Chapter 9
Principles of Serial Communication

Serial channels are, without doubt, the main form of communications used in digital systems nowadays. Diverse forms of serial communication formats and protocols can be found in applications ranging from short inter- and intra-chip interconnections, to the long range communication with distant spaceships traveling to other planets. Virtually, all forms of communications used nowadays in consumer electronics are supported via serial channels. The Universal Serial Bus (USB), Blue-tooth, Local Area Networks, WiFi, IrDA, the traditional RS-232, FireWire, I^2C, SPI, PCIe, and many other protocols and formats are all based on serial links.

9.1 Data Communications Fundamental

A serial communication channel, when transmitting an n-bit character, instead of using n simultaneous signal links, one per bit, as is done in a parallel communication, uses only one signal link, with the n bits composing the character sequentially transmitted over the channel. Each bit serially transmitted takes a pre-determined amount of time t_{bit}, requiring $n \cdot t_{bit}$ seconds to transmit the entire character. Figure 9.1 illustrates the conceptual[1] difference between a parallel and a serial 8-bit channel transmitting character 0A5h.

The transmission rate of a serial channel is determined by the amount of time taken to transmit each bit (t_{bit}). Two speed metrics commonly used in serial channels are the *bit rate* and the *baud rate*. The bit rate expresses the number of bits-per-second or *bps* transmitted in the channel. Given t_{bit} we can obtain the bit rate as $bps = 1/t_{bit}$. For example, a serial channel with $t_{bit} = 10$ ns in NZR will have a bit rate of 100 Mbit-per-second or 100 Mbps.

The second metric, the *baud rate*, refers to the number of symbols per second sent through a serial channel. When each symbol represents one bit, as is the case of

[1] The actual appearance of bits in the channel will depend on the signal carrier and modulation scheme. Figure 9.1 illustrates a voltage carrier in Non-Zero Return (NZR) format.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_9,

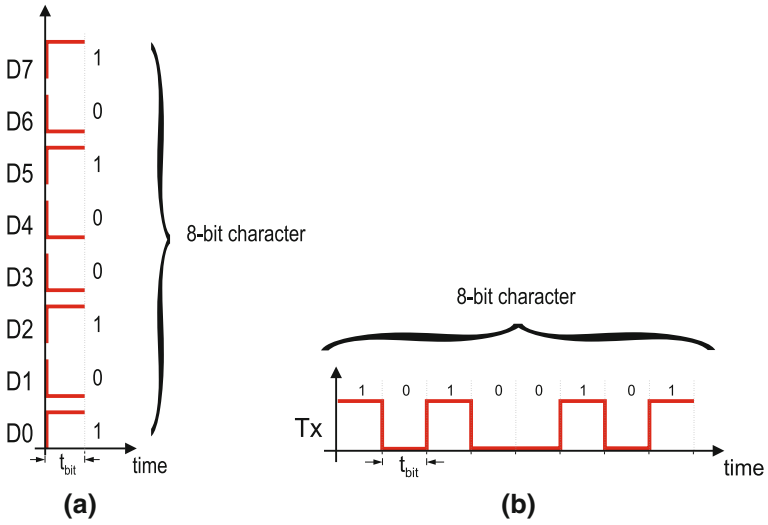


Fig. 9.1 Signal diagrams for serial and parallel communication channels. **a** Parallel. **b** Serial

NZR, then bit rate and baud rate are the same. By using more complex modulation schemes it is possible to encode more than one bit per signal change. For example, schemes based on phase modulation, like Phase-shift Keying (PSK), can encode two or more bits per phase change. In such cases the bit rate will be higher than the baud rate.

Communication links can be implemented in diverse forms. Wired channels can use either *single-ended* or *differential* links. When a *single-ended* connection is used each link consists of ground referenced wire. An n-bit, single-ended parallel channel would require $n + 1$ wires to make the connection: one for each link plus one for the ground reference. For the same type of connection, a serial channel would use only two wires: one for the link and one for the ground reference.

In a *differential channel* each link is represented by the voltage difference between two wires. This type of link is more robust than a single-ended line due to the common mode rejection ratio (CMRR) of the receiving differential pair. A parallel channel using differential signaling would require $2n+1$ wires to make the connection, versus only three used in a serial connection. Figure 9.2 shows the difference between a single-ended versus a differential link.

A wireless link could be optical like in infrared transceivers, acoustic like in underwater channels, or via a radio-frequency (RF) like in WiFi. In either of these cases a common ground reference between transmitting and receiving ends is not necessary, but still n links would be needed for establishing a parallel connection, versus only one for a serial link.

This simple analysis reveals one of the fundamental advantages of serial channels over parallel: their cost. It is much cheaper to have a single serial link for transmitting

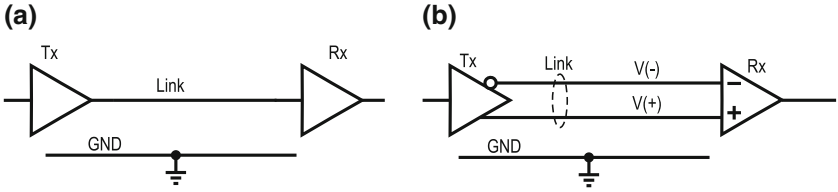


Fig. 9.2 Wired connection modalities for serial channels. **a** Single ended. **b** Differential

a 16-bit character than sixteen simultaneous links for a parallel transmission. The cost advantage of serial links escalates as the distance they cover increases.

Assuming equal bit times in serial and parallel channels, in principle, parallel communications would be n-times faster than serial. Up to a certain bandwidth and distance this comparison holds true. However, as communication speeds and/or distances increase, transmission line effects, cross-talk, and other noise manifestations begin to take a toll on parallel channels that limit their practical application. This has been one of the drivers behind the proliferation of high-speed serial links in modern communication channels.

9.2 Types of Serial Channels

Serial channels are said to be *simplex, half duplex, or full-duplex*, depending on their type of connectivity.

A simplex serial channel transmits permanently in only one direction over a dedicated link. One end of the communication channel features a dedicated transmitter while the other has a dedicated receiver, as illustrated in Fig. 9.3. Simplex channels have no means of acknowledging or verifying the correct reception of data. Examples of simplex channels include conventional radio and TV broadcasts channels, write-only printers and displays, or read-only devices interfaced to a CPU.

A half-duplex serial channel features a single link that allows communication in either direction, but only in one direction at a time. Both ends of the channel feature interfaces operating as *serial transceivers*, denoting their ability of working as either a transmitter or a receiver. Whenever the transmission direction needs to

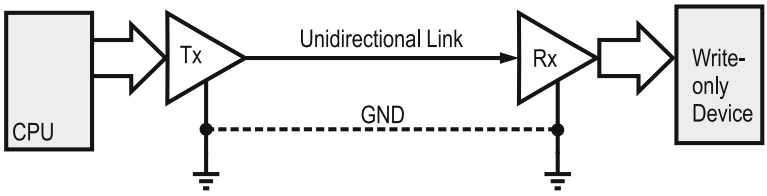


Fig. 9.3 Simplex serial channel connecting a CPU to a write-only device

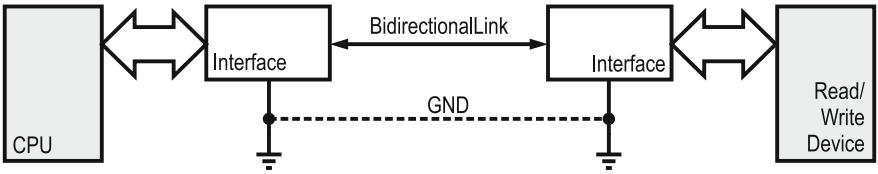


Fig. 9.4 Half-duplex serial channel connecting a CPU to a read/write device

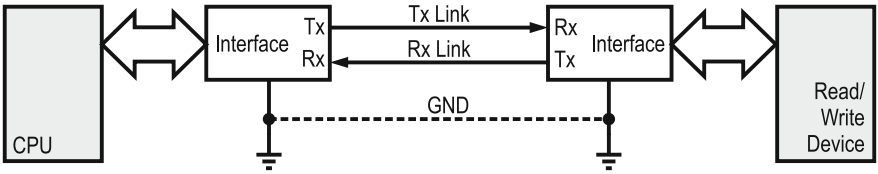


Fig. 9.5 A full-duplex serial channel connecting a CPU to a read/write peripheral

be changed, the interfaces in both ends need to switch their modes. Such a change requires rules to avoid having both ends attempting to simultaneously transmit. The set of rules deciding how devices at the ends of a communication channel behave and how they organize and interpret the data in the channel is called a *communications protocol*. Protocols avoid conflicts but also introduce a certain amount of latency in the communications. Figure 9.4 illustrates a topology for a half-duplex channel, denoting its single link. The ground connection is required for wired channels.

Full-duplex serial channels feature two separate links, one dedicated to transmit and another to receive, allowing for simultaneous communication in both directions. The interfaces at each end of the channel have the ability of handling both links simultaneously, requiring no switching, and therefore enabling uninterrupted bidirectional communication. Most serial connections in contemporary embedded systems use full-duplex channels. Figure 9.5 shows the topology of a full-duplex serial channel. Note the Tx→Rx and Rx←Tx connections in their interfaces.

The links used in each of these communication modalities could either be wired or wireless. In the case of wired links, both sides of the channel must have a common ground connection, and therefore, a ground wire must be shared between both ends of the channel. In wireless links such as RF, optical, or other medium, there is no need for a common ground.

The links illustrated in Figs. 9.3, 9.4 and 9.5 correspond to point-to-point topologies, as they contain only two devices, one at each end of the channel. This is the simplest topology in a communication channel and a common one.

Other topologies may incorporate more than two devices in the channel. These are called multi-point or multi-drop links. There are several topologies for multi-point channels, most of them studied as computer networking technologies. In embedded applications, multi-drop serial buses are quite common. Later in this chapter we discuss some of the most common serial bus topologies found around microcontrollers.

9.3 Clock Synchronization in Serial Channels

All serial channels require a stable clock signal to establish their transmission and reception rates and to internally synchronize the operation of their interfaces.

Depending on how the clock signal is provided, a channel might be asynchronous or synchronous. Asynchronous channels use independent clock generators at each end, while synchronous transmit the clock along with the data.

Both asynchronous and synchronous serial communication protocols divide a message to be sent through the channel into fundamental units called *data packets* or *datagrams*. Each packet contains three sections: a header, a body, and a footer. The header and footer are leading and trailing handshaking information fields added by the communication protocol to the portion of the message making the body of packet being transmitted.

A packet header includes data indicating the beginning of the packet, optional addressing information necessary in multi-drop channels, and an also optional message length and type fields needed in protocols accepting different types and lengths packets. A packet footer fundamentally contains data delimiting the packet and an optional field with error checking information. Headers and footers in asynchronous and synchronous channels vary in their length format, depending on the protocol. Below we describe distinguishing characteristics of each of them.

9.3.1 Asynchronous Serial Channels

An asynchronous communication channel, as indicated earlier, has independent clocks at each end of the channel. To make possible their communication, the interfaces at each channel end are configured to produce the same data rate, as well as having the same parameters defining the exchanged data packets. Figure 9.6 illustrates the topology of an asynchronous, full-duplex serial channel denoting its independent baud rate clock sources.

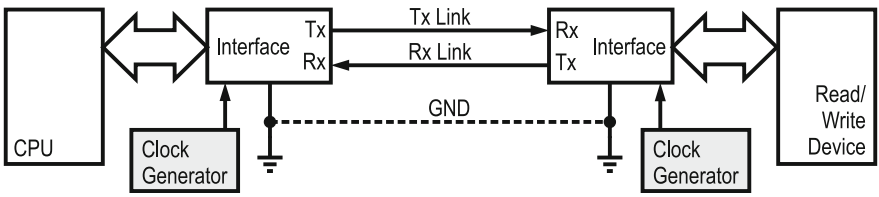


Fig. 9.6 An asynchronous serial channel denoting independent clock sources at channel ends

9.3.2 Asynchronous Packet Format

Asynchronous data packets have a simple structure. Their headers contain only one bit called a *start bit*, which is always a logic zero or space symbol.[2] The packet body contains a five- to eight-bit character arranged from its least- to its most-significant bit. The footer contains an optional *parity bit* that allows for error checking and one or more *stop bits* to indicate the packet end. A stop bit is always a logic one (mark). Figure 9.7 shows the format of an asynchronous packet.

The idle state of an asynchronous channel is in the mark level (logic high). A *start bit* indicates a channel transition from idle to active. A valid start bit has a length of one full bit time. When a one-to-zero transition is detected while in the idle state, this is assumed to be a start bit. To confirm that in fact a start bit is being received, the line is sampled again, half t_{bit} later, at the middle of the bit time, and if it is still found low, then a valid start bit is accepted. Otherwise, the condition will be flagged as a framing error.

After a start bit has been validated, the line is sampled at intervals separated by one bit-time to detect all remaining bits in the packet. In the transmitter side, the length of each bit is determined by the transmission rate, obtained from the clock in the transmitter side. In the receiver end another clock establishes the reception rate. Although transmission and reception clocks are independent, both sides must agree on the same bit rate to make communication possible. The interfaces at each end of the channel use frequency dividers to obtain the correct bit rate from their corresponding local clocks.

Due to the drift that usually develops between independent clocks, the length of asynchronous packets is kept short, and the start bit re-synchronizes them. Also, the sampling rate at the receiver end is always higher than the bit rate of the transmitter by a factor k, with typical values of 16, 32, or 64.

The number of bits per character in most contemporary channels is seven or eight, although we might find channels that still maintain compatibility with early teletype machines and thus use five- or six-bit characters. Seven-bit characters support only the base ASCII set, while those with eight bits also support extended ASCII symbols. Both ends of the channel must agree on the number of bits per character to be able to communicate.

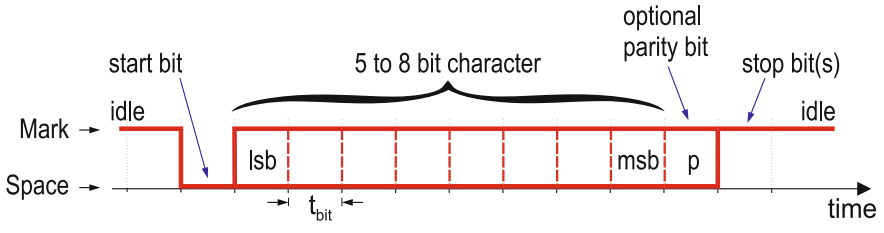


Fig. 9.7 Format of an asynchronous packet

[2] In digital communications jargon, a logic zero is called a *space* and a logic one a *mark*.

9.3.3 Error Detection Mechanism

The parity bit field is optional, meaning that communicating devices must agree on wether using it or not. Parity check is an elementary error checking mechanism that allows for detecting single bit flips in a transmitted character. It counts the number of bits set (logic one) in the character an adds an extra bit, called the *parity bit*. The parity bit is set or clear according to the character itself to ensure every transmission will have a pre-determined parity: even or odd.

When the sides of a channel agree on enabling parity check, they must also specify wether parity will be even or odd. In even parity, the parity bit is set or clear to always make even the number of ones in the transmitted character plus parity. In odd parity, the parity bit forces the number of bit in one in the transmitted bitstream to be odd. When a character completes reception, the receiving end strips out the start and stop bits and then calculates the parity of the character plus parity bit. If the check matches the agreed parity, the parity bit is stripped and the character accepted. It the parity does not match, the condition is flagged as a *parity error*.

The last field in the packet, the *stop bit*, marks the end of the packet. Stop bits are always logic high and introduce a forced idle interval between characters. The length of the stop bit might be one or two bit times, depending on the interface, but both sides must agree in the minimum length. Failure to detect the minimum stop bit length is another condition for triggering a framing error.

Example 9.1 (Asynchronous Bitstreams): *Consider transmitting the 8-bit character 0B7h with even parity enabled and one stop bit. The bitstream to be transmitted after adding the parity bit would be 111011010 (before attaching the start and stop bits). These leftmost eight bits correspond, from left to right and starting with the least significant bit, to the character to be transmitted; followed by the parity bit in the rightmost position. In this case the parity bit was zero because the character already had even parity. Note that even parity does not mean the character is even. The complete bitstream sent over the channel, including start and stop bits, is illustrated in Fig. 9.8.*

If instead, character 046h were to be transmitted, the resulting bitstream plus parity would be 011000101 (before start and stop bits). In this case the parity bit is set, forcing the parity of the transmitted character plus parity bit to be even.

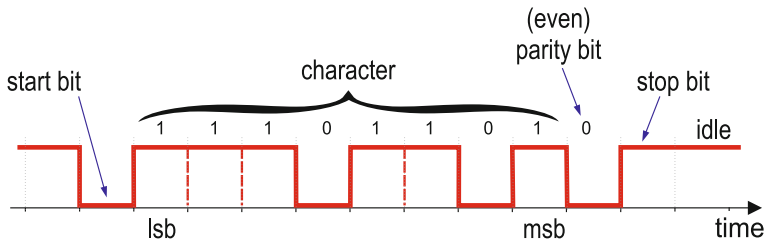


Fig. 9.8 Transmitting character 0B7h with even parity

Start, parity (if enabled), and stop bits are automatically inserted by the serial interface module in the transmitter side, and also automatically removed by the receiving interface. Similarly, the detection of potential errors like framing of parity are also automatically performed by the adapter. The user software only provides the characters to be sent or processes the received characters. When an error is detected, user software shall decide the proper course of action, if any. An interface module for an asynchronous serial communication channel is called an UART: Universal Asynchronous Receiver and Transmitter. A discussion of UART structure and capabilities is provided in Sect. 9.3.4.

In summary, devices communicating over an asynchronous serial channel must program their UARTs to agree in the baud rate, the number of data bits per character, wether or not parity will be used or not, and the number of stop bits to use when communicating. If parity is used, then the type of parity expected must also be the same in both devices.

9.3.4 Asynchronous Communication Interface Modules

In order to have a processor communicating over a serial channel, an interface module is needed. Serial interfaces, or adapters, fundamentally convert data from parallel to serial and viceversa, allowing the CPU to communicate through the serial channel.

In asynchronous serial channels the most widely used interfaces are UARTs. UARTs are available as stand alone peripherals or as embedded modules within MCUs either by themselves or as part of a larger communication peripheral. Common examples of such bundles are USARTS (*Universal Synchronous/Asynchronous Receiver and Transmitter*), which combine in a single module serial interfaces supporting asynchronous and synchronous communications channels.

A USART operates in one mode at a time, either asynchronously or synchronously. In asynchronous mode is just an UART, while their synchronous operation will support a particular protocol, such as SPI, I^2C, or other. The next subsections discuss the structure, functionality, and usage of UARTS. A similar discussion for synchronous adapters is included in Sect. 9.4.

9.3.5 UART Structure and Functionality

The structure of a UART combines the functionality of a dedicated transmitter and a dedicated receiver into a single module to provide a serial interface with full-duplex capability. Figure 9.9 shows a diagram of a simplified UART module denoting its most important components.

