به نام خدا

ارتباط سریال در AVR

آشنایی با ارتباط سریال

Dr. Aref Karimiafshar
A.karimiafshar@iut.ac.ir

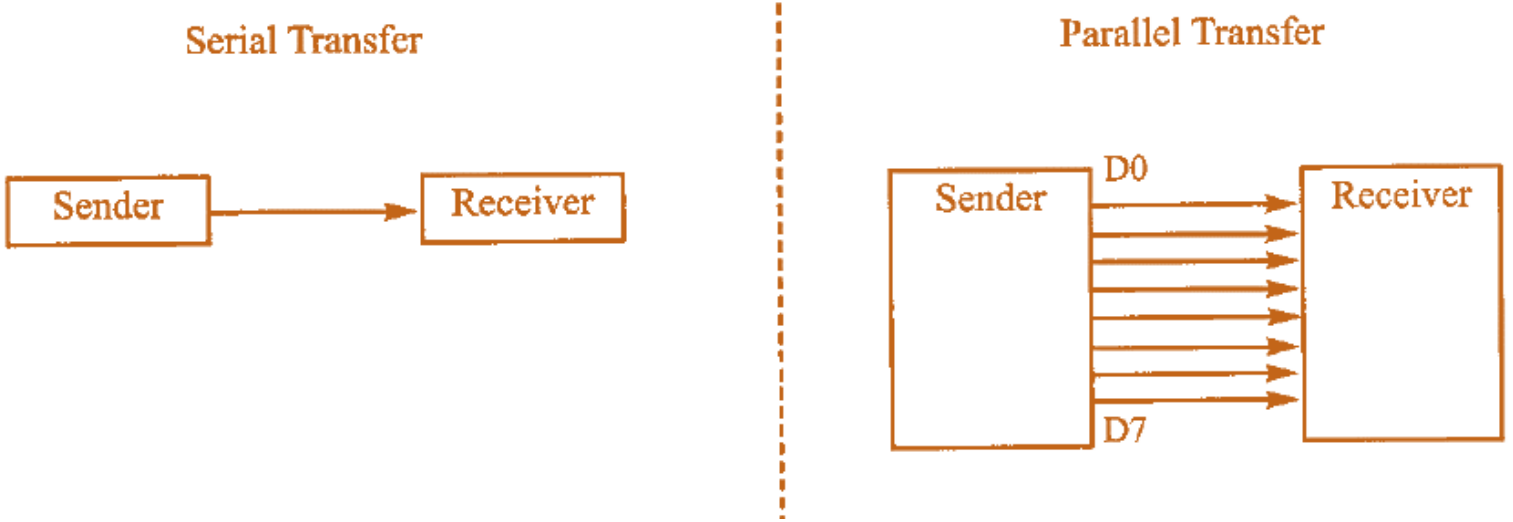
Transferring Data

- Two ways of data transferring
  - Parallel
    - Eight or more lines are used to transfer data
      - A few meters away (short distance)
        - » Printers and IDE hard disks
  - Serial
    - Send one bit at a time
      - Many meters away
        - » keyboards

در یک دسته بندی می تونیم روش های انتقال داده رو به دو دسته موازی و سریال تقسیم بندی بکنیم:
موازی:
در هنگام انتقال داده به صورت موازی ما معمولا میایم و یک بایت یا بیشتر رو انتقال میدیم و این مستلزم اینه که ما حداقل 8 خط یا بیشتر رو برای انتقال داده در نظر گرفته باشیم ولی هر چند این انتقال داده می تونه پر سرعت باشه ولی برای فاصله های کوتاه مفید است مثل پرینترهای قدیمی یا هارد دیسک های IDE
سریال:
در هر زمان میاد و فقط یک بیت رو ارسال میکنه - این انتقال برای فواصل دور هم مفید است مثل کیبورد

Data Transferring

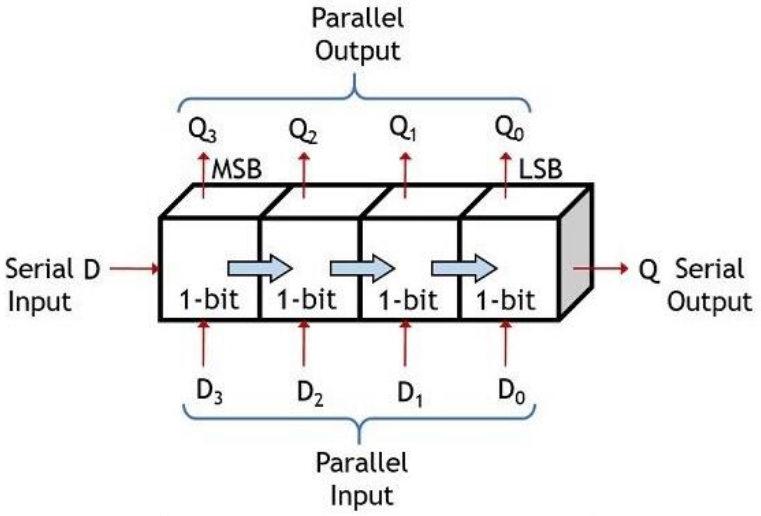
When a microprocessor communicates with the outside world, it provides the data in byte-sized chunks. For some devices, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the device. This can work only if the cable is not too long, because long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart.



