; and, if expecting data from a slave, slave not responding error and checksum error. When receiving slaves must detect identifier parity errors and check sum errors; and bit errors when sending.

Some microcontroller manufacturers now integrate "enhanced" UARTs that provide some basic hardware support for specific LIN requirements, like, for example, generation of an interrupt upon receiving the synch break signal and automatic baud-rate calibration on every sync byte.

LIN's physical layer is basically a slew rate controlled NPN transistor whose collector is connected via a pull-up resistor and diode to the car battery plus terminal and whose emitter is connected to ground. The microcontroller's receive/transmit pin drives the basis of the transistor. The transistor has to be able to survive 40V across its collector-emitter connections and must be able to sink 200mA when switched on. This physical layer can of course be implemented externally to the microcontroller, but you can also find microcontrollers that have the physical interface integrated on-chip.

About the Author

Volker Soffel is the General Manager of MicroController Pros Corporation (uCPros). uCPros offers services in: Electronic system design with a focus on embedded systems; marketing and management consulting; employee training; technical writing and translations. uCPros also publishes a free monthly Embedded News Digest email newsletter, covering the latest events in the microcontroller industry.