The internal components of a UART are centered around two shift registers: a Parallel-Input Serial Output (PISO) in the transmitter, and a Serial-Input Parallel-Output (SIPO) in the receiver. To perform a transmission, the CPU writes into the

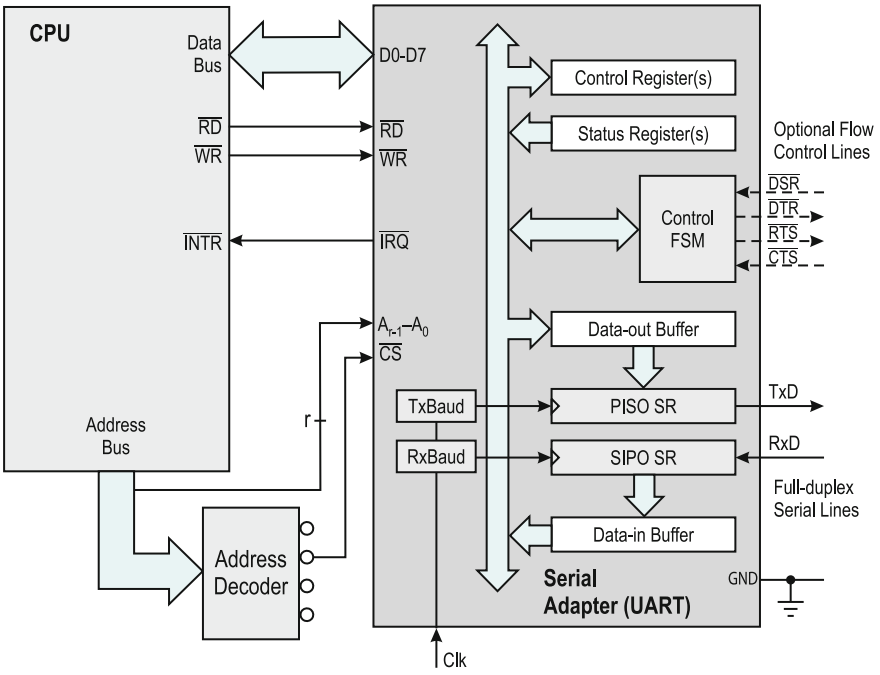


Fig. 9.9 General organization of a Universal Asynchronous Receiver and Transmitter (UART)

Data-out buffer the characters to be sent. Each character, taken one at time, is then framed with header and footer portions by the UART finite state machine (FSM) to form an outgoing packet and places it into the PISO shift register. In this step, the control FSM calculates and inserts the parity bit (if enabled) as part of the footer when chosen. The contents of the PISO is then transmitted over the serial channel at the rate configured in the transmitter baud rate generator. When all characters in the data-in buffer complete transmission, the condition is flagged by a TxReady flag, denoting that the transmitter portion of the UART is ready to accept a more characters to be sent. Writing the data-out buffer automatically clears TxReady. Simple UARTs have a single character buffer, while more advanced devices can accept multiple characters, reaching even kilobyte capacities.

In the receiving process a valid start bit resets the SIPO and begins loading it with an incoming packet. When all bits have entered the SIPO SR, the FSM strips out the start and stop bits, performs a parity check (if enabled), removes the parity bit (if any), and places the received character in the data-in buffer where the CPU can read it. This event is flagged by a RxReady flag, indicating the receiver is ready with a newly received character. Reading the data-in buffer automatically clears RxReady. If parity or framing errors were detected in the reception process the condition would be flagged.

If during the reception process the data-in buffer were filled faster than the CPU were retrieving the incoming characters, some data could be lost as incoming characters could overwrite previously received data no yet retrieved by the CPU. Such a condition is a reception error known as an *overrun error*. Data-in buffers with multi-character capability help alleviating the situation but the CPU has to promptly remove received characters to avoid the data loss.

The PISO and SIPO shift registers are clocked by the transmitting and receiving clocks, respectively, establishing the transmission and reception data rates. All indicators denoting the transmitter, receiver, and error status are accessible through the designated status registers. The control register(s) allow for configuring all UART parameters defining its operation. The number and functions of control and status registers will depend on the UART capabilities.

Internal control register(s) allow for configuring the diverse options in the module such as character length, enabling/disabling parity, parity type, stop bit length, and baud options. In addition, some of the control bit allow for enabling the transmitter and/or receive portions of the UART and their ability to generate interrupts to the CPU. The specific list of setting options changes depending on the specific unit being configured.

The status register(s) allow for determining specific channel and device status. Typical flags include those indicating error conditions, such as overrun, framing, and parity. The transmitter and receiver statuses are also held here and the pending interrupts. As with control registers, the topology of status registers will vary from one unit to another. Actual examples of such registers are included in the discussion of MSP430 communication interfaces in Sect. 9.5.

9.3.6 UART Interface

A UART interface can be seen from three different perspectives: the CPU interface, the clock interface, and the channel interface. The CPU interface becomes relevant when interfacing a UART chip to the CPU address, data, and control buses. The clock interface deals with the way the time base signals used to generate the transmission and reception baud rates are provided. The Channel-side interface connects to the actual serial channel, and becomes relevant in every UART application. The paragraphs below discuss considerations for each of these interfaces.

CPU-Side Interface

A UART connects to the CPU buses through a set of bidirectional data lines (typically 8-bit) that transfer data into or out from the adapter, read/write control lines to specify the transfer direction, and selection lines, A_{r-1}–A0 and $\overline{CS}$, like any other I/O interface. This model is illustrated in Fig. 9.9. Line $\overline{CS}$ is driven by an address decoder that assigns the module base address. Lines A_{r-1}–A0 allow for addressing 2^r

internal module registers. For example, if the module base address were 0F520h, assuming a byte-wide data bus, the connection of line (A0) would enable internal addresses 0F520h and 0F521h. If lines A0 and A1 were provided, the module would host four consecutive locations starting at its base address.

The CPU side of the UART also includes one or more interrupt request lines that allow for the UART to place service requests to the CPU, enabling interrupt-based servicing of the adapter. Depending on the particular UART used, one or multiple independent $\overline{\text{IRQ}}$ lines could be provided. A single $\overline{\text{IRQ}}$ would feature a single multi-source vector for Rx, Tx, and possibly error events. Multiple request lines allow for separating individual vectors for different UART events.

When UARTs are embedded within MCUs, the CPU-side interface is hidden inside the chip and the developer only deals with the channel side lines. Clock signals are in most cases internally configured via control registers.

Clock Interface

All UART interfaces also include one or more clock input lines that feed the baud rate generators inside the module. These clocks provide the base frequency f_{clk} used by internal frequency dividers TxBaud and RxBaud to establish the transmitter and receiver baud rates. Most MCUs derive this frequency from the system clock generator, although some might allow dedicated external oscillators. The frequency dividers might be integral part of the UART or provided from an on-chip timer. Section 9.3.7 on p. 487 provides additional details about the selection and configuration of the clock source.

Channel Interface

The main lines in the channel side of a UART are TxD and RxD, which along with the signal ground (GND) carry the incoming and outgoing serial streams. TxD (Transmitted Data) is the serial output and is driven by the PISO, while RxD (received data) is the serial input, driving the SIPO SR input.

Dedicated UART chips use single-functioned pins for these lines. In most MCUs, however, these lines are shared with GPIO or other functions, making necessary to configure them as UART lines when intended to be used as such.

Some UARTs may also include in their channel side a set of handshaking lines that allow for modem and/or hardware flow control. These include:

DSR Data Set Ready, an input to the UART indicating an external data communication equipment (DCE, originally a modem[3]) is ready to receive data.

[3] MODEM = MOdulator-DEModulator: a communication interface that converts digital logic levels into tones and viceversa, allowing for sending them over a telephone line.

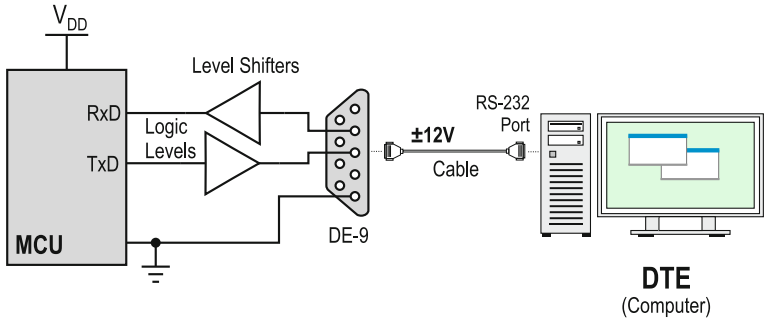


Fig. 9.10 Level shifting for physical standard compliance: RS-232C

DTR Data Terminal Ready, an output from the UART (originally designated as a data terminal equipment or DTE) indicating it is ready to operate.

RTS Request to Send, a UART output indicating to the modem it desires to send data.

CTS Clear to Send, a UART input from the DCE or modem indicating it will accept new data. CTS is asserted in response to a RTS.

These lines were originally introduced for modem control, and along with the signal ground (GND) and two other lines not pictured in Fig. 9.9 (Ring Indicator (RI) and Data Carrier Detected (DCD)) conformed the signal set designated by the Electronics Industries Association (EIA) for the RS-232C standard. Some dedicated UART chips still in the market, like National Semiconductor's 8250 and 16550 directly support these lines. When provided, these lines can be used for hardware flow control in the channel. MCU embedded UARTs rarely feature dedicated channel handshaking lines. If for some reason these were needed, their functionality can be implemented with GPIO lines under software control.

All signals running into or out from a UART in the channel side must use the same voltage levels as those of the digital logic levels in the UART chip itself, dictated by the supply voltage. For example, a dedicated UART chip operated at 5VDC will send and tolerate TTL compatible signals. A UART embedded in a 3.3VDC MCU, will provide and accept signals in a 0–3.3 V range. Using logic levels directly to transmit and receive serial data will limit the maximum distance signals can travel to only a few inches before being garbled by noise, particularly as baud rates increase. To enable longer transmission distances, different voltage levels and signal formats become necessary.

Physical standards for serial communication use large voltage swings and/or signalization schemes that improve the noise immunity of the signals transmitted over the channel. These levels, are however, not directly compatible with the logic level signals of bare UART chips, requiring level shifters or voltage translators for their interconnection. This situation is exemplified in Fig. 9.10 for the particular case of

an RS232 channel. Failing to observe this rule would cause irreversible damage to the UART or MCU chip. Section 9.3.8 on p. 489 discusses the physical requirements of RS-232C and other standards used in serial channels, describing required voltage levels, current limits, and signalization schemes for proper operation.

9.3.7 UART Configuration and Operation

Configuring and operating a UART from a software standpoint is a simple task when a systematic approach is followed to configure, enable, and access the UART registers.

Configuration

Configuring the UART requires the following steps:

- **Choosing the clock source for the baud rate generator(s):** Microcontrollers might have several clock sources to choose from. Although one of them is typically set as default source, it is worth verifying that it will satisfy the channel requirements or else choose one that would result appropriate. This will also establish the input clock frequency f_{clk} to the baud rate generator.
- **Configuring the baud rate generator:** With (f_{clk}) known, achieving a given baud rate *BR* only takes dividing f_{clk} by a factor N determined as:

$$N = \frac{f_{\text{clk}}}{BR} \tag{9.1}$$

  Simple UARTs frequently use a timer to set the baud rate. In such cases, N would be set through the product of the timer prescaler (if any) and the timer channel terminal count. It is always desirable to choose an f_{clk} value that when divided by *BR* yields an integer value of N. When a non-integer N results, the nearest integer is taken, resulting in an actual baud rate deviated from the desired value. This deviation introduces a *baud rate error* with respect to the other channel end. Small baud rate errors ($\pm 2\,\%$ to $\pm 4\,\%$) are usually tolerable, particularly at low baud rates. At higher baud rates, as the bit time becomes shorter, increasing baud rate errors translate into severe reception errors that might render the channel unusable. Baud rate "sweet" f_{clk} values producing integer N with $0\,\%$ error are listed in Table 6.4 on p. 279.

  Modern UARTs include more complex baud rate generators able to handle the fractional part of N while introducing small error. In such units, although the principle for calculating N is the same as above, the particular scheme used to handle the fractional part of N changes with the UART topology.

- **Choosing the correct synchronization mode:** When a UART is contained within a USART, it becomes necessary to configure the corresponding bits in the control register to make the module operate as a UART.
- **Choosing and configuring parity check:** As parity check is optional, both channel sides need to agree on whether or not it will be used. If enabled, both sides must be configured to use the same type of parity (even or odd), and configure the corresponding control bits.
- **Configuring character and stop bit length:** Many UARTs can accommodate different character lengths, as discussed earlier. Both sides must be configured for the same character length. Similarly, UARTs with configurable stop bit lengths should agree on the same length. In some cases this is a soft requirement as even when using more than one stop bit, many UARTs only sample the first stop bit.

Enabling

Depending on the communication direction and whether operation will be handled via polling or via interrupts, several enabling actions are required. Some UARTs require enabling their transmitter and/or receiver portions individually to be operational, some enable the entire UART or USART. If interrupt servicing were desired, the transmitter and/or receiver interrupt enable bits must be set. For simplex communication, only the corresponding Tx or Rx is enabled, while for duplex operation both must be enabled. Moreover, if error conditions were to trigger interrupts in the UART operation, their enable bit(s) shall also be set. Many UARTs use multi-source shared vectors in their operation, requiring properly designed ISRs capable of identifying the trigger source.

Operation

The actual operation of the channel is fairly simple. When polled operation is being used (although discouraged), the user software must always poll the readiness of the transmitter and receiver flags before any operation in the UART data buffers. Before writing a new character for transmission in the data-out buffer, the TxReady flag must be polled to determine readiness. Otherwise data loss might occur.

When receiving by polling, the RxReady flag must be polled before retrieving incoming data from the data-in buffer to avoid repeated retrieval of the same data. After each character is received, error flags must be checked to determine if an error was detected in the retrieved character. The user software decides the action to take place in the case of detecting an error.

When interrupt-based operation is enabled, the software is even simpler. Activation of the receiver interrupt means a new character was received and therefore the ISR can directly proceed with its retrieval, polling the error flags if they do not trigger interrupts by themselves. In an analogous way, a transmitter IRQ signals the

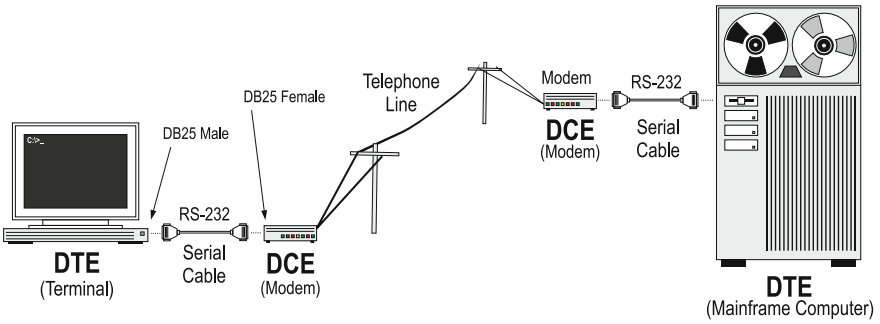


Fig. 9.11 RS-232 DTE-DCE connections used in early serial data communication channels

readiness of the transmitter to accept new characters to be sent through the channel and the ISR can directly proceed to write into the data-out buffer.

9.3.8 Physical Standards for Serial Channels

The physical-level standard for a serial channel establishes the signal definition for carrying bits from one end to another. In wired channels it establishes voltage levels, signalization formats, connectors, and line definitions and notations. In wireless channels, the physical standard defines carriers, frequencies, modulation schemes, and signal strength, among the most important parameters.

RS-232

For long time, the most used serial standard in computers and peripherals was the RS-232. The standard was originally developed to specify the signal voltages, timing, handshake protocol, and mechanical connectors between terminals and modems. In those days, computer communication was fundamentally used to connect terminals and peripherals to remotely located mainframe computers via modems over telephone lines, as illustrated in Fig. 9.11. As such, the RS-232 standard assumed connections would mostly happen between a Data Terminal Equipment (DTE) (computer or terminal) and a Data Communication Equipment (DCE) (modems, printers, or other serial devices).

The original RS-232 specification called for a 25-pin male connector (DB25) on the DTE and a female DB25 connector on the DCE with signals at ± 25 V. Due to space and cost savings manufacturers began using a 9-pin connector instead, which omitted the least frequently used signals. By the time of the advent of the RS-232C revision, which reduced the voltage requirement to ± 12 V, the most widely used serial connector had only nine pins. Figure 9.12 shows the organization of signals in

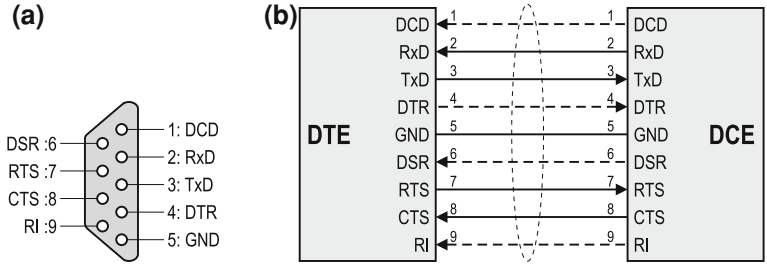


Fig. 9.12 Organization of RS-232C signals in a DE9 connector and serial cable. **a** DE-9 Connector **b** Standard serial cable

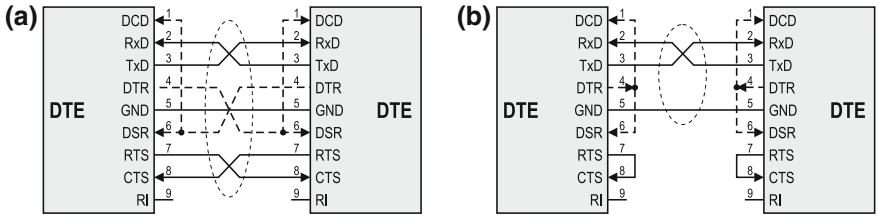


Fig. 9.13 Null-modem connection with and without handshaking. **a** Null-modem cable, **b** null-modem without handshaking

a DE9 RS-232C connector and how a standard DCE-DTE modem cable is wired. Observe that in a standard modem connection the inputs on the DTE are outputs on the DCE and viceversa.

This arrangement of signal, although convenient for DTC-DCE connection, becomes awkward for a short DTE to DTE connection, where a modem is not required. A slight modification of the ordering in the lines allowed getting rid of the modem, yielding a null-modem serial connection. Exchanging TxD and RxD, and RTS/CTS allowed for bypassing the modem. Supporting the full handshake protocol included also exchanging DTR and DSR, as illustrated by the dashed lines in Fig. 9.13a. An even simpler connection was enabled by completely bypassing the handshaking protocol, as shown in Fig. 9.13b. As the handshaking protocol became less prevalent, the most common configuration became that including only three wires: TxD, RxR, and GND.

This is the mostly used in embedded applications and it is still the format used in most serial asynchronous interfaces. In such cases, the microcontroller and peripheral both play the role of DTEs and no handshaking signal are provided at all. When a peripheral does require flow control lines (typically RTS/CTS), their functionality in the MCU side can be provided using GPIO lines under user software control.