وقتی یک میکروپروسسور میخواد با دنیای بیرون ارتباط برقرار بکنه این ارتباط به صورت بایتی است و اون میاد بایتش رو روی باس داده قرار می ده و این منتقل میشه و اگر ما 8 خط داده داشته باشیم این کار به راحتی انجام میشه ولی اگر بخوایم اینو در قالب بایتی انجام بدیم مجبوریم از اون دسته بندی موازی استفاده بکنیم که برای فواصل طولانی ممکنه چندان مناسب نباشه چون به یک سری مشکلاتی برخورد می کنیم توی انتقال سیگنال ها و هزینه این کار هم خیلی بیشتر است برای همین می ریم سراغ ارتباط سریال
ارتباط سریال می تونه برای فواصل طولانی تر مناسب باشه

Serial Data Communication

- For serial data communication to work
  - Byte of data must be converted to serial bits
    - Parallel-in-Serial-out shift register
      - Then it can be transmitted over a single data line
  - At the receiving end
    - Serial-in-Parallel-out shift register
      - Then pack them into a byte



برای اینکه یک انتقال داده بیاد و به صورت سریال انجام بشه ما باید بایت داده رو به صورت سریال در اختیار اون رسانه ای که قراره اینو انتقال بده قرار بدیم این کار از طریق یک شیفت رجیستری انجام میشه تحت عنوان Parallel-in-Serial-out --> ما به صورت موازی داده رو وارد میکنیم ینی یک بایت رو در اختیار این شیفت رجیستر قرار میدیم و اون به صورت سریال بیت به بیت روی خط اونو ارسال میکنه و سمت گیرنده هم یک اتفاق مشابه باید بیوفته ینی ما داریم بیت ها رو این بار تک به تک دریافت میکنیم اما باید اونو به صورت بایت در اختیار سیستم قرار بدیم ینی اون بیت هایی که میاد رو بسته بندی میکنیم و تحت عنوان یک قالبی می تونه مثل به صورت بایت در اختیار اون سیستم قرار بگیره و ازش استفاده بکنه

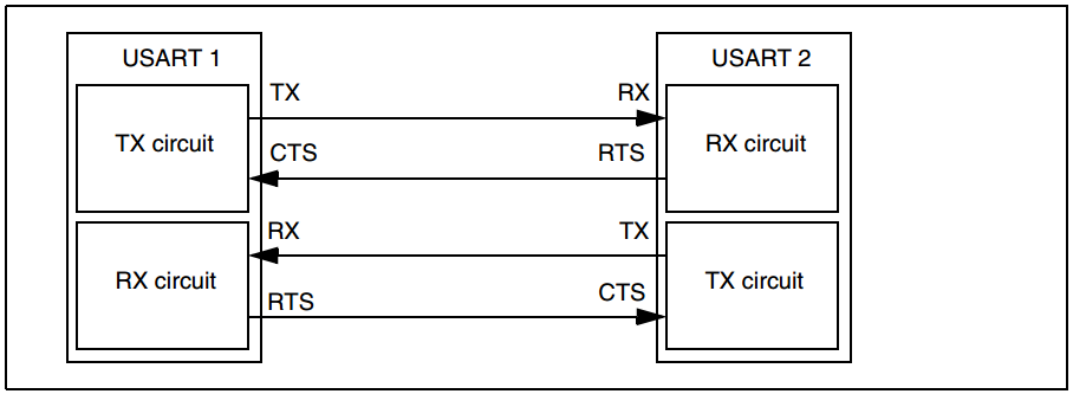
توی شکل:

داده به صورت موازی توی این شیفت رجیستر قرار میگیره و بعد بیت به بیت به صورت سریال از این رجیستر خارج میشه و وقتی که میخوایم دریافت بکنیم بیت ها دونه به دونه اون ور دریافت میشه و وقتی که همش دریافت شد به صورت یک بایت در اختیار سیستم قرار میگیره