When a connection is made through a compliant RS232 channel or port as was illustrated in Fig. 9.10 on p. 486, level shifters become mandatory to provide for voltage compatibility. One of the most widely used level shifters for RS232 channels was introduced by Maxim Integrated, the MAX232. This chip features internal

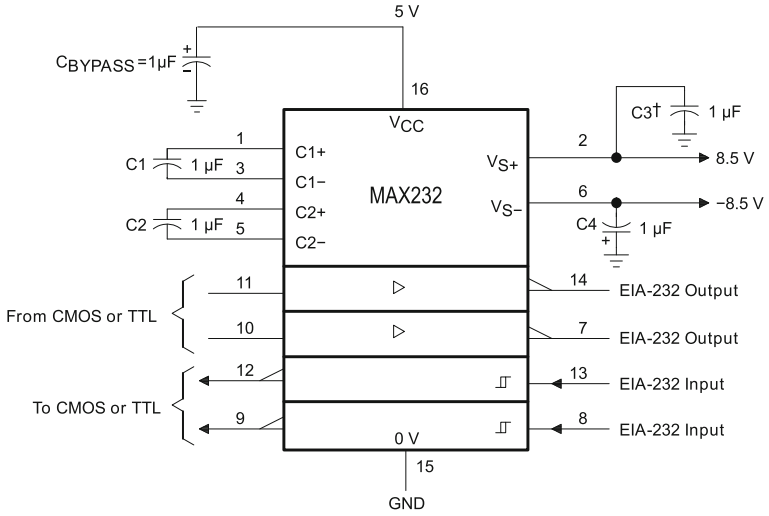


Fig. 9.14 Typical connection of a MAX232 driver/receiver (*Courtesy of Texas Instruments, Inc.*)

charge pumps that allow for producing RS232C-compatible voltage levels without the usage of a ±12 V bipolar power supply. The original chip used a single 5.0 V TTL-compatible supply, but later versions like the MAX13223E also offer 3.3 V and 3.0 V operation and low-power modes. Several manufacturers nowadays produce MAX232 compatible chips.

The MAX232 provides two pairs of buffers, a first pair supporting TTL to RS232 conversion, and the second pair supporting RS232 to TTL conversion. This arrangement allows for driving and receiving RS232C levels in a full-duplex channel with basic handshaking signals RTS/CTS within a single IC. External component requirement included only five capacitors of typically 1 μF. Figure 9.14 shows the recommended connection of the chip in a variant produced by Texas Instruments.

Since its introduction in 1962 by the EIA, RS232 has gone through multiple revisions (A through F), being the most prevalent revision C. RS232 dominated the serial interfaces in computers and peripherals for more than three decades, and it is still found in many computer applications. However, its inherent limitations that include support for only point-to-point connections, narrow bandwidth (up to 20 Kbps at 50 ft), and inconsistencies requiring deviation from the standard, gave way to newer and improved specifications. At present, new designs have migrated to standards like FireWire and the Universal Serial Bus (USB), being the later the predominant standard in contemporary serial interfaces. USB is a more robust form of serial communications that includes a full-fledged multi-drop protocol and allows for faster transfer rates than were attainable with the aging RS232 standard. Section 9.3.9 on p. 494 offers additional details on the USB standard.

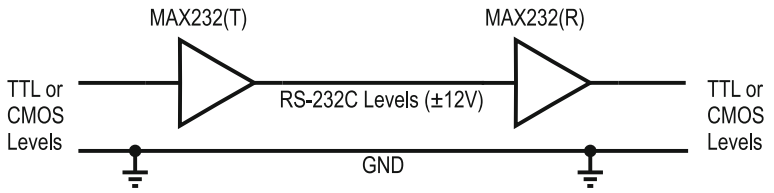


Fig. 9.15 Single-ended RS232C line

RS-422 and RS-485

Single-ended serial links like those used in RS232 channels, offer a cost effective way of carrying signals over wires, since they require only a single wire per signal plus a common ground. This topology is illustrated in Fig. 9.15. These connections however have a limited ability to fight noise. The main robustness factor they feature is their noise margin. RS232C for example uses up to ±15 V in the transmitter side, while the receiver needs only ±3 V to discriminate between marks and spaces (1's and 0's), yielding up to 12 V of noise margin. This large noise margin is however ineffective for rejecting noise coupled into the line by the unavoidable voltage loss caused by the line impedance and electromagnetic interference (EMI) picked-up by wires when passing through magnetic fields.

A differential channel overcomes this weakness by providing two wires per signal and representing signals by the relative voltage of one wire with respect to the other. This signalization scheme allows for cancelling any form of noise coupled in common mode into the lines. Noise caused by EMI, balanced voltage drops, and ground bounce are naturally eliminated in a differential channel due to its high common mode rejection ratio.

These characteristics also allow for using narrower voltage swings, which combined with the improved noise immunity enables higher transmission rates over longer distances. A drawback of differential channels is their cost, space, and routing complexity due to the requirement of using two wires per signal. Despite these factors, differential channels are the top choice for high-speed serial communications channels.

Two widely accepted physical standards based on differential connections are RS-422 and RS-485.

RS-422: This is a differential data transmission standard that offers robust mechanisms for communications over noisy environment. Officially designated as ANSI TIA/EIA-422, RS-422 uses a shielded, differential twisted pair with 100 Ω characteristic impedance and 16 pF/ft distributed capacitance. The signal swing can be between ±2 V and ±5 V with respect to ground and up to ±10 V from line to line, with a maximum short-circuit current output in its line drivers of 150 mA. Despite this seemingly narrow swing, when compared to RS232, RS422 can achieve data rates up to 10 Mbps at 50 ft or less, and up to 100 Kbps at a distance of 4,000 ft, a

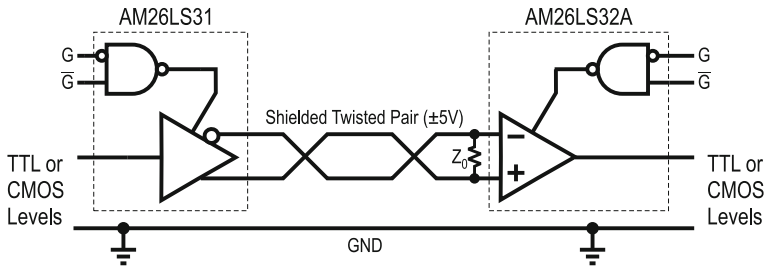


Fig. 9.16 Differential RS-422 line with AM26LS31/AM26LS32A driver/ receiver pair

whopping improvement with respect to that of RS-232. Figure 9.16 shows a typical RS-422 differential line.

RS-422 is designated as a *simplex multidrop* standard implying that a single RS-422 line can have only one driver but up to ten receivers without the need of repeaters or buffers. Its interconnections are ended by a termination impedance Z_0 that matches the characteristics impedance of the line in about $\pm 20\,\%$ to suppress far-end reflections in the line. Traditional drivers and receivers for RS-422 include the driver/receiver pair AM26LS31/AM26LS32A by Texas Instruments. This matched pair allows for enabling/disabling all chip buffers with a single control signal (G/$\overline{\text{G}}$), as illustrated above in Fig. 9.16, providing a more flexible control of the channel. RS-422 buffers can also be used as custom replacements for RS-232 line drivers, allowing to extend speed and distance in the channel. Such a change, although effective, requires additional logic to connect with standard RS-232 ports.

RS-485: This standard offers improved characteristics over RS-422 by specifying bidirectional drivers that allow for operating a single differential serial line as a half-duplex bus. This implies that the channel can be written by more than one driver, although only one bus driver can be active at a time. The standard, officially designated TIA/EIA-485, also calls for lines with lower characteristic impedance (54 Ω) and receivers with higher input impedance (120 KΩ) than RS-422, increasing the driver's fan-out. This change allows for accommodating 32 drivers and 32 receivers in a single bus topology and extending to longer line lengths. The remaining electrical and timing characteristics are similar to those of the RS-422 standard.

Figure 9.17 shows an RS-485 bus topology featuring three bidirectional, half-duplex nodes connected via RS-485 compatible transceivers. The shown transceiver is an SN65HVD10 driver/receiver by Texas Instruments. This transceiver features one driver and one receiver per IC, the latter with 1/8 unit load, extending the maximum number of nodes in the channel to 256. Independent, complementary driver and receiver enable lines $\overline{\text{RE}}$ and DE allow for individual or complementary direction control.

Since RS-485 lines are bidirectional, termination impedances (Z_0) are required at all ends of the line to suppress reflections.

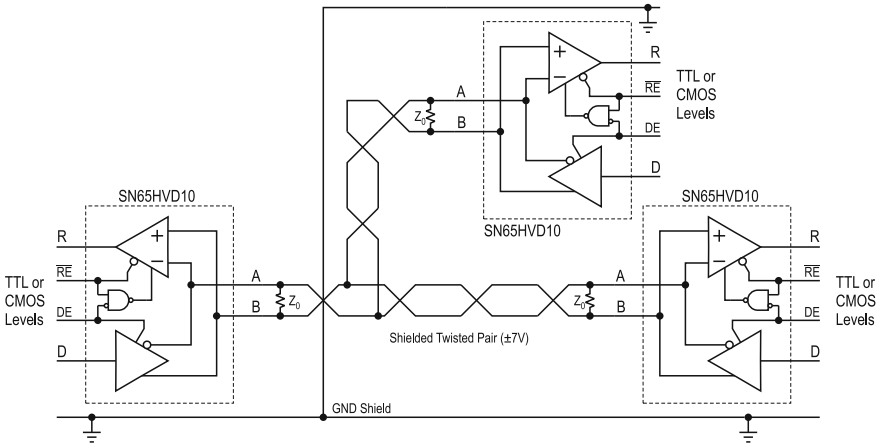


Fig. 9.17 Differential RS-485 line denoting its multidrop capability

Electrical RS-485 signals are compatible with those of RS-422, allowing the former to be used on RS-422 channels. Although RS-422 and RS-485 offer significant improvements over RS-232, they never reached the widespread usage level of RS-232.

9.3.9 The Universal Serial Bus (USB)

The Universal Seral Bus, or USB, was introduced in the mid nineties as a standard to provide a simple, yet consistent, fast, and robust way to communicate peripherals such as printers, scanners, keyboards, media players, digital cameras, and other similar devices to a host computer. USB was also designed to provide power to low-power peripherals, freeing each of them from requiring individual power supplies.

The acceptance gained by the protocol allowed it not only to displace power chargers and earlier computer communication standards like RS232, PS2, and Centronics. It also penetrated to the broad market of embedded applications providing an effective and reliable standard communicating and powering smart phones, industrial equipment, medical, military, and even space applications. USB has been the most successful computer standard in the history of computers.

In its initial 1996 conception, USB 1.0 supported both low- and full-speed data transfer rates of 1.5 Mbps and 12 Mbps, respectively, in half-duplex mode. USB 2.0, released in year 2000, increased the bandwidth with an additional high-speed data rate of up to 480 Mbps. The latest USB generation, USB 3.0 released in 2008, boosts transfer rates with a Super-Speed mode reaching up to 5 Gbps with the ability of supporting full-duplex communication. Every newer version is backward compatible, allowing to support earlier generation devices.

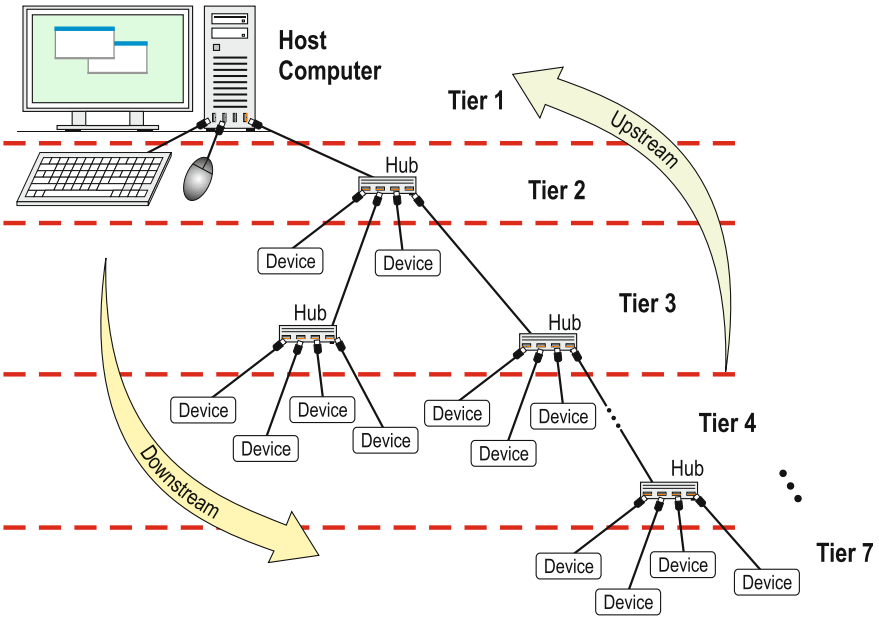


Fig. 9.18 Tiered structure of a Universal Serial Bus

USB Architecture

Unlike RS232 and RS485, USB is more than just a standard for cables, voltages, and connectors. The Universal Serial Bus is a complete communication protocol that includes physical, link, session, and application layers enabling host-supported functionality with a low-cost and consistent specification.

The architecture of a USB includes three types of elements: Host, devices, and cables. In its original version a USB topology can feature only a single host computer (master) and up to 127 devices in a tiered star configuration connected by USB cables, as illustrated in Fig. 9.18.

The USB host, although originally thought to be a personal computer, can be any kind of computing platform fitted with a USB host controller. The host controller is responsible for most of the functions that control the communication in the serial bus. It includes hardware and software layers that take care of the functionality supporting hot swapping operations, device enumeration, session and flow control, power management, and communicating with the host CPU application. All communications in the bus can only be initiated by the host, which acts as bus master.

USB devices, identified in the standard as "*functions*", are the particular peripherals connected to the bus. From the point of view of the host, devices behave like slaves. A USB device functionality will depend on the type of peripheral they support, be it a keyboard, a mouse, a digital camera, a flash drive or any other kind.

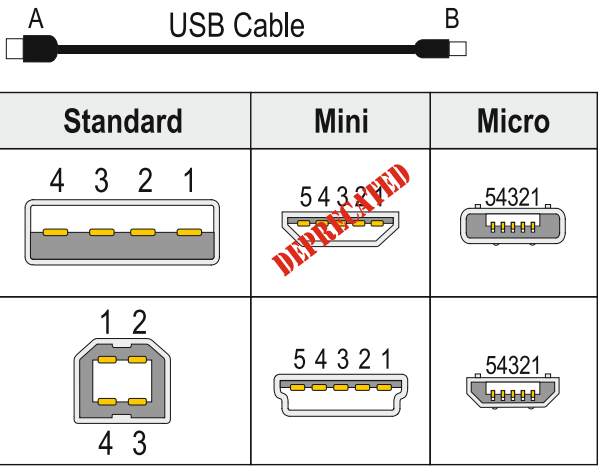


Fig. 9.19 Connectors in USB 2.0 cables

One special type of device is a USB hub: A device featuring multiple USB ports that allow for connecting additional devices or hubs to the bus. The host controller includes a hub by itself, creating the first tier in the bus. Each subsequent external hub located downstream in the bus establishes a new tier supporting multiple USB ports. A hub typically provides four ports, although a larger number is possible. A single bus can have a maximum of 127 devices in at most, seven tiers.

Electrical and Physical Characteristics

USB cables provide the physical link connecting devices in each tier to either the host or to the corresponding hub. Each USB cable can be up to 5 m (12.7 ft) long, implying the entire bus can extend to a theoretical length of 35 m (90 ft).

The form of a USB cable depends on the standard. A USB 2.0 or lower carries four shielded wires: two for power (V_{DD} (+5 V) and GND) and two forming a differential twisted pair for carrying data(D+ and D−). The cable has two different connectors: connector "A" for the upstream side and connector "B" for the downstream side. The shape and size of the connector includes three sizes: standard, mini, and micro. Figure 9.19 shows the forms of connectors "A" and "B" for the different connector sizes. Table 9.1 lists the signal assignment for each connector type.

The most commonly used USB cables feature a standard connector A for the upstream and a Mini or Micro B connector for the downstream. This arrangement provides for a standard size for the PC/hub side and a small footprint on the device side, highly convenient for miniature devices like smart phones and cameras. This usage pattern has become so prevalent that the Mini-A (also known as Mini-AB) connectors have been removed from the standard.

Table 9.1 Signal assignment in USB 2.0 connectors

Pin No.	Standard	Mini	Micro
1	$V_{Bus} = 5\,\text{V}$	$V_{Bus} = 5\,\text{V}$	$V_{Bus} = 5\,\text{V}$
2	D−	D−	D−
3	D+	D+	D+
4	GND	NC	NC
5	N/A	GND	GND

USB signals are encoded using NRZI (non return to zero inverted) with a unipolar differential signaling driver. A logic one is transmitted by pulling up D+ $\geq 2.8\,\text{V}$ via a $15\,\text{K}\Omega$ resistor, and pulling down D− $\leq 0.3\,\text{V}$ via a $1.5\,\text{K}\Omega$ resistor. A logic zero is obtained by swapping the voltages on D+ and D− with their respective pulling resistors. Connections to drivers and receivers do not require termination resistors.

USB 3.0 includes a second differential link in its cable and connector, allowing for full-duplex communication. The connectors for USB 3.0 are downward compatible with earlier versions, implying that they accept USB 1.1 and 2.0 devices, although a USB 3.0 function will not necessarily work in a USB 2.0 port.

The power bus (V_{Bus}–GND) allows powering low-power devices directly from the USB port. It provides a nominal voltage of $5\,\text{V} \pm 5\,\%$. The current limit per unit load in ports up to USB 2.0 is 100 mA, with a maximum total of five unit loads (500 mA). In USB 3.0 the unit load current is 150 mA and allows a maximum of six unit loads (900 mA).

When a function is plugged into a USB port, it is first recognized as a low-power device and granted a current limit of one unit load (100 mA in USB 2.0). After an enumeration transaction, the device can ask to be granted a high-power status, which if granted, allows it to draw the maximum allowed current by the version (500 mA in USB 2.0 or 900 mA in USB 3.0). Current consumption by functions (devices) in the bus is monitored and if any of them exceeds the granted limit, the infracting function will be disabled.

USB Interfacing and Programming

Interfacing and programming for USB devices can be seen from two different perspectives: from the perspective of a function developer creating from scratch interfaces that comply with the USB standard, or from the perspective of the application developer that seeks to use COTS parts to enable embedded applications that communicate over an existing USB infrastructure.

For those in the first group, that might need to possibly develop silicon for USB hardware and software compliance, the best resource would be the complete USB specification published by the USB Implementer's Forum [71, 72]. These documents provide all the details necessary for developing from scratch applications that meet the physical, electrical, and functional specifications of the USB standard.

For those in the second group, seeking to develop applications with COTS parts, capable of communicating over an existing USB infrastructure without having to implement the protocol from scratch, there are a number of resources that allow for developing embedded applications able to comply with USB function requirements and communicate with a host. Most of these solutions are based on ICs that implement the USB protocol and provide simple interfaces that can be directly handled by an MCU without the burden of having to develop an entire USB compliant function protocol.

The pool of available options includes ICs able of converting UART, I^2C, or SPI channels to USB, enabling easy integration of custom developed applications as USB functions. Specific examples include FTDI Chip's FT201X, FT220X, and FT231X, able to interface USB to I^2C, SPI, and UART, respectively. Many of these ICs are available conveniently packaged within cables or drop-in modules that can be directly driven from MCU ports, providing transparent translation between formats.

9.4 Synchronous Serial Communication

Synchronous serial channels are characterized by having both, transmitting and receiving devices synchronized with the same clock signal. This is achieved by having both, data and clock transmitted over the channel. Figure 9.20 shows a synchronous full-duplex channel configuration illustrating the concept in a point-to-point connection.

Synchronous serial channels usually operate in a master/slave mode where the master device initiates transfers and provides the clock signal driving the timing and synchronization in the channel. A slave device is controlled by the master to either receive or send information when instructed to do so.

Since both master and slave devices are driven with the same clock signal, datagrams in synchronous channels can be longer than those used asynchronous channels. Also, by using suitable line drivers, many synchronous channels allow multidrop configurations enabling the establishment of a network of devices communicating over the same channel.

Several synchronous protocols have been developed, each one establishing its own set of rules of how data and clock signals are routed through the channel and

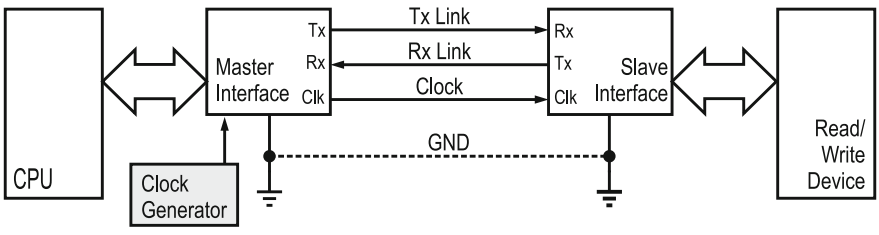


Fig. 9.20 A synchronous, full-duplex serial channel denoting the transmitted clock

how interconnecting devices behave. Some protocols establish a single master device per channel with the rest behaving as dedicated slaves, while others allow devices to behave as either master or slave. In the latter case, specific protocol rules are introduced to ensure that only one device in the channel behaves as master at a given time.

Examples of synchronous serial protocols include the Serial Peripheral Interface (SPI), the Inter-Integrated Circuit (IIC or I^2C), and the Controller Area Network (CAN) among others. Several of these protocols are explained in the sections below.

9.4.1 The Serial Peripheral Interface Bus: SPI

The Serial Peripheral Interface (SPI) is a synchronous serial bus standard with full-duplex capability introduced by Motorola to support communications between a master host processor and one or multiple slave peripheral devices.

SPI is one of the simplest synchronous communications protocols ever developed, as it only establishes a basic mechanism to relay packets between a dedicated master and one or more slaves, without specifying any data, session, or higher-level protocol. This simplicity translates into a protocol with little overhead, capable of achieving a high efficiency in the channel usage, particularly in point-to-point connections.

An SPI controller is developed around a single shift register that serves as both, receiver and transmitter, synchronized by the transmission clock. Figure 9.21 illustrates the structure of an SPI master-slave pair implementing a point-to point channel. The CPU side of the interface connects to the data lines D0-D7 and the selection and control lines like any other interface, omitted in the figure for simplicity,

The channel side features four signals whose functions are:

- SCLK: Serial clock, sent by the master and synchronizing both master and slave.
- SDO: Serial data-out, the serial output stream from the device.
- SDI: Serial data-in, the serial input stream into the device.

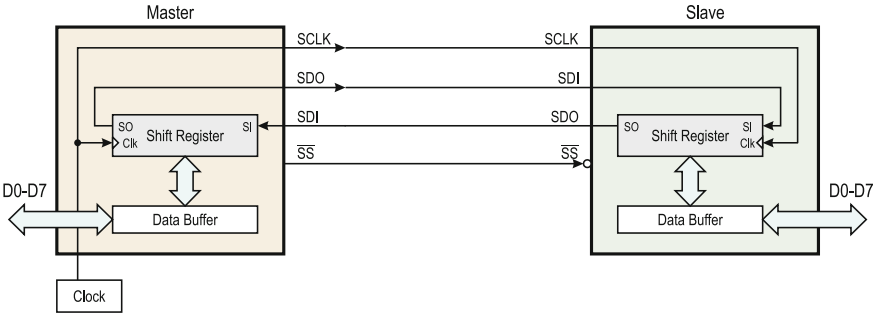


Fig. 9.21 SPI synchronous bus for a point-to-point connection

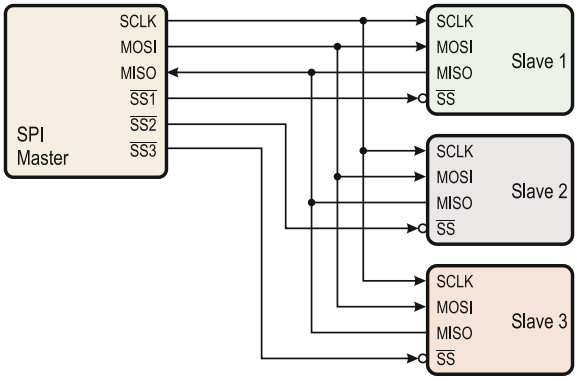


Fig. 9.22 SPI single-master, multi-slave connection

- SS: Slave select, a selection line to enable the slave. The SS line is omitted in point-to-point interconnects, grounding the slave SS input.

A character sent by the CPU is stored in the interface data buffer and copied into the shift register. The master initiates the transfer by activating the slave select signal. In a point-to point, the slave select signal can be hardwired, allowing a 3-wire connection with the master.

Since the master and slave shift registers are tied together and synchronized by the same clock, both interfaces simultaneously transmit and receive. As bits are shifted out from the master they are also shifted into the slave and viceversa. A device doing a transmit-only transfer simply discards the received frame.

Since an SPI does not specify an address field, additional hardware is required for managing multiple slaves. The slave selection lines can be implemented with GPIO lines or an external decoder and user software must activate the corresponding slave. Figure 9.22 illustrates a single master, multi-slave SPI configuration.

The SPI clock signal can be derived from the system clock, from an external clock source, or from a timer channel, depending on the particular implementation. Interrupt-based servicing is possible in most MCUs. The specific details will depend on the manufacturer's design.

Like other serial formats, SPI transfers using the logic levels of the interface is limited to only a few inches. By adding appropriate drivers and receivers, SPI channels could be extended over longer distances, although it is mostly used for short intra-board distances. SPI has no error checking mechanism and is not defined as an standard.

Due to its simplicity, SPI has been adopted by numerous manufacturers of serial EEPROMs, real-time clock modules, data converters, LCDs, and other peripherals. Due to the absence of upper-level protocols and no standardization, each implementation can define its own particularities, and upper-level protocols are left to the application implementer.

9.4.2 The Inter-Integrated Circuit Bus: I^2C

The Inter-Integrated Circuit bus, or I^2C is a synchronous serial protocol developed by Philips Semiconductor (now NXP Semiconductors) in the early 1980s to support board-level interconnection of IC modules and peripherals. The protocol uses two lines, SDA (Serial Data) and SCL (Serial Clock), (and ground) to establish a half-duplex, master/slave, multidrop bus capable of handling multiple masters and slaves. The serial clock line (SCL) synchronizes all bus transfers, while SDA carries the data being transferred.

I^2C was designed to be an intra-system serial bus capable of accommodating all kind of peripherals found in embedded systems. These include MCUs, data converters (ADCs and DACs), display devices, memories, real-time clock calendars, GPIO modules, etc. A large number of IC peripheral manufacturers offer products compatible with I^2C.

Devices in an I^2C bus are software addressable, with 7- or 10-bit address fields. Although the most common usage establishes a single master/multiple slave topology, the protocol allows for any device to be a master, as it incorporates collision detection and arbitration mechanisms necessary for a multi-master operation. Nominal maximum speeds can reach up to 5 Mbps, although in reality the limit is imposed by the total bus capacitance, with its maximum specified at 400 pF or 500 pF depending on the version. With input capacitances at 10 pF, plus that of cables, practical numbers call for a few dozen devices per bus. Bus extenders might be used to expand that number if the application requires doing so.

Being a bus for interconnecting ICs in a board, distances in I^2C are expected to be short, in the order of a few inches. It might be possible to stretch its length to several feet, at the expense of reduced speed due to the effect of the increased capacitance.

The bidirectional multidrop capability of the SCL and SDA lines is achieved by driving them with open-collector or open-drain drivers. This calls for fitting bus lines pull-up resistors to complete their driver circuit. Figure 9.23 shows an I^2C bus topology featuring an MCU as master and several other devices as slaves.

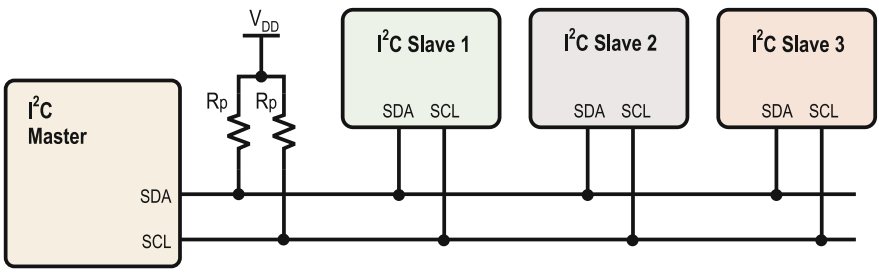


Fig. 9.23 Topology of an I^2C bus connection

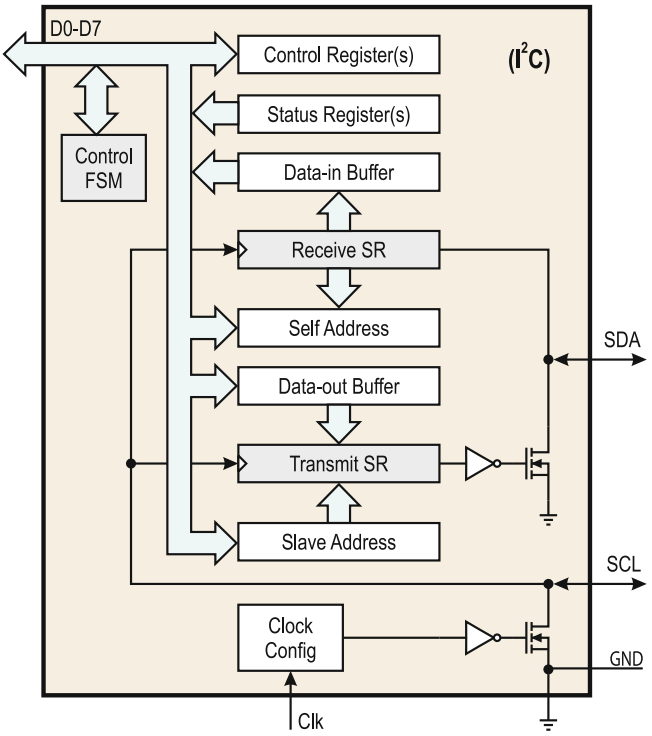


Fig. 9.24 Internal structure of an I^2C interface

I^2C Interface Structure

Each device connecting to an I^2C bus requires an interface able to produce the signalization specified by the protocol. An I^2C interface uses two shift registers, one for receiving, one for transmitting, joined by a transceiver that allows for bidirectional operation in the SDA line. Figure 9.24 illustrates the internal structure of an I^2C interface. Note how the SDA transceiver is implemented with a single NMOS and an inverter. A similar approach is used for SCL as this line must allow for receiving the clock signal if the device operates as a slave, or driving the clock signal when operated as master.

The transmitter section shift register can be loaded from either the slave address register with the address of the destination slave device, or from the data-out buffer with the character to be transmitted. The receiving shift register transfers incoming packets to either an address comparator to determine packet ownership, or to the data-in buffer for CPU access and when matched.

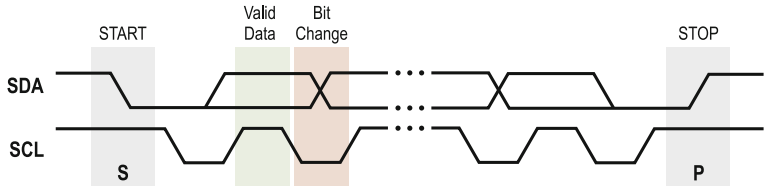


Fig. 9.25 Timing diagram of I^2C signals denoting the start, transfer, and stop conditions

I^2C Operation

The protocol in an I^2C specifies three different conditions in the channel: *Start*, *transfer*, and *Stop*. A start condition (S) occurs when a master takes control of the bus to initiate a transfer. Before taking any action, the master listens to the SDA line to determine if the channel is idle. An idle status is indicated when both SCL and SDA remain high for at least one clock period. Upon channel availability, the master lowers the SDA line to set the start condition, making the channel busy. At this point all devices in the bus will be placed in listen mode for the incoming data. A start condition can also be sent in the middle of an active message. This action is called a *Restart* condition and can be used to change the transfer direction within a message without the current master actually releasing the bus.

A data transfer occurs when data bits are being sent over the channel. I^2C supports eight-bit characters transfers. Character bits are transferred with the most significant bit (msb) first. A data bit is valid when the clock (SCL) is high, and the change from one bit to the next occurs when the clock is low. A transfer might contain one or multiple characters.

When a transfer is complete, the master issues a stop condition (P) by raising SDA while keeping SCL in high. Figure 9.25 illustrates the timing relations of SDA and SCL for all three conditions.

A message sent over an I^2C bus begins right after a start condition with an address field sent by the master to select the destination slave device. The address field is ended by a read/$\overline{\text{write}}$ bit indicating whether the transfer would read or write the addressed slave. After this last bit the master releases the SDA line.

All slaves in the bus will receive the address field and compare it to their own address. Only the slave whose address matches that sent by the master will send an acknowledgment. The selected slave acknowledges the reception of its address by pulsing the SDA line.

The master issues a pulse in the SCL line to sample the SDA line and detect if the ACK condition is present. If the acknowledgment is received, the transmitter takes control of the SDA line and proceeds sending the next data byte, msb first. Note that in case of a write transfer, the master would be the transmitter and thus it would control both SDA and SCL. In a read transfer the slave would be the transmitter and thus it will control SDA while SCL remains controlled by the master.

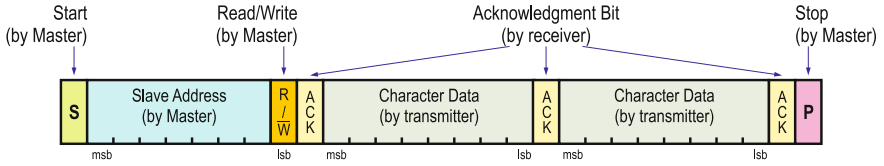


Fig. 9.26 Structure of an I^2C message

If after sending the address field the master does not detect the SDA line going low to denote an acknowledgment, this would lead to a "Not Acknowledgment" condition (NACK). User software shall determine wether aborting the transmission (by sending a stop) or retrying the transfer.

The message following a successful address transfer might contain one or several bytes, each followed by an acknowledgment field. If after a byte transfer the receiver were not yet ready for the next character it would hold low SCL, serving as an indication to the master to wait. Next the transmitter sends each character in its message, each followed by an acknowledgement by the receiver. Note that after receiving the address field, the roles of transmitter and receiver will depend on the read/write bit.

There is no preestablished limit in the number of 8-bit characters that can be accommodated in a message. When the transmitter completes its message, the master issues a stop condition, setting the channel idle. If necessary, the master might issue a repeated start condition at the end of a message, maintaining control of the bus. Figure 9.26 illustrates an I^2C message, denoting the order in which each field appears in the packet.

Besides the basic addressing and data transfer, I^2C also provides clock synchronization mechanisms, clock stretching, and arbitration for contending masters.

Clock Synchronization and Arbitration in the I^2C Bus

Clock synchronization and arbitration are two mechanisms used in multi-master bus configurations to decide which master will take control of the bus in the event that more than one master begin transmitting on an idle channel. Arbitration relies on the synchronized clock operation of masters.

Clock Synchronization: Clock synchronization allows for synchronizing the clock signals of two or more masters that might be simultaneously operating at different speeds in the same bus. Consider two I^2C master devices, namely *master 1* and *master 2*, attempting to simultaneously transfer information on the bus, as illustrated in Fig. 9.27. They both will attempt to drive the common clock line SCL. Let's call the master's clock line interfaces CLK1 and CLK2, respectively. Without loss of generality, lets assume CLK1 runs faster than CLK2.

When a high-to-low transition occurs in the SCL line, both devices will begin counting their low periods. As CLK1 runs faster than CLK2, *master 1* would disable

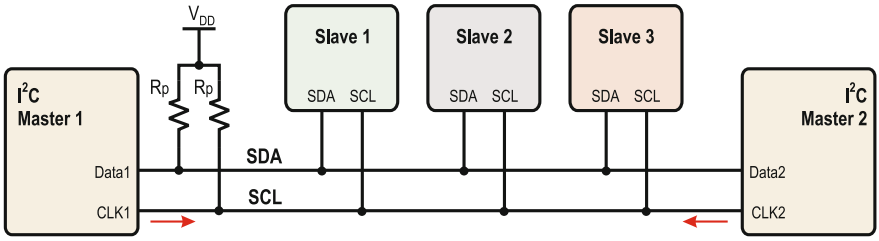
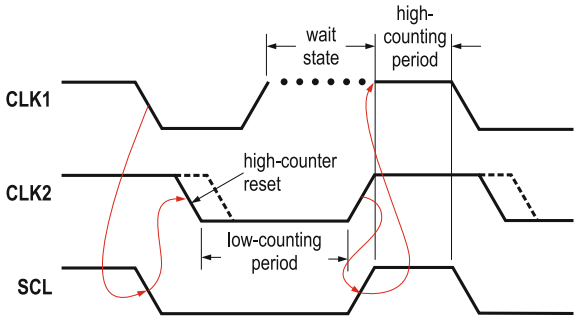


Fig. 9.27 Multimaster I^{2}C bus configuration

Fig. 9.28 Multimaster signal timing for a clock synchronization process



its clock driver first trying to bring the SCL line high. However, as CLK2 is slower, *master 2* would still keep the SCL line low, as illustrated in Fig. 9.28. Recall that SCL is an open-collector or open-drain line that operates as a wired-AND.

When *master 1* senses that despite having released its clock drive, SCL is still low, it will enter into a *wait state* until detecting that SCL has actually gone high. When *master 2* finally completes it low period counting it would release its clock driver allowing SCL to finally go high. At this point both devices start counting their high periods.

In the high period, again, *master 1* will end first its high-period count, driving SCL low. At this point the slower master (*master 2*) would just reset its high-period counter and drive SCL low to begin counting its low period. This mechanism allows for synchronizing the clocks of all masters: the master with the slowest clock determines the length of the low period, while the fastest master sets the length of the high period. This synchronization mechanism is essential to enable an efficient I^{2}C bus arbitration protocol.

I^{2}C Bus Arbitration: An arbitration process takes place when multiple masters attempt to simultaneously seize control of the bus. The arbitration process determines which master will remain in control. The protocol is very simple: the fist master to place a logic high level conflicting with a low-level data placed by another master losses control of the bus.