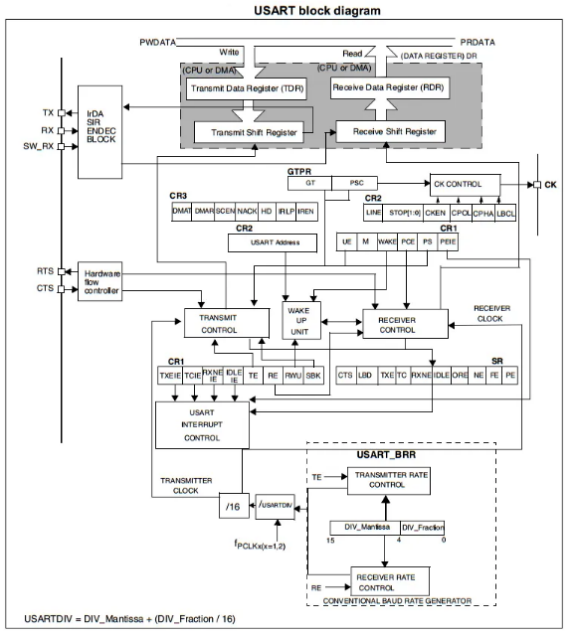
Methods Serial Data Communication

Serial data communication uses two methods, asynchronous and synchronous. The *synchronous* method transfers a block of data (characters) at a time, whereas the *asynchronous* method transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, special IC chips are made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The AVR chip has a built-in USART.

Hardware flow control between two USARTs



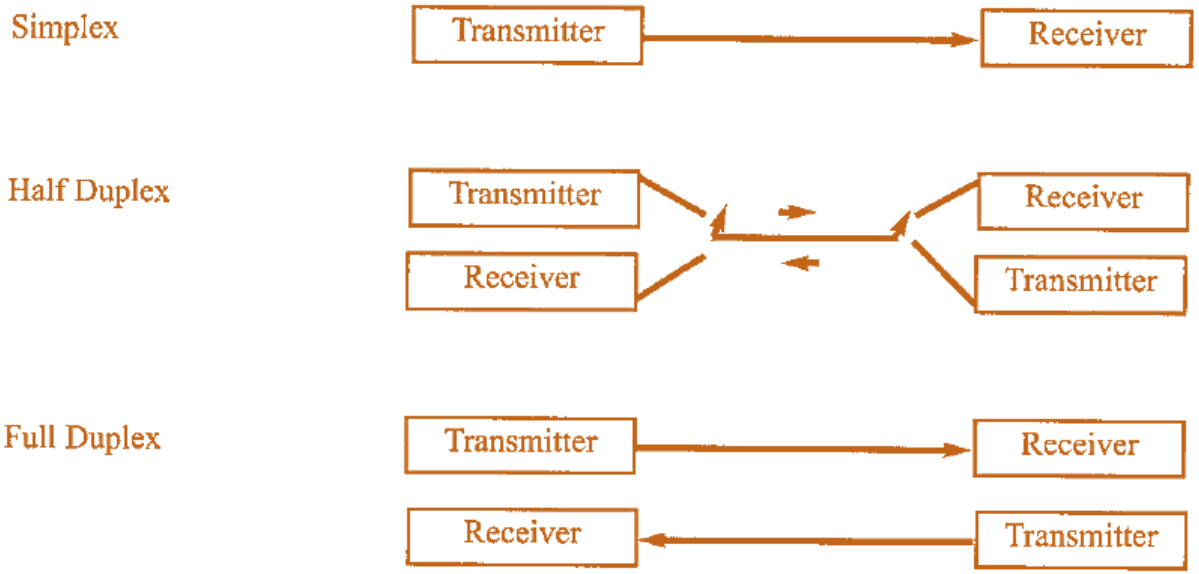
USART block diagram



از دیدگاه دیگری: ما روش انتقال سریال رو میتونیم به دسته synchronous , asynchronous تقسیم بندی بکنیم:
توی حالت synchronous: معمولا این ها روش هایی هستند که میان یک بلاک از داده ها رو به صورت همزمان منتقل میکنه اما در روش asynchronous در هر زمان فقط یک بایت هستش که میاد و منتقل میشه
انجام این دو نوع انتقال در زیر دسته در خانواده انتقال سریال می تونه به صورت نرم افزاری توی میکرو انجام بشه که یک کار بسیار پرزحمتی خواهد بود و چندان مورد اقبال قرار نگرفته برای همین یکسری ICهایی وجود داره که توسط کاربردهای متفاوتی این ها تولید شده و این ارتباط سریال رو برای ما برقرار میکنه و این ها به دوتا دسته خیلی معروف تقسیم میشن: UART , USART
UART فقط نوع asynchronous برای ما انجام میدن
USART که هم به صورت asynchronous و هم به صورت synchronous میاد این انتقال سریال رو مدیریت میکنه و برای ما انجام میده
AVR هم این تراشه رو به صورت سخت افزاری داخل خودش داره و نیاز نیست که این ها رو اون استفاده کننده به صورت نرم افزاری انجام بده

Methods Serial Data Communication

In data transmission, if the data can be both transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously.

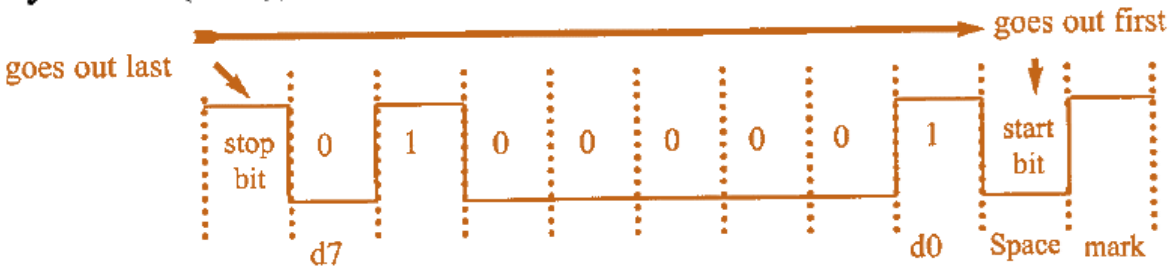


خطوطی که ما داریم می تونه یک مقدار وابسته به اون سیستمی که میخوایم استفاده بکنیم متفاوت باشه که این ها معمولا در سه دسته تقسیم بندی میشن:
فقط از یک سمت امکان ارتباط با سمت دیگر رو داریم ینی یک سمت ارسال میکنه و سمت دیگر دریافت میکنه: simplex
یا به صورت duplex توی این حالت می تونه یا half.. باشه در این حالت ما در هر زمان امکان ارسال از یک سمت به سمت دیگر رو داریم ولی به صورت بالقوه امکان ارتباط از هر سمت به سمت دیگر وجود داره ولی نه همزمان و حالت full.. توی این حالت این ارتباط می تونه از هر سمت به سمت دیگر باشه و همزمان هم باشه

Asynchronous Serial Data Communication

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing*. In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low), and the stop bit(s) is 1 (high).



Notice in the Figure that when there is no transfer, the signal is 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit (space) followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

ارتباط asynchronous:
بخاطر اینکه ما بیایم این ارسال داده هارو به صورت دقیقی کنترل بکنیم یکسری قواعدی وجود داره که شروع ارسال و اتمام ارسال رو برای ما مشخص میکنه برای این کار توی ارتباط asynchronous که به صورت وسیعی این ها برای ارسال هایی که مبتنی بر کارکتر هستن استفاده میشه میاد و یکسری بیت های اضافه تری به اون پیام اصلی ما اضافه میشه که یک بیت ابتدا قرار میگیره و یک بیت انتها که این هارو تحت عنوان start bit , stop bit می شناسه و به این کار کلا framing میگیم ینی ما برای اینکه به درستی تشخیص بدیم اونور کجا شروع شده و کجا این ارسال ما تموم میشه یکسری بیت اضافه میکنیم و قبل از این هم خط ما به صورت high است که به این میگیم mark میگیم ینی به خط به صورت اولیه توی وضعیت high قرار داده و وقتی که میخواد ارتباطش رو شروع بکنه یک بیت صفر یا space می فرسته و بعد حالا اون متن اصلی رو از کم ارزش ترین بیت شروع میکنه به ارسال کردن و وقتی که پیامش تموم شد یک بیت stop می فرسته که این معمولا یک هم هستش و بعد خط توی اون حالت نرمال خودش قرار میگیره تا اینکه بخواد اون داده مورد نظر خودش رو ارسال بکنه
نکته: این 8 بیتی که میخوایم بفرستیم از اون بیت کم ارزش تر شروع میکنه به ارسال شدن

Asynchronous Serial Data Communication

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, 8-bit data has become common due to the extended ASCII characters. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs, however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, each 8-bit character has an extra 2 bits, which gives 25% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd parity bit the number of 1s in the data bits, including the parity bit, is odd. Similarly, in an even parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

این بیت های اضافه تری که ما داریم اضافه میکنیم باعث میشه که یکسری overhead به ما اضافه میکنه این overhead تقریبا چیزی نزدیک به 25 درصد است که میتونه overhead قابل توجهی هم باشه
در کنار این بیت هایی که گفتیم ما یکسری بیت های دیگه هم داریم تحت عنوان parity که برای چک کردن صحت ارسال و درستی پیام انجام میشه که به صورت parity زوج و فرد هستش --> توی حالت parity زوج تعداد یک ها با احتساب اون بیت parity که اضافه میشه باید زوج باشه و توی حالت فرد تعداد بیت های یکی که توی اون بسته وجود داره با احتساب بیت parity باید فرد باشه
خود این بایتی هم که می خوایم ارسال بکنیم می تونه 7 یا 8 بیت باشه که این برمیگرده یه مقدار به جنبه های تاریخی که میگه کد های اسکی توی نسخه اولیه 7 بیت بود و بعد