To understand this rule it only takes to remember that SDA works as a wired-AND line. Thus, when two drivers place conflicting data, the SDA line will assume a low

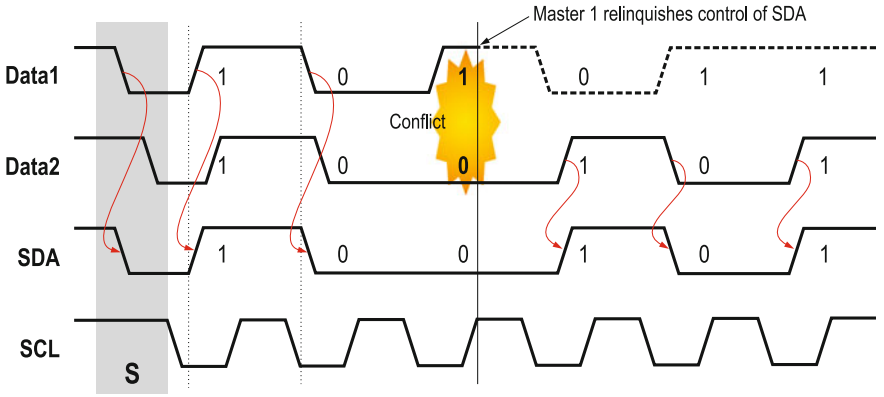


Fig. 9.29 Timing of a bus arbitration process

level. Every time a master drives the SDA line placing a data bit on it, the master will also listen to the line to determine if it actually took the issued logic level. A master who places a logic-one level on SDA and listens a zero identifies that there is a conflict. As a consequence, the conflicting master relinquishes control of SDA line and enters into listening mode, like a slave, in case the other master were trying to address it. Otherwise it keeps waiting for a stop sequence to make a new attempt to send its data. Figure 9.29 illustrates a signal sequence exemplifying this type of transaction.

In this example Data1 and Data2 represent the data drivers in Master1 and Master2, respectively. At the beginning of the timing sequence both masters detect an idle channel, so both of them issue start conditions and begin driving the SDA line. As long as the placed data does no conflict, both masters continue to drive the SDA line. Recall that their SCL lines are synchronized. In the first conflict, the master placing a logic-high level, Master1 in this example, will not see its data on SDA and thus proceeds to relinquish control of the bus.

It deserves to mention that these features are transparent to the application developer when the MCU at hand has an I^2C peripheral module. The only instance when the programmer needs to consider such protocols is when the task at hand involves implementing an I^2C interface in software. This situation could arise when interfacing an MCU without I^2C hardware support to the bus, a not so common situation due to widespread availability of MCUs supporting the protocol. In such a case, a good source for detailed information is the official I^2C-bus specification published by Philips [74].

9.5 Serial Interfaces in MSP430 MCUs

MSP430 devices provide several serial on-chip communication resources that include UARTs, SPI, I^2C, and USB. These are made available through different modules that include:

USARTs Universal Synchronous/Asynchronous Receiver/Transmitter modules, which provide basic UART and SPI connectivity.

SCI Serial Communication Interface modules that provide fundamental SPI and I^2C capabilities.

USCI Universal Serial Communication Interface, with wide support for UART, SPI, and I^2C modes. Series x5xx/x6xx feature enhanced USCIs as well.

USB Modules Selected x5xx/x6xx devices feature Universal Serial Bus modules.

Not all resources are available in all families or all devices of a family. The datasheets of each specific device provides a comprehensive list of the serial capabilities they provide. Note that depending on the particular device, several types of serial adapters might coexist in the same MCU, including multiple instances of the same type. The descriptions in this section focus on the capabilities of each type of serial communication resource available for MSP430 devices.

9.5.1 The MSP430 USART

The Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is one of the earliest serial peripheral interfaces supported by MSP430 devices. Featured in devices series x3xx, x4xx, x1xx, and x2xx, the USART is one of the most prevalent and classical serial interfaces in the MSP430.

The USART architecture in MSP430 devices has experimented little changes across generations. In all generations it supports both UART and SPI modes with the same peripheral module. This implies that a single on-chip module can be used as either one or the other. If both types of serial peripherals were required for a particular application, more than one module would be required. In addition to UART and SPI modes, the USART in generation x1xx devices also supported I^2C communication.

MSP430 series x5xx/x6xx devices discontinued supporting the classical USART, to stick only to the more recent Universal Serial Communication Interface (USCI) and its Enhanced version (eUSCI), discussed in Sect. 9.5.3.

The MSP430 USART Module in UART Mode

In its UART mode, the MSP430 USART module behaves like a classical asynchronous interface with a few enhancements. Like UART configurations discussed earlier, in Sect. 9.3.4, this module supports 7- or 8-bit characters, full-duplex transfers, and

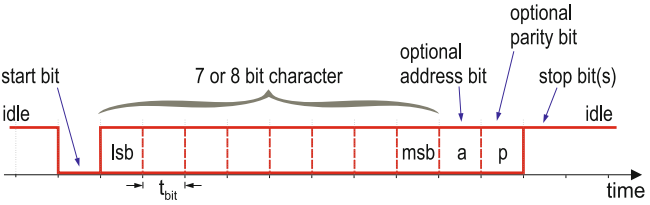


Fig. 9.30 Structure of asynchronous packets in MSP430 UART

interrupt triggering capability for receive and transmit events. Particular enhancements include support for idle-line and address-bit protocols, and the inclusion of a baud rate generator capable of accommodating fractional rates with reduced channel error. The receiver and transmitter sections share the same baud rate generator module.

UART Configuration and Operation: The UART mode is selected by clearing the SYNC bit in the USART Control Register (UxCTL). This register also allows for configuring most UART parameters such as: parity enable/disable and type, number of stop bits, character length, and multiprocessor mode.

The USART is reset by setting the SWRST bit in the UxCTL. The SWRST bit is automatically set by a PUC, so by default the USART is disabled. The USART initialization and enabling must be completed while holding the module in its reset mode. If interrupt operation were desired, the corresponding interrupt enable bits must also be set, recalling that transmitter and receiver have independent enables.

The character format follows the general organization of an asynchronous packet, except that it might include an optional address bit after the most significant data bit. This address bit is inserted when the UART is configured in the multiprocessor mode, allowing for specifying wether the transmitted packet is an address or data field. Figure 9.30 shows the modified structure of an asynchronous packet.

Multiprocessor Mode (MM): When used in conventional point-to-point connections, the UART is configured in the idle-line format. This is the traditional form of an asynchronous point-to-point connection where each character in a message is individually framed and separated from the next by an idle channel period. The bit time is determined by the chosen baud rate.

The MSP430 UART also supports having three or more devices on the bus through either the idle-line or address-bit multiprocessor formats.

In the idle-line multiprocessor mode (MM = 0), characters within a message are also individually framed, but the idle time between characters is used to indicate the separation between characters and messages. A short idle time (less than ten-bit time) separates characters within a message and an idle-line interval of ten-bit times or longer separates messages. No address bit is inserted in this mode. Instead, the first character received after a valid idle-line interval is accepted as an address field.

When an address field is received, it is transferred to the data-in buffer and wakes-up the receiver. If interrupts were enabled, a receiver interrupt would be triggered.

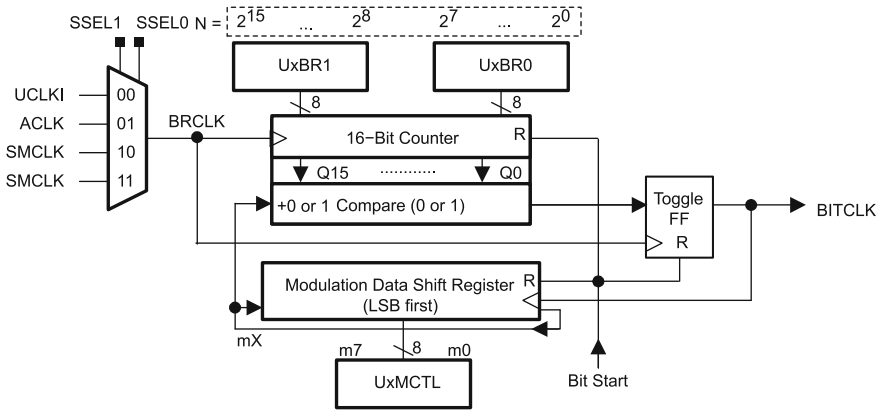


Fig. 9.31 Structure of the UART baud rate generator in MSP430 MCUs (*Courtesy of Texas Instruments, Inc.*)

Characters received after an address field do not generate interrupts nor get transferred to the input buffer. User software must take care of checking the received address field and compare it to any internally configured address, and if a match were detected, clear the receiver wake-up flag. This will cause the incoming characters to be transferred to the input buffer, making them available to the CPU.

Transmission in this mode only requires setting the transmitter awake control bit and place any character into the transmission buffer to send an idle frame. Afterwards, user software must just place the address information onto the transmission buffer, followed by the corresponding data characters.

In address-bit multi-processor mode (MM $=$ 1) an address bit is inserted to each packet to denote wether the transmitted character is an address (a $=$ 1) or data (a $=$ 0) field, and the inter-character idle time has no meaning. If parity were enabled, the address bit wold be part of the parity computation. Character reception and transmission would happen similar to the cases described above for idle-time mode.

Error Detection: The UART can detect framing, parity, overrun, and break errors in the channel, and set the appropriate bits in the receiver control register (UxRCTL). A break error indicates the reception of ten or more low bits after missing a stop bit. Optionally, a receiver interrupt can be triggered by any of the detected errors. Recall that receiver interrupts are multi-sourced and therefore the ISR must check the UxRCTL register to determine the cause of the trigger.

UART Baud Rate Generation: The MSP430 UART has a baud rate generator capable of producing standard baud rates from non-standard clock frequencies. By combining a prescaler and a modulator, the generator is capable of supporting fractional divisors. This results in target baud rates with reduced error. Baud rates up to one third the source clock (BRCLK) are possible. Figure 9.31 shows the structure of the baud rate generation module.

To produce a desired baud rate, the source clock (BRCLK) is divided by a factor $N = \text{BRCLK}/\text{baud rate}$. When a non-standard frequency is used, N usually results in a non-integer number. To reduce the error that would otherwise be introduced if the fractional part of the quotient were discarded, the prescaler in the baud rate generator is loaded with the integer part of N, while the modulator is configured to perform an approximation of the bit time corresponding to the fractional part.

The fractional bit time approximation is achieved by choosing the length of each individual bit ($t_{bit(j)}$) to the nearest integer BRCLK count that produces the smallest cumulative packet time error. This computation is made by the programmer in an iterative procedure while determining the commands to configure the UART.

Equation (9.2) provides an expression for estimating the cumulative packet time error in the transmitter side up to bit j ($\xi^t_{\text{pkt}}(j)$) obtained by setting or clearing each individual modulator bit m_j. The iterative evaluation of this formula tells wether bit m_j needs to be set or clear, as we want to choose the condition yielding the minimum error. Input parameters include the desired baud rate (baud rate), the frequency fed to the baud rate generator (BRCLK), and the integer part of N (UxBR). The modulation factor UxMCTL is built by taking the bit values that produce the minimum error at each position j.

$$\xi^t_{\text{pkt}}(j) = \frac{\text{baud rate}}{\text{BRCLK}} \left[\text{UxBR} + \frac{1}{(j+1)} \sum_{i=0}^{j} m_i \right] - 1 \tag{9.2}$$

The error of each individual bit time would be a measure of how much $t_{bit(j)}$ deviates from the nominal $t_{bit} = 1/\text{baud rate}$. Equation (9.3) provides an expression to quantify such an error.

$$\xi^t_{\text{bit}}(j) = \frac{\text{baud rate}}{\text{BRCLK}} \left(\text{UxBR} + m_j\right) - 1 \tag{9.3}$$

Example 9.2 illustrates the process of computing the packet and bit errors an deciding the value of m_j for $j = 0, 1, \ldots, 7$. Note that bit values for $j > 7$ simply reuse the already devised values of m_j starting at m_0.

Example 9.2 *Consider USART0 in UART mode in an MSP430F169, driven by a 32.768 KHz clock in ACLK. Determine the values of registers U0BR and U0MCTL to obtain a baud rate of 4,800 bps.*

Solution: *As the quotient $N = BRCLR/(\text{baud rate})$ is non-integer ($N = 6.83$), we set U0BR = 6 and determine the bits in U0MCTL to approximate the 0.83 fractional part. Assuming the channel is configured for 8-bit characters, with parity even and one stop bit, each packet will consist of eleven bits. Table 9.2 shows the modulator bit values obtained by taking the bits that minimize the error computed with (9.2).*

The third column indicates the exact expected value for bit transitions while columns $\xi^t_{pkt}(j)|_{m_j=0}$ and $\xi^t_{pkt}(j)|_{m_j=1}$ denote the errors when using N or $N+1$ BRCLK cycles for the bit time. The absolute minimum of these two columns determines the value chosen for m_j. After bit eight, the values repeat from bit zero. The

Table 9.2 Timing values and relative errors in computing the modulation bits in the transmitter baud rate generator

j	BIT	$t_{\text{bit}}(j)$	$t_{\text{bit}}(n)$	$\xi^t_{\text{pkt}}(j)\Big\vert_{m_j=0}$ (%)	$t_{\text{bit}}(n+1)$	$\xi^t_{\text{pkt}}(j)\Big\vert_{m_j=1}$ (%)	m_j
0	Start	208.3E-6	183.1E-6	−12.11	213.6E-6	2.54	1
1	D0	416.7E-6	396.7E-6	−4.79	427.2E-6	2.54	1
2	D1	625.0E-6	610.4E-6	−2.34	640.9E-6	2.54	0
3	D2	833.3E-6	793.5E-6	−4.79	824.0E-6	−1.12	1
4	D3	1041.7E-6	1007.1E-6	−3.32	1037.6E-6	−0.39	1
5	D4	1250.0E-6	1220.7E-6	−2.34	1251.2E-6	0.10	1
6	D5	1458.3E-6	1434.3E-6	−1.65	1464.8E-6	0.45	1
7	D6	1666.7E-6	1647.9E-6	−1.12	1678.5E-6	0.71	1
8	D7	1875.0E-6	1861.6E-6	−0.72	1892.1E-6	0.91	1
9	Parity	2083.3E-6	2075.2E-6	−0.39	2105.7E-6	1.07	1
10	Stop	2291.7E-6	2288.8E-6	−0.12	2319.3E-6	1.21	0

per bit error (not listed in the table) has only two possibilities, 2.54 % when $m_j = 0$ and −12.11 % when $m_j = 1$. The resulting register values are: U0BR = 06h and U0MCTL = FBh.

The receiver side is also subject to errors. To identify the error sources let's review how the receiver works: the falling edge of a start bit wakes-up the UART receiver. The start bit is sampled at the middle of the bit time and thereafter the RxD line is sampled every t_{bit}. Note that both actions introduce error factors. The first is due to the mismatch between the occurrence of the start bit falling edge and the moment the UART actually accepts it. The second error factor is due the bit-to-bit timing variations introduced by the transmitter.

As in the case of the transmitter, a formula can be derived to estimate the cumulative bit error in a packet up to bit j, $\xi^r_{\text{pkt}}(j)$. The MSP430 User's Manual provides Eq. (9.4) [32].[4]

$$\xi^r_{\text{pkt}}(j) = \frac{\text{baud rate}}{\text{BRCLK}} \left(2 \left[m_0 + \left\lfloor \frac{\text{UxBR}}{2} \right\rfloor \right] + \left[j \cdot \text{UxBR} + \sum_{i=1}^{j} m_j \right] \right) - (j+1) \tag{9.4}$$

Iterative evaluation of (9.4), with both $m_j = 0$ and $m_j = 1$ allows identifying the values of m_j yielding the smallest error. As in the case of the transmitter, these are the bits making up UxMCTL.

Example 9.3 *For the case analyzed in Example 9.2, determine the value of U0MCTL in the receiver baud rate generator.*
Solution: *Iterative evaluation of Eq. (9.4) for $m_j = 0$ and $m_j = 1$ yields the values listed in Table 9.3, resulting in a value of U0MCTL = FEh.*

[4] This equation has not been verified.

Table 9.3 Computing modulation bits in the receiver baud rate generator

j	BIT	$\xi^r_{\text{pkt}}(j)\Big\vert_{m_j=0}$ (%)	$\xi^r_{\text{pkt}}(j)\Big\vert_{m_j=1}$ (%)	m_j	$\left\vert\xi^r_{\text{pkt}}(j)\right\vert$ (%)
0	start	−12.11	17.19	0	12.11
1	D0	−24.22	−9.57	1	9.57
2	D1	−21.68	−7.03	1	7.03
3	D2	−19.14	−4.49	1	4.49
4	D3	−16.60	−1.95	1	1.95
5	D4	−14.06	0.59	1	0.59
6	D5	−11.52	3.13	1	3.13
7	D6	−8.98	5.66	1	5.66
8	D7	−6.45	8.20	0	6.45
9	parity	−18.55	−3.91	1	3.91
10	stop	−16.02	−1.37	1	1.37

Interrupt-based and Low-power Mode Operation: The MSP430 USART has independent interrupt vectors for the transmit and receive sides. Whenever the transmit buffer can accept new characters for transmission, the UTXIFx is set. The interrupt will be triggered if both GIE and UTXIEx are set.

In the receiver side, whenever a new character is loaded into the receiver buffer (UxRXBUF), the URXIFGx is set. Its interrupt would be triggered if both GIE and URXIEx are enabled.

In addition to the receive and transmit events, error conditions can generate interrupts in the receiver side. Any detected errors: framing (FE), parity (PE), overrun (OE), or break (BRK) will set their corresponding error flags and also will set the RXERR flag. The URXIFGx would be set only if URXEIE is set, as this last setting allows the erroneous data to be transferred into the receive buffer. User software needs to check for any possible errors whenever the receiver interrupt is triggered. In the case of errors, error flags will remain set until UxRXBUF is read or the user's software clears them.

When operated in low-power mode with the DCO off, it is recommended to activate the Receive-Start Edge Detection mode. This allows turning-on the DCO to enable character reception out from the low-power mode. Enabling URXSE, URXIEx, and GIE would cause a start bit edge to trigger a receiver interrupt with URXIFGx clear. User software in the ISR must then cancel the low-power mode or turn on the chosen baud rate clock source to actually enable character reception.

The example below illustrates using USART0 in an MSP430F149 to implement a terminal echo. The code is written for TI CCE IDE.

Example 9.4 (UART Echo at 115,200 bps on 8.0 MHz Xtal) *Write a program to implement a terminal echo using an MSP430F149 running on an external 8 MHz clock. Use a low-power mode to save energy when no transfer is taking place. The port configuration is 8-bit per character, no parity, one stop bit (8N1).*

Solution: *This solution assumes the MCU has an 8.0 MHz HF crystal connected to XIN/XOUT pins. Pins P3.4 is used as TxD and P3.5 as RxD. A quick way to setup this system is on an 64-pin target board, installing the crystal, FET tool, and using an FTDI Chip TTL-232RG cable that allows for connecting the MSP430 UART pins to a USB port in the PC. A Hyperterminal (or similar) program can provide a quick interface on the PC. Below a brief code written by M. Buccini and G. Morton from Texas Instruments providing the MSP430 side of the solution [75].*[5]

```
;-------------------------------------------------------------------------
;MSP-FET430P140 Demo - USART0, UART 115200 Echo ISR, HF XTAL ACLK
;By M. Buccini and G. Morton - 05/2005
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
            .cdecls C,LIST,  "msp430x14x.h"
;-------------------------------------------------------------------------
            .text                           ; Program Start
;-------------------------------------------------------------------------
RESET       mov.w   #0A00h,SP               ; Initialize stack pointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupP3     bis.b   #030h,&P3SEL            ; P3.4,5 = USART0 TXD/RXD
SetupBC     bis.b   #XTS,&BCSCTL1           ; LFXT1 = HF XTAL
SetupOsc    bic.b   #OFIFG,&IFG1            ; Clear OSC fault flag
            mov.w   #0FFh,R15               ; R15 = Delay
SetupOsc1   dec.w   R15                     ; Addnl. delay to ensure start
            jnz     SetupOsc1               ;
            bit.b   #OFIFG,&IFG1            ; OSC fault flag set?
            jnz     SetupOsc                ; OSC Fault, clear flag again
            bis.b   #SELM_3,&BCSCTL2        ; MCLK = LFXT1
SetupUART0  bis.b   #UTXE0+URXE0,&ME1       ; Enable USART0 TXD/RXD
            bis.b   #CHAR,&UCTL0            ; 8-bit characters
            mov.b   #SSEL0,&UTCTL0          ; UCLK = ACLK
            mov.b   #045h,&UBR00            ; 8 MHz 115200
            mov.b   #000h,&UBR10            ; 8 MHz 115200
            mov.b   #000h,&UMCTL0           ; 8 MHz no modulation 115200
            bic.b   #SWRST,&UCTL0           ; **Initialize USART FSM **
                                            ;
Mainloop    bis.b   #CPUOFF+GIE,SR          ; Enter LPM0, intrpts enabled
            nop                             ; Needed only for debugger
                                            ;
;-------------------------------------------------------------------------
USART0RX_ISR;  Confirm TX buffer is ready, then Echo back RXed character
;-------------------------------------------------------------------------
TX1         bit.b   #UTXIFG0,&IFG1          ; USART0 TX buffer ready?
            jz      TX1                     ; Jump is TX buffer not ready
            mov.b   &RXBUF0,&TXBUF0         ; TX -> RXed character
            reti                            ;
                                            ;
;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"                ;
            .short  RESET                   ; POR, ext. Reset, Watchdog
            .sect   ".int09"                ;
            .short  USART0RX_ISR            ; USART0 receive
            .end
;-------------------------------------------------------------------------
```

[5] See Appendix E.1 for terms of use.

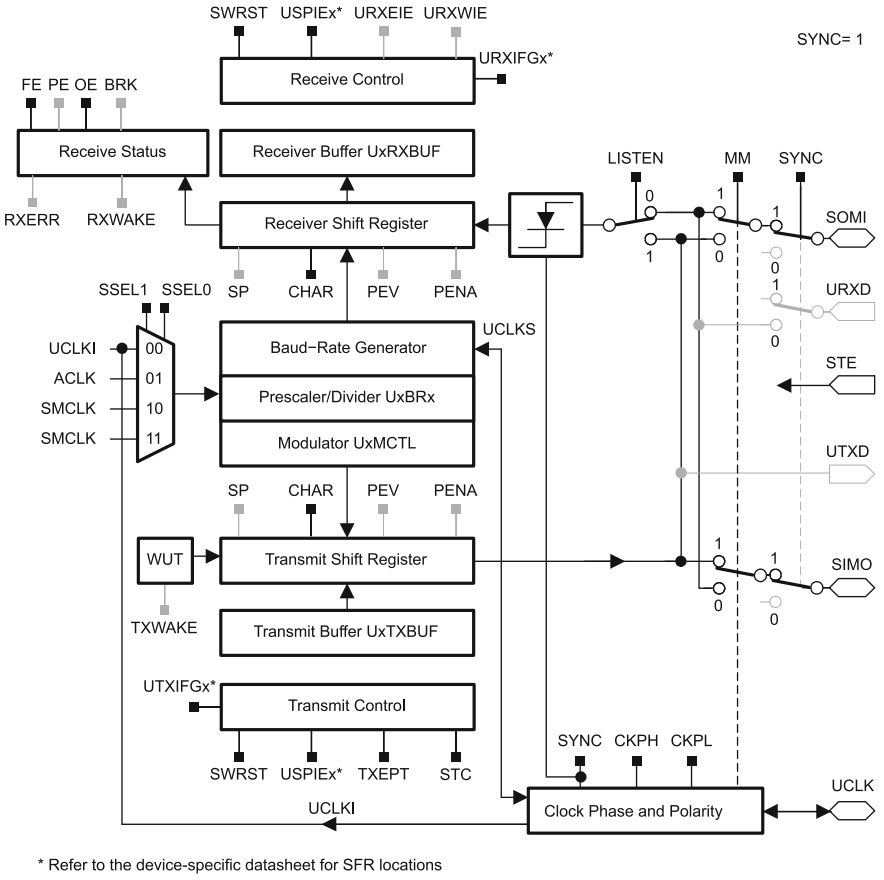


Fig. 9.32 Block diagram of MSP430's USART in SPI mode (*Courtesy of Texas Instruments, Inc.*)

The MSP430 USART Module in SPI Mode

The synchronous serial operation of the USART supports SPI mode transfers in three- or four-pin modes. The SPI mode features 7- or 8-bit characters, master or slave operation, 3- or 4-pin operation, separate transmit and receive shift registers, and configurable clock polarity.

The Interface lines are designated SIMO: Slave in, master out (same as SDO); SOMI: Slave out, master in (same as SDI); UCLK: Clock line (same as SCL); and STE: Slave transmit enable to allow for multi-master operation. Figure 9.32 shows the structure of the USART configured in the SPI mode.

Upon a reset, the SWRST bit is set, configuring the entire USART in reset mode. The SPI interface is enabled via the USPIEx bit in the Module Enable (MEx) register. MEx is configured with the USART in reset mode. Its programming is recommended to be performed after all other USART registers have been configured. The SPI

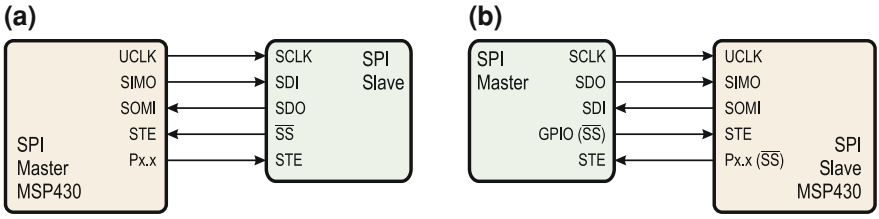


Fig. 9.33 Connecting the MSP430 SPI in four-pin mode. **a** MSP430 as master. **b** MSP430 as slave

becomes usable after the USART is taken out from the reset mode, by clearing the SWRST flag.

An SPI transfer is initiated in the master when data is moved into the transmit data buffer (UxTXBUF). When the transmission shift register becomes empty, data loaded in UxTXBUF is transferred into the transmission shift register, flag UTXIFGx is set indicating the receiver can accept new data, and the actual transfer of the character in the shift register is initiated over the SIMO line. Recall that in an SPI, as data is shifted out from the master into the slave(s), slave data is shifted-in into the master RX register (see Fig. 9.21 on p. 499). The received data is transferred into the UxRXBUF and the URXIFGx is set, denoting the completion of the TX/RX operation. Recall that in SPI transmission and reception occur simultaneously, and thus a data reception is initiated in the same way.

When in four-pin master mode, the STE input prevents conflicts with other potential masters in the bus. When STE is high, SIMO and UCLK lines operate normally, while STE in low makes them work as inputs.

In slave mode, having STE in high prevents the RX/TX operation. A master MSP430 SPI drives the slave STE input via an I/O port, while a slave GPIO output connects to the master STE. The reciprocal connection is necessary when the MSP430 SPI is a slave. Figure 9.33 shows the MSP430 connections in each case.

Although the SPI module uses the USART baud rate generator, only the prescaler portion takes effect to establish the data rate. The master device provides UCLK using its baud rate generator. In slave mode, the baud rate generator goes unused. SSELxx control bits allow for selecting the clock source. The maximum transfer rate allowed for a given BRCLK frequency is BRCLK/2.

The clock polarity and phase can be configured through the CKPL and CKPH control bits in the USART transmit control register (UxTCTL).

The SPI module has a single interrupt vector triggered by flag UTXIFGx when both UTXIEx and GIE are enabled. UTXIFGx is automatically cleared when the UxTBUF is written, upon a PUC, or when SWRST = 1.

9.5.2 The MSP430 USI

The Universal Serial Interface (USI) is one of the simplest serial peripheral interfaces supported by MSP430 devices. Featured exclusively in series x2xx devices, the USI supports only synchronous serial communications in the SPI or I^2C modes through a single module.

To operate in either mode, the USI is activated through the USI port enable control bits (USIPEx) in the USI control register USICTL0. These bits also define the mode and master/slave function of the module.

The USI clock module allows for choosing the phase and level of the clock signal. The USI clock can be fed from different sources that include SCLK, ACLK, SMCLK, SWCLK, TA0, TA1, or TA2, selected through USISSEL control bits in the USI clock control register (USICLKCTL). The clock division factor can be set to any power of two between 1 and 128.

MSP430 USI in SPI Mode

In the SPI Mode, the USI closely resembles the basic interface described in Sect. 9.4.1 on p. 499, as it features a single shift register, a bit counter, and a clock divider to implement a module that can be used as either SPI master or slave. The SPI mode of the USI is selected by clearing bit USII2C in the USICTL1 register.

The shift register (USISR) directly serves as data IN/OUT buffer, and is configurable to support 8- or 16-bit words. The bit counter provides indication of transfer completion, setting the USI flag (USIIFG) upon TX/RX completion. Figure 9.34 shows a simplified diagram with the USI module configured as SPI.

As an SPI master, the USI module generates the clock signal for the slave(s), so in its configuration a suitable clock source must be chosen. The master mode is chosen by setting the master bit (USIMST) in the USI control register USICTL0.

To initiate a master mode transfer, the character to be transmitted must be written to USISR and the actual transmission or reception begins when the USICNTx bits in the USI bit counter register (USICNT) are written with the number of bits to be transferred.

In the slave mode transfers are initiated similarly to the master mode, except that no clock source is specified as the master provides the clock signal. Before the a slave can transfer data, its output must be enabled by setting the control bit USIOE. This allows using the USI SPI in a multi-slave configuration without causing output conflicts when not selected. Note that the GPIO enabled slave select signals become necessary for any multidrop configuration.

As the entire USI module provides a single interrupt vector, whenever a USI interrupt is triggered, status indicators must be checked by the user software to determine the type of even that caused the exception.

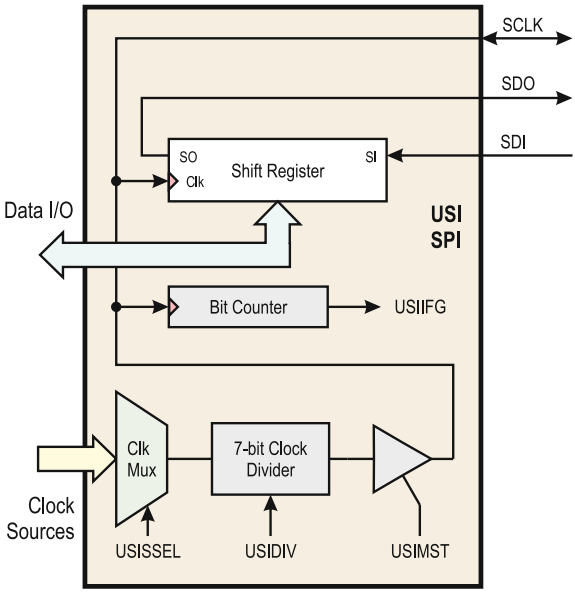


Fig. 9.34 Simplified structure of USI module in SPI mode

The example below illustrates the usage of the USI SPI to interface an external analog-to-digital converter (ADC) to an MSP430F20x2 in a simple voltage level indicator.

Example 9.5 (USI SPI Interface of a TLC549 8-bit ADC) *Provide a connection diagram for interfacing an external SPI serial ADC in 3-wire mode and an LED to an MSP430F2032. Write a C-language program solution to set the MSP430 as SPI master and turn the LED on when the analog input connected to the ADC falls below* $V_{DD}/4$.

Solution: *In this case, a suitable ADC is TI's TLC549, which provides a 3-wire simplex serial SPI interface [76]. As the ADC works as a read-only slave SPI peripheral, no SIMO line is needed nor is STE. The ADCs references VR+ and VR− are connected to* V_{DD} *and ground, respectively. The analog input is assumed to change between* V_{DD} *and GND. Figure 9.35 shows a connection diagram for this application.*

The software solution is adapted from a TI's USI SPI Interface demo written by M. Buccini and L. Westlund [77].[6] *The CPU clock is taken from the default DCO and the USI clock is chosen as SMCLK/4. The design is assumed to operated at* $V_{DD} = 3.3\,V$.

[6] See Appendix E.1 for terms of use.

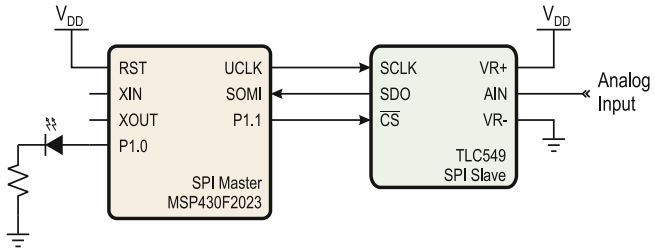


Fig. 9.35 Connecting an SPI external ADC to an MSP430 SPI master

```
//*******************************************************************
// Adapted from MSP430F20x2/3 Demo - USI SPI Interface to TLC549 8-bit ADC
// Original code by M. Buccini and L. Westlund - 10/2005
// IAR V3.40A/CCE V3.2.0 * Copyright (c) Texas Instruments, Inc.
//-------------------------------------------------------------------
#include <msp430.h>

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer
  P1DIR |= 0x03;                         // P1.0 & P1.1 as outputs
  P1OUT = 0;                             // LED Off & ADC CS asserted

  USICTL0 |= USIPE7 + USIPE5 + USIMST + USIOE; // Port, SPI master
  USICTL1 |= USIIE;                     // Counter interrupt, flag left set
  USICKCTL = USIDIV_2 + USISSEL_2;      // /4 SMCLK
  USICTL0 &= ~USISWRST;                 // USI released for operation
  USICNT = 8;                           // init-load counter
  _BIS_SR(LPM0_bits + GIE);             // Enter LPM0 w/ interrupt
}
//-------------------------------------------------------------------
// USI interrupt service routine
#pragma vector=USI_VECTOR
__interrupt void universal_serial_interTface(void)
{
  P1OUT |= 0x02;                              // Disable TLC549
  if (USISRL < 0x3F)
    P1OUT |= 0x01;
  else
    P1OUT &= ~0x01;
  P1OUT &= ~0x02;                              // Enable TLC549
  USICNT = 8;                                  // re-load counter
}
//*******************************************************************
```

MSP430 USI in I^{2}C Mode

In the I^{2}C Mode, the USI supports synchronous serial communication conformal to the Inter-Integrated Circuit protocol described in Sect. 9.4.2 on p. 501. The hardware configuration of the module expands that of SPI mode by including open-drain drivers for the bidirectional SDA and SCL lines, detectors for the start and stop

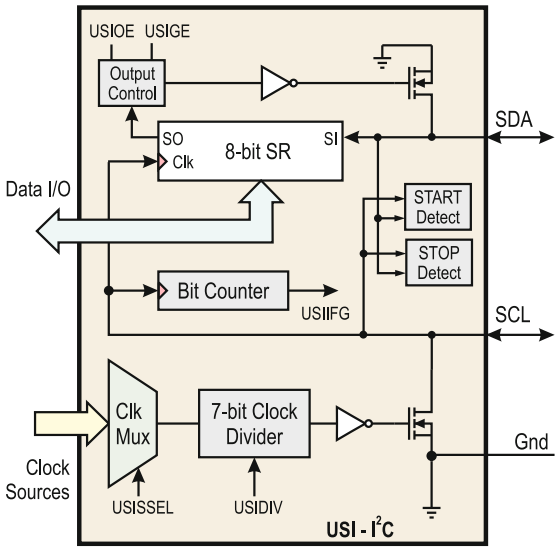


Fig. 9.36 Simplified structure of USI module in I^2C mode

conditions plus the output control logic needed to handle synchronization and arbitration inherent to the I^2C. Figure 9.36 shows a simplified diagram of the USI module in I^2C mode.

The I^2C mode of the USI is selected by setting bit USII2C in the USICTL1 register, while fixing the clock polarity and phase with USICKPL = 1, and USICKPH = 0. Moreover, character length must be fixed to eight bits and the SCL and SDA ports need to be enabled.

To set the module in master mode, the USIMST bit is set, and the module generates the transmission clock while USIIFG is clear. For slave mode operation, USIMST must be cleared. While as slave, the SCL line is set low by driving USIIFG and USISTTFG low while USICNTx is high. Flag USISTTIFG must be software cleared after setup to enable the reception of address fields at the beginning of each packet.

Transmissions require writing the output data into the shift register (USISRL) with the output enabled (USIOE = 1). Setting USICNTx to 08h (character length) will actually begin the transmission. When all eight characters are sent, USIIFG is set, and SCL is stopped. Reception of acknowledgement bits require software clearing USIOE.

I^2C reception requires disabling the output driver and writing 08h to USICNTx to set the required character length. When a character arrives USIIFG will be set, triggering an interrupt, if USIIE and GIE are enabled. An acknowledgment (or NACK) is sent by setting (clearing) the MSB of the USISRL, enabling the output and writing 01h to the USICNTx register. Once the ACK is sent the USIIFG will be set, allowing to prepare for the reception of the next character.

Start and stop conditions are sent by clearing or setting the MSB of the shift register and using USIGE and USIOE to drive the corresponding transitions on the SDA line by making the output latch transparent while SCL is high.

To prevent holding low the SCL line when the modules is in slave mode and has detected that it has not been addressed by the master, setting the USISCLREL bit in the USI Bit Counter Register (USICNT). This allows releasing SCL without requiring to clear the USIIFG.

The USI I^2C module can also be used in multi-master configurations, as it is capable of detecting a loss of arbitration in the case of contending masters. In an arbitration process, if the module looses arbitration by sending a one when the other contending master sent a zero, the USI arbitration lost flag (USIAL) in the USI Control Register 1 (USICTL1) will be set and the USIOE bit cleared, effectively relinquishing control of the bus. To detect such an event, user software must check the USIAL and USIIFG flags and configure the USI to slave receiver. The USIAL flag must be cleared by software.

9.5.3 The MSP430 USCI

The Universal Serial Communications Interface (USCI) is the most complete serial peripheral interface supported by MSP430 devices. Using a single hardware module, a USCI is able of supporting different serial communication formats including asynchronous and synchronous modes. USCI module configurations have been featured in MSP430 device families in series x4xx, x2xx, and x5xx/x6xx. Designators USCI_A, USCI_B are used to identify modules with different capabilities. Identical modules of the same kind coexisting in a particular MCU are labeled with a number following the designator (Ex. USCI_A0).

USCI_A modules can be seen as enhanced versions of the USART module described in Sect. 9.5.1 on p. 507, as both support UART and SPI communication modes. USCI_A enhancements include the ability to perform pulse shaping for IrDA[7] and automatic baud rate detection for supporting Local Interconnect Network[8] (LIN) protocols.

USCI_B modules support only synchronous serial communication modes in the form of SPI and I^2C, similar to the capabilities of the USI module described in Sect. 9.5.2 on p. 516.

Due to the similarities of USCI_A and USCI_B with the previous USART and USI modules, the discussion in the next sections will mainly focus in the enhancements brought up in the new modules, while referring to the earlier serial communication modules for the shared basic functionality aspects. For specific MSP430 devices, developers are strongly encouraged to refer to the family user's guide and device

[7] IrDA = Protocol for wireless infrared communication established by the Infrared Data Association.

[8] LIN is a serial communication format designed for localized vehicle networks.

data sheets for configuration details. Configuration and usage details change from one family to another and available resources change from one family member to another.

MSP430 USCI in UART Mode

The UART mode of the USCI, supported by USCI_A modules, provides all the capabilities listed for the USART in UART mode, with added features to encode and decode IrDA bit streams in the UC0RX and UC0TX lines. Moreover, the USCI UART has autobaud detection capabilities, a feature that allows using it for LIN communications. Figure 9.37 shows a block diagram of USCI_A in UART mode.

Configuration: Before attempting to configure the USCI_A UART, the device needs to be placed in reset mode by setting the UCSWRST flag. The USCI is placed in its asynchronous mode by clearing UCSYNC bit in the UCAxCTL0 control register. The overall sequence to initialize the USCI can be outlined as follows:

Step 1: Set UCSWRST to place the USCI in reset mode. UCSWRST is by default set by a PUC.

Step 2: Initialize all USCI registers, including UCAxCTL1, while holding UCSWRST = 1.

Step 3: Configure the USCI ports.

Step 4: Clear UCSWRST via software to enable the USCI.

Step 5: Enable interrupts via UCAxRXIE and/or UCAxTXIE.

The USCI asynchronous character format is consistent with that of the USART UART, illustrated in Fig. 9.30 on p. 508.

In its asynchronous mode the USCI can support point-to-point and multiprocessor transfers.

For common point-to-point connections, the USCI UART is configured in the idle-line mode, with no multiprocessor capability enabled. This is achieved by setting bits UCMODEx = 00 in control register UCAxCTL0.

Idle-Line and Address-Bit Multiprocessor Modes: When three or more devices need to be connected, the USCI UART can be configured in either idle-line or address-bit multiprocessor modes. Although these modes were explained in Sect. 9.5.1 on p. 507, they will be re-addressed here in terms of the USCI registers.

Bits UCMODEx in the UCAxCTL0 control register determine the mode of operation of the USCI.

For the USCI UART, the detection of an idle-line period is indicated with UCIDLE bit in the USCI_A status register (UCAxSTST). When in idle-line multiprocessor mode, the first received character after an idle period is an address. Recall that in this mode the address bit in the frame is not transmitted.

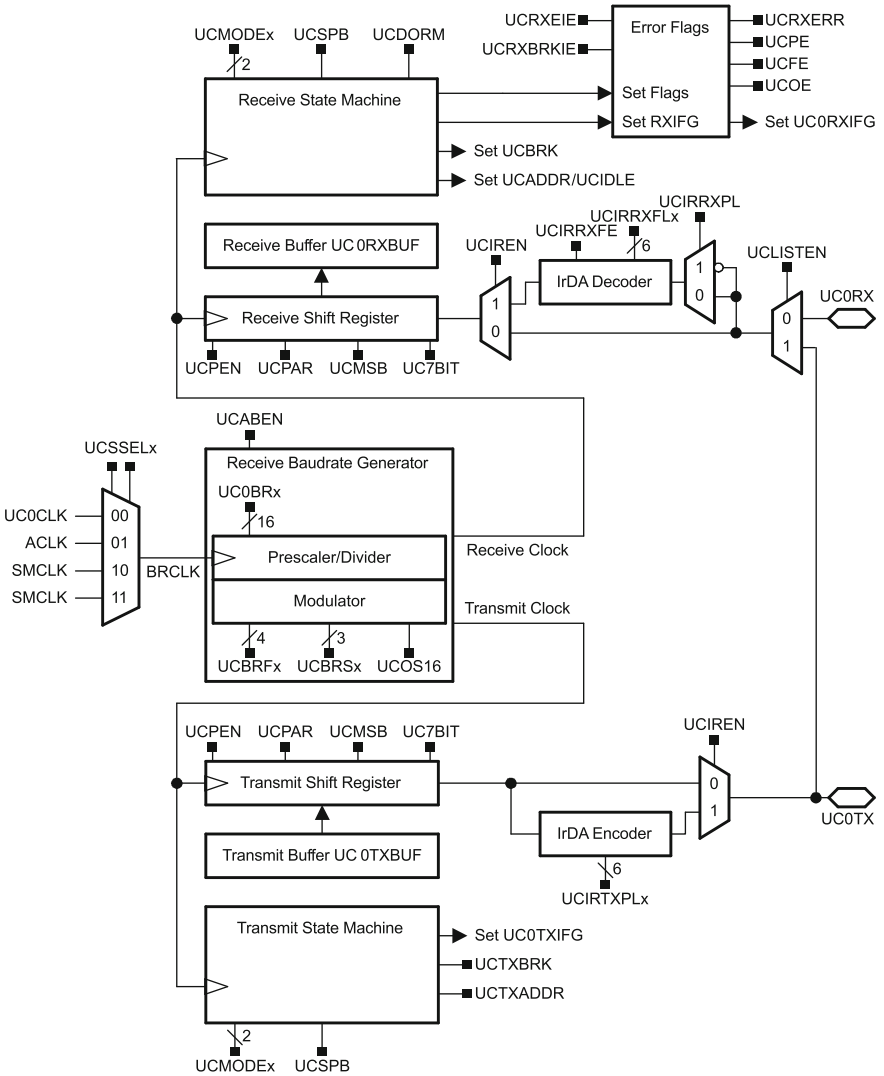


Fig. 9.37 Block diagram of USCI_A configured in UART mode (*Courtesy of Texas Instruments, Inc.*)

Upon an address reception, the incoming value is placed into the receive buffer and bit UCAxRXIFG set and control bit UCDORM is set, placing the USCI in dormant mode. In this mode, only address fields are received. Incoming data characters, although assembled, are not transferred into the receiving buffer and no reception interrupt is triggered. User software must validate the received address and, when matched, clear UCDORM to make possible the reception of incoming data. Once

UCDORM is clear, data characters will be transferred to the in-buffer and the reception flag set.

Three different actions can be initiated from the transmitter: sending an address field, a data field, or an idle frame. Transmission of an address field is preceded by an idle period. This is achieved by setting the UCTXADDR flag and then loading the address character into the UCAxTXBUF. UCAxTXIFG must be set for this action, indicating the transmitter is ready. The UART will then generate an idle line frame of eleven bits followed by the address character. UCTXADDR will be automatically reset. Afterwards, data characters to be sent can be transferred to the transmission buffer (upon transmitter ready flag set) ensuring the time between characters does not exceed the idle-line time to prevent their misinterpretation as addresses.

In address-bit multiprocessor mode each transmitted character includes the address bit to denote wether it corresponds to an address or data field. When a character with an address bit set is received, the UCADDR bit is set and the character placed in the reception buffer. UCAxRXIFG and UCDORM bits are set. User software must validated the address and clear UCDORM to enable reception of the incoming data. UCDORM set will allow receiving only address characters.

A transmission event must send first an address character. This is achieved by setting the UCTXADDR bit and placing the address character in the transmission buffer when the transmitter is ready. The transmission of the address character automatically clears UCTXADDR, so subsequent transmitted characters will be sent as data. In address-bit multiprocessor mode the time between characters has no significance.

Break Characters: In either point-to-point or any of the multiprocessor modes a break character might be received, triggering (if enabled) a break exception. A break is just a frame with all bits in zero. Transmission of a break is achieved by setting the UCTXBRK bit, and writing 00h to the transmission buffer. The transmission of the break character clears UCTXBRK.

Auto Baud Function: Setting UCMODEx to 11 in UART mode will enable auto baud detection. In this mode control registers UCAxBR0, UCAxBR1, and UCAxMCTL are set based on the measurement of a synchronization character (055h) inserted in front of a data frame after a break character. Control bit UCABDEN must be set to enable automatic measurement of the frame length. If the receiver were unable to synchronize, UCSTOE would be set.

Bit UCDORM controls data reception in autobaud mode. When set, only the break and synch fields will be accepted. User software must clear UCDORM to enable reception of the remaining characters in the stream. Auto baud events will set the UCAxRXIFG. If USCI RX interrupt and GIE are enabled, the triggered ISR needs to check the status flags to determine the channel status. Autobaud is recommended only for half duplex operation, as the USCI cannot transmit while receiving a break-synch sequence.

The transmission of a break-synch sequence is achieved by setting UCTXBRK with UMODEx $= 11$, then writing 055h to UCAxTXBUF when UCAxTXIFG $= 1$. UCTXBRK is automatically reset when the synch character is transmitted.

IrDA Pulse Shaping: This feature is enabled by setting UCIREN in the IrDA Transmit Control Register (UCAxIRTCTL). This causes pulsing the encoder to send a pulse through the transmit line for every zero in the transmitted data. UCIRTXPLx bits specify the pulse width. The decoder detects high pulses when UCIRRXPL $= 0$. Otherwise it detects low pulses.

Channel Operation: The operation of the UART channel proceeds according to the standard rules for asynchronous communication, discussed earlier in this chapter. Transmission and reception require the corresponding section of the UART to indicate their readiness. If enabled, these conditions can trigger their corresponding interrupts. If any error (framing, parity, overrun, or break) were detected, the corresponding error flags, UCRXERR, and the receiver flag will be set, and if enabled, would cause a reception interrupt. User software must check the error flags to determine if the reception was erroneous, what type of error caused it (if any), and the consequent corrective actions (if any) after an error is detected.

Baud Rate Generation: The USCI_A baud rate generator can operate in either a low-frequency or an oversampling modes, depending on whether UCOS16 bit in the USCI_A Modulation Control Register is clear or set.

In the low-frequency mode, the baud rate generator operates in a way similar to the USART baud rate generator, discussed in Sect. 9.5.1, with a few exceptions. Like in the USART case, it uses a prescaler and a modulator to produce standard baud rates from non standard frequency sources, where the maximum baud rate is one third the BRCLK. The module's power consumption is reduced by limiting the bandwidth of the baud rate generator clock BRCLK. The prescaling factor is set through the baud rate control registers UCAxBR0 and UCAxBR1, while the modulation factor, unlike the USART case, is now selected from a set of eight predefined values selected with control bits UCBRSx in the modulation control register UCAxMCTL. A simple formula now provides the value to choose:

$$\text{UCBRSx} = \text{round}[8 \cdot (N - \lfloor N \rfloor)], \tag{9.5}$$

where $N = f_{\text{BRCLK}}/(\text{baud rate})$. The device family user's manual lists the available modulation values and the recommended settings for standard baud rate values using typical frequencies.

In the oversampling mode (UCOS16 $= 1$), the generator also uses the approach of combining a prescaler and modulator to produce the desired baud rate. However, an intermediate bit clock 16 times the desired baud rate is generated from BRCLK and an additional divider produces the desired BITCLK. This approach limits the maximum baud rate to 1/16 the frequency of BRCLK.

The prescaler value is derived from the integer part of the quotient $f_{BRCLK}/(\text{baud rate})$, while the modulation factor is chosen from a list of 16 pre-determined values selected with UCBRFx bits in the Modulation Control Register UCAxMCTL.

The value of UCBRFx is determined as:

$$\text{UCBRFx} = \text{round}\left(N - 16 \cdot \left\lfloor \frac{N}{16} \right\rfloor\right) \tag{9.6}$$

As in the case of the USART baud rate generator, the actual bit times obtained in either the low-frequency or oversampled mode are not exact, and different frequencies will yield different levels of error for different target baud rates. The User's Manual of the selected target device provides tables with recommended values of prescaler and modulation factors for standard baud rates derived from several common frequencies obtained from the MSP430 clock generator and external crystals. These tables also include the estimated transmission and reception errors.

The following two examples below illustrate the usage of the USCI_A UART to implement the same application as in Example 9.4 on p. 512. In this both cases USCI_A0 is configured to work at 9,600 bps, 8N1 running from the DCO at 1 MHz. In the first case using the baud rate generator in low-frequency mode, while the second uses its oversampled mode.

Example 9.6 (USCI_A0 Echo at 9,600 bps on DCO at 1 MHz) *Write a program to implement a terminal echo using an MSP430G2452 running on the DCO at 1.0 MHz. Use a low-power mode to save energy when no transfer is taking place. The port configuration is 8N.*

Solution: *This solution assumes the MCU has is running from the internal DCO at 1.0 MHz. Pins P3.4 and P3.5 are used as TxD and RxD, respectively. A quick way to setup this system on the MSP430 Launchpad is using an FTDI Chip TTL-232RG cable connecting the MSP430 UART pins to a USB port in the PC. A Hyperterminal program or similar could be used to provide a quick interface on the PC. Below a brief code written by B. Nisarga from Texas Instruments providing the MSP430 side of the solution [78].*[9] *Observe that this solution is using the low-frequency mode of the baud rate generator with* $N = 1\,MHz/9{,}600 = 104.167\,Hz$ *and UCBRSx = 1. The DCO is configured using the standard calibration factor for 1 MHz provided in the header file, with a failsafe provision in case of value erasure.*

```
;-------------------------------------------------------------------------
;MSP430x24x Demo - USCI_A0, 9600 UART Echo ISR, DCO SMCLK - IAR V3.42A
;By B. Nisarga - 09/2007 * Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include <msp430.h>
;-------------------------------------------------------------------------
            RSEG    CSTACK                  ; Define stack segment
;-------------------------------------------------------------------------
            RSEG    CODE                    ; Assemble to Flash memory
;-------------------------------------------------------------------------
RESET       mov.w   #SFE(CSTACK),SP         ; Initialize stack pointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
CheckCal    cmp.b   #0xFF,&CALBC1_1 MHZ      ; Check calibration constant
            jne     Load                    ; if not erased, load.
Trap        jmp     Trap                    ; else, do not load, trap CPU!
Load        clr.b   &DCOCTL                 ; Select lowest DCOx and MODx
            mov.b   &CALBC1_1 MHZ,&BCSCTL1   ; Set DCO to 1 MHz
```

[9] See Appendix E.1 for terms of use.

```
            mov.b   &CALDCO_1 MHZ,&DCOCTL    ;
SetupP3     bis.b   #030h,&P3SEL             ; Use P3.4/P3.5 for USCI_A0
SetupUSCI0  bis.b   #UCSSEL_2,&UCA0CTL1      ; SMCLK
            mov.b   #104,&UCA0BR0            ; 1 MHz 9600
            mov.b   #0,&UCA0BR1              ; 1 MHz 9600
            mov.b   #UCBRS0,&UCA0MCTL        ; Modulation UCBRSx = 1
            bic.b   #UCSWRST,&UCA0CTL1       ; **Init USCI state machine**
            bis.b   #UCA0RXIE,&IE2           ; Enable USCI_A0 RX interrupt
                                             ;
Mainloop    bis.b   #CPUOFF+GIE,SR           ; Enter LPM0, interrupts enabled
            nop                              ; Needed only for debugger
                                             ;
;-------------------------------------------------------------------------
USCI0RX_ISR;  Echo back RXed character, confirm TX buffer is ready first
;-------------------------------------------------------------------------
TX0         bit.b   #UCA0TXIFG,&IFG2         ; USCI_A0 TX buffer ready?
            jz      TX0                      ; Jump if TX buffer not ready
            mov.b   &UCA0RXBUF,&UCA0TXBUF    ; TX -> RXed character
            reti                             ;
                                             ;
;-------------------------------------------------------------------------
            COMMON  INTVEC                   ; Interrupt Vectors
;-------------------------------------------------------------------------
            ORG     USCIAB0RX_VECTOR         ; USCI0 Rx Vector
            DW      USCI0RX_ISR              ;
            ORG     RESET_VECTOR             ; RESET Vector
            DW      RESET                    ;
            END
;=========================================================================
```

Example 9.7 (USCI_A0 Echo at 9,600 bps on DCO at 1 MHz—C) *Repeat Example 9.6 providing a solution in C Language. Change the baud rate generator to operate in oversampled mode while keeping unchanged all other parameters.*

Solution: *A porting of the solution in Example 9.6 running on CCE V3.2.0 and IAR V3.42A and using the baud rate generator oversampled mode is provided by B. Nisarga [79].*[10] *By keeping the same clock source (DCO at 1 MHz), the 16x requirement between BRCLK and the target baud rate is satisfied. In this case,* $N = 1\,MHz/9{,}600\,Hz = 104.17$ *and the modulation factor from (9.6) yields* $UCBRSx = round(104.17 - 16\lfloor 104.17/16 \rfloor) = 8$. *Again, a calibrated DCO is used for MCLK =* $1\,MHz$.

```
//=========================================================================
// MSP430x24x Demo - USCI_A0, 9600 UART, SMCLK, LPM0, Echo w/over-sampling
// By B. Nisarga - 09/2007 * Copyright (c) Texas Instruments, Inc.
//-------------------------------------------------------------------------
#include <msp430.h>

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;           // Stop WDT
  if (CALBC1_1MHZ==0xFF)              // If calibration constant erased
  {
    while(1);                         // do not load, trap CPU!!
```

[10] See Appendix E.1 for terms of use.

```
  }
  DCOCTL = 0;                           // Select lowest DCOx & MODx values
  BCSCTL1 = CALBC1_1MHZ;                // Set DCO
  DCOCTL = CALDCO_1MHZ;
  P3SEL = 0x30;                         // P3.4,5 = USCI_A0 TXD/RXD
  UCA0CTL1 |= UCSSEL_2;                 // SMCLK
  UCA0BR0 = 6;                          // 1 MHz 9600
  UCA0BR1 = 0;                          // 1 MHz 9600
  UCA0MCTL = UCBRF3 + UCOS16;           // Modln UCBRSx=8, over sampling
  UCA0CTL1 &= ~UCSWRST;                 // **Initialize USCI FSM**
  IE2 |= UCA0RXIE;                      // Enable USCI_A0 RX interrupt

  __bis_SR_register(LPM0_bits + GIE); // Enter LPM0, interrupts enabled
}

// Echo back RXed character, confirm TX buffer is ready first
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
  while (!(IFG2&UCA0TXIFG));                // USCI_A0 TX buffer ready?
  UCA0TXBUF = UCA0RXBUF;                    // TX -> RXed character
}
//============================================================================
```

MSP430 USCI in SPI Mode

The USCI SPI mode is functionally equivalent to that of the USART SPI explained in Sect. 9.5.1 on p. 501. Both modules offer three- and four-pin SPI operation with separate receive and transmit shift registers. As the SPI baud rate generator does not use the modulator section (disabled in the USART SPI), the USCI SPI mode omits it from its structure. Other than that, both modules are equivalent, and understanding the functionality of one of them allows to operate the other. The reader is referred to the description of the USART SPI for functional information. For programming details, register and control/status bit names, and available units, the reader is referred to the device-specific data sheets and user manuals.

MSP430 USCI in I^2C Mode

The synchronous I^2C serial communication modules sported in USCI_Bx modules, are designed to fully comply with Phillips Semiconductor I^2C specification V2.1. By using separate shift registers for the transmit and receive sections, the USCI_B I^2C architecture, illustrated in Fig. 9.38, closely resembles the structure depicted in Fig. 9.24 on p. 502 for a standard interface for this communication modality.

It supports operation as either master or slave in single or multimaster buses. Moreover, this module, featured in MSP430 devices series x4xx, x2xx, and x5xx/x6xx, is designed for compatibility with the MCU low-power mode operation.

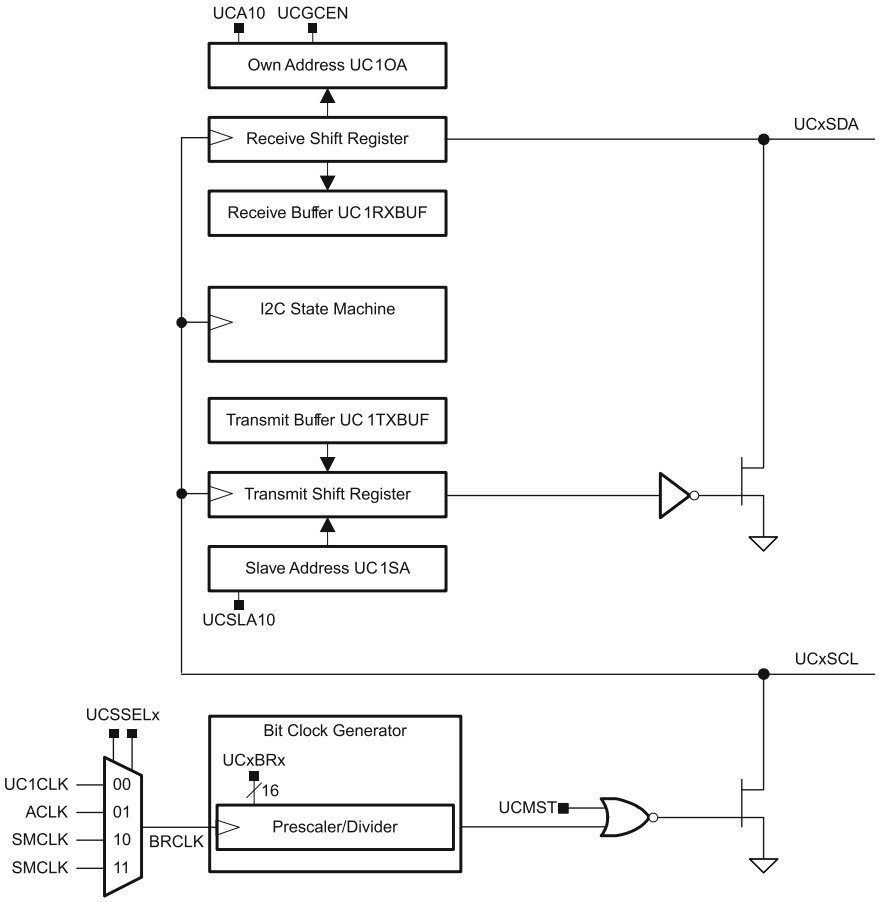


Fig. 9.38 Block diagram of USCI_B configured in I^2C mode (*Courtesy of Texas Instruments, Inc.*)

Like any other USCI module, the I^2C is reset upon a PUC by setting control bit UCSWRST in the UCxCTL1 UCSI control register 1. It requires that all configuration be made with the module in reset mode to prevent undesirable behavior.

In adherence to Phillips standard, the module supports 7- and 10-bit address fields. For ten-bit addresses, the standard requires two consecutive 8-bit characters: the first containing the pattern $11110a_9a_8$ plus the R/$\overline{\text{W}}$ and acknowledged by the addressed slave, and the second containing address bits $a_7 \ldots a_0$, with no R/$\overline{\text{W}}$ field like a regular data character.

Unlike the simple operation of USI modules where many of the tasks related to using the bus rely on user software, USCI_B I^2C supports most aspects of the communication by hardware actions. Master transmit/receive, slave transmit/receive, clock synchronization, bus arbitration, address generation and clock stretching are

all supported requiring minimal user software. SW actions fundamentally need to enable and configure the module in the desired mode, and afterwards essentially accessing the transmit and/or receive buffers upon activation of the receiver and transmitter ready flags. A few isolated instances require user software actions during transfers. These include a master initiating a transmission or reception transfer, a master aborting a transfer, a master issuing a restart condition, or a master/slave issuing a NACK condition when the acknowledgment of the last transmitted character is not received.

As the protocols needed to establish an I^2C communication are described in detail in Sect. 9.4.2 on p. 501, we refer the reader to that section for understanding the protocol. The programming details specifying particular register names and status/control flags are dependent on the instance of USCI_Bx being used in a specific MSP430 device. The reader is referred to the corresponding family User's Manual for a detailed description of them.

A few observations deserve to be made about the module.

- Although each event occurring in an USCI module has it own indication flag and enable bits, the entire module has a single interrupt vector multiple-sourced by the transmitter, receiver, and status sections. This implies that USCI interrupt service routines need to check the corresponding flags to determine the actual cause of the interrupt.
- For convenience in identifying USCI I^2C interrupts, all triggering flags are prioritized and combined in register UCBxIV, facilitating the implementation of a jump table based on the value of this register (see Sect. 7.3.3 on p. 318 for an example). Any R/W access to this register automatically clears the highest-pending interrupt flag.
- There are six fundamental events that could trigger an interrupt from the USCI module in I^2C module. These include:

  1. **Transmitter Ready:** Indicated by UCTXIFG to denote that the transmission buffer (UCBxTXBUF) is ready to accept a new character. Automatically cleared when UCBxTXBUF is written.
  2. **Receiver Ready:** Signaled by UCRXIFG flag to indicate that a new character has been received and loaded into the reception buffer (UCBxRXBUF). Automatically cleared when UCBxRXBUF is read.
  3. **Arbitration Lost:** Indicates the device lost a bus arbitration process. Signaled with the UCALIFG flag.
  4. **No-acknowledgment (NACK):** Signaled with UCNACKIFG flag to denote an expected acknowledgment was not received. Cleared by the reception of a start condition.

5. **Start Condition:** Indicates the detection of a start condition and valid address by a slave device. This condition is exclusive for slave configured modules. Automatically cleared when a stop condition is detected.
6. **Stop Condition:** This status is also exclusive for slave devices and indicates the detection of a stop condition in the bus. Automatically cleared when a start condition is detected, regardless the address.

- When the module is used in conjunction with a DMA channel, the DMA transfer requests are triggered from the USCI via the UCTXIFG and UCRXIFG flags.
- To support low-power mode operation, the USCI clock source is automatically activated by transfer events. As only master configured modules can generate the SCL signal, this activation is not necessary in devices configured as slaves.

The examples below illustrate how to use USCI_B in its I^2C mode to implement a master-slave bus connection between two MSP430x5xx devices. The first example will illustrate the master side, while the second the slave side.

Example 9.8 (USCI_B0 I2C MSP430 Master-slave TX) *Provide a program to transmit via an* I^2C *bus master a multi-character message stored somewhere in memory.*

Solution: *The solution will use two MSP430F5438 connected one as master, the other as slave via their I2C interfaces (P3.1 = SDA and P3.2 = SCL). This solution provides a transmitter code demo written in C-language by D. Dang [80].*[11] *The code uses the internal DCO frequency through SMCLK to feed BRCLK. The actual data rate will be around 100 Kbps as BRCLK is further divided by 100. The MCU is placed in LPM0 after the data to be transmitted is placed in the transmission buffer. The transmission ISR illustrates how to process the multi-sourced USCI interrupt, in this case using a switch statement.*

```
//******************************************************************************
// MSP430F543xA Demo - USCI_B0 I2C Master multi-byte TX to MSP430 Slave
// By D. Dang - 12/2009 * Copyright (c) Texas Instruments, Inc.
//------------------------------------------------------------------------------
#include <msp430.h>

unsigned char *PTxData;                     // Pointer to TX data
unsigned char TXByteCtr;

const unsigned char TxData[] =              // Table of data to transmit
{
  0x11,  0x22,  0x33,  0x44,  0x55
};

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P3SEL |= 0x06;                            // Assign I2C pins to USCI_B0
  UCB0CTL1 |= UCSWRST;                      // Enable SW reset
```

[11] See Appendix E.1 for terms of use.

```
  UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;     // I2C Master, synchronous mode
  UCB0CTL1 = UCSSEL_2 + UCSWRST;            // Use SMCLK, keep SW reset
  UCB0BR0 = 12;                             // fSCL = SMCLK/12 = ~100 kHz
  UCB0BR1 = 0;
  UCB0I2CSA = 0x48;                         // Slave Address is 048h
  UCB0CTL1 &= ~UCSWRST;                     // Clear SW reset, and resume
  UCB0IE |= UCTXIE;                         // Enable TX interrupt

  while (1)
  {
    __delay_cycles(50);                     // Delay  between transactions
    PTxData = (unsigned char *)TxData;      // TX array start address
                                            // Place breakpoint here to see
                                            // each transmit operation.
    TXByteCtr = sizeof TxData;              // Load TX byte counter

    UCB0CTL1 |= UCTR + UCTXSTT;             // I2C TX, start condition

    __bis_SR_register(LPM0_bits + GIE);     // Enter LPM0, enable interrupts
    __no_operation();                       // Remain in LPM0 until all data
                                            // is TX'd
    while (UCB0CTL1 & UCTXSTP);             // Ensure stop condition got sent
  }
}

//------------------------------------------------------------------------------
// The USCIAB0TX_ISR is structured such that it can be used to transmit any
// number of bytes by pre-loading TXByteCtr with the byte count.
// Also, TX Data points to the next byte to transmit.
//------------------------------------------------------------------------------
#pragma vector = USCI_B0_VECTOR
__interrupt void USCI_B0_ISR(void)
{
  switch(__even_in_range(UCB0IV,12))
  {
  case  0: break;                           // Vector  0: No interrupts
  case  2: break;                           // Vector  2: ALIFG
  case  4: break;                           // Vector  4: NACKIFG
  case  6: break;                           // Vector  6: STTIFG
  case  8: break;                           // Vector  8: STPIFG
  case 10: break;                           // Vector 10: RXIFG
  case 12:                                  // Vector 12: TXIFG
    if (TXByteCtr)                          // Check TX byte counter
    {
      UCB0TXBUF = *PTxData++;               // Load TX buffer
      TXByteCtr--;                          // Decrement TX byte counter
    }
    else
    {
      UCB0CTL1 |= UCTXSTP;                  // I2C stop condition
      UCB0IFG &= ~UCTXIFG;                  // Clear USCI_B0 TX int flag
      __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
    }
  default: break;
  }
}
//------------------------------------------------------------------------------
```

This second example illustrates how to use the MSP430 USCI in its I^2C slave mode to receive the message string sent by the code in the previous example.

Example 9.9 (USCI_B0 I2C MSP430 Master-slave TX) *Provide a program to receive via an I^2C bus slave a multi-character message and store it somewhere in memory.*

Solution: *The solution will use two MSP430F5438 connected one as master, the other as slave via their I2C interfaces (P3.1 = SDA and P3.2 = SCL). This solution provides the receiver code using a demo written in C-language by P. Thanigai and M. Morales [81].*[12] *The code uses the internal DCO frequency for the CPU, but as the I^2C interface is configured in slave mode, no clock generator is needed. The MCU is placed in LPM0. The receiver ready state of the USCI is used to wake-up the CPU and place the received data in memory.*

```
//******************************************************************************
// MSP430F543xA Demo - USCI_B0 I2C Slave multi-byte RX from MSP430 Master
// By P. Thanigai and M. Morales - 06/2009
// Copyright (c) Texas Instruments, Inc.
//------------------------------------------------------------------------------
#include <msp430.h>

unsigned char *PRxData;                     // Pointer to RX data
unsigned char RXByteCtr;
volatile unsigned char RxBuffer [128];       // Allocate 128 byte of RAM

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT

  P3SEL |= 0x06;                            // Assign I2C pins to USCI_B0
  UCB0CTL1 |= UCSWRST;                      // Enable SW reset
  UCB0CTL0 = UCMODE_3 + UCSYNC;             // I2C Slave, synchronous mode
  UCB0I2COA = 0x48;                         // Own Address is 048h
  UCB0CTL1 &= ~UCSWRST;                     // Clear SW reset and resume
  UCB0IE |= UCSTPIE + UCSTTIE + UCRXIE;     // Enable STT, STP & RX interrupt

  while (1)
  {
    PRxData = (unsigned char *)RxBuffer;    // Start of RX buffer
    RXByteCtr = 0;                          // Clear RX byte count
    __bis_SR_register(LPM0_bits + GIE);     // Enter LPM0, enable interrupts
                                            // Remain in LPM0 until master
                                            // finishes TX
    __no_operation();                       // Set breakpoint *here* and read
  }                                         // read out the RxData buffer
}

//------------------------------------------------------------------------------
// The USCI_B0 data ISR RX vector is used to move received data from the I2C
// master to the MSP430 memory. It wakes the CPU from LPM0 to process
// received data in the main program upon (re)start or stop conditions.
//------------------------------------------------------------------------------
#pragma vector = USCI_B0_VECTOR
__interrupt void USCI_B0_ISR(void)
```

[12] See Appendix E.1 for terms of use.

```
{
  switch(__even_in_range(UCB0IV,12))
  {
  case  0: break;                           // Vector  0: No interrupts
  case  2: break;                           // Vector  2: ALIFG
  case  4: break;                           // Vector  4: NACKIFG
  case  6:                                  // Vector  6: STTIFG
    UCB0IFG &= ~UCSTTIFG;
    break;
  case  8:                                  // Vector  8: STPIFG
    UCB0IFG &= ~UCSTPIFG;
    if (RXByteCtr)                          // Check RX byte counter
      __bic_SR_register_on_exit(LPM0_bits);
    break;
  case 10:                                  // Vector 10: RXIFG
    *PRxData++ = UCB0RXBUF;                 // Get RX'd byte into buffer
    RXByteCtr++;
    break;
  case 12: break;                           // Vector 12: TXIFG
  default: break;
  }
}
//============================================================================
```

9.5.4 The MSP430 Enhanced USCI

The Enhanced Universal Serial Communication Interface (eUSCI) is a modified version of the USCI module discussed in the previous section. The eUSCI is featured in MSP430x5xx/x6xx devices, and at the time of this printing had been released in the form of eUSCI_A and eUSCI_B. Like their USCI predecessors, eUSCI_A supports UART and SPI communications, while eUSCI_B supports ^{2}C and SPI modes.

The eUSCI_A

The eUSCI_A in UART mode is functionally equivalent to the USCI_A UART, except in the form of generating the baud rate clock. Its support for fractional dividers, although still using a prescaler and modulator, now features an 8-bit UCBRSx code for selecting one of 256 modulation patterns when operated in the low-frequency mode. This offers an improved baud rate approximation, reducing the error due to deviation from the desired baud rate in both the transmitter and receiver sides. The device family User's Manual provides revised tables for determining the values for configuration registers given the desired baud rate and clock frequency [59]. The tables also provide the recalculated expected errors resulting from using this new scheme.

The oversampled baud rate generation mode, also supported in the eUSCI_A UART remains the same as in the standard USCI.

In SPI mode, eUSCI_A is functionally and structurally equivalent to its predecessors in the original USCI module. Addresses, register names, and flags change, so the reader is referred to the datasheets and family's manuals of the specific device used for the specific nomenclature of configuration registers and flags.

The eUSCI_B

The eUSCI_B in I^2C mode provides improvements over its predecessor by including extra features that facilitate its usage when operated via interrupts and coordinated with DMA channels. Specifically, eUSCI_B adds the following features:

- An 8-bit byte counter (UCBxBCNT) with interrupt capability and automatic STOP assertion. This counter allows to keep track of the number of characters that have been transferred over the I^2C bus since the last start condition was detected. A threshold value can be programmed to generate an interrupt when reached or automatically generating stop condition in the bus (master mode only).
- Up to four hardware slave addresses, each having its own interrupt and DMA trigger. This addition helps accelerating transactions when a slave module is needs to be operated responding to multiple addresses. Although this kind of operation is possible via user software, the addition of hardware addresses and their corresponding DMA channels allows for using this feature without leaving a low-power mode.
- Mask register for slave address and address received interrupt. This new feature allows extending the number of addresses to which a slave module can respond. When an address has been masked, the address match only looks at the unmasked address bits, resulting the masked bits as don't care conditions.
- Clock-Low-Timeout interrupt to avoid bus stalls. This feature allows detecting if the SCL line is held low by a clock stretching operation longer than a predefined period. An optional interrupt can be triggered with this condition.

As a result of the enhancements introduced into the eUSCI, the module now supports sixteen interrupt conditions, ten more than its predecessor. Despite these additional sources, the eUSCI_B continues to feature a single interrupt vector, making necessary to apply multi-sourced ISR techniques in its programming. Despite the enhancements, the module functionality remains compatible in most aspects to the standard I^2C description. This allows to use the description of its predecessor to understand its functionality and referring to the user's manual and data sheet of the specific device used for details regarding addresses, registers, and flags.

The operation of eUSCI_B in SPI mode, like that of eUSCI_A remains unchanged with respect to that of the original USCI.

9.6 Chapter Summary

Serial communications have become the most widely used data transfer modality in modern embedded applications. This chapter provided a comprehensive discussion of the fundamentals of serial communication, establishing the conceptual and practical foundations for designing and using serial channels in embedded applications.

From the elemental protocols developed for RS-232 to the intricacies of the universal serial bus, the sections presented systematically discussed asynchronous and synchronous protocols for serial channels.

Formats and interfaces that include UART, SPI, I^2C, and USB were discussed. Physical standards for short and long range channels, at low-, medium-, and high-speed were discussed in detail.

Lastly, an exploration of the serial modules embedded in all MSP430 series devices were analyzed and illustrative examples provided to guide the reader through an learning experience of serial subjects.

9.7 Problems

9.1 A router's console port is used to communicate with a computer using a serial communication port. Both the router and the computer must be set using the same parameters. If the communication is set for 115,200 bits per second, 8 data bits, and 1 stop bit, how long would a 1 MB file take to be transferred from the computer to the router?

9.2 What is the purpose of the start bit in an asynchronous serial communication? How does the receiver synchronize with the transmitter?

9.3 Two computers are communicating using a serial communication channel. However, garbage is seeing at the receiving end. What is the possible cause of this problem?

9.4 In an SPI connection the master and slave need to communicate. What would the slave have to do to send data to the master? What would the master need to do to send data to the slave? Which station controls the data transfer and generates the clock? Can there be several slaves selected at the same time?

9.5 Draw a time diagram showing how would the following data be transmitted over a serial connection: 01100111.

9.6 Parallel communication had many advantages over serial communication. However, it was serial communication that ended on top. Can you explain why?

9.7 Provide a connection diagram and software modules to establish a master-slave I^2C connection between two MSP430 launchpads connected one as master and

the other as slave using the USI module. Provide a code implantation including both, master and slave sides in assembly language.

9.8 Show a derivation of the formula to estimate the transmitter baud rate error in the USCI module of a series x5xx MSP430 when configured in oversampled mode. Compare it against the formula in the device user's manual. Comment about your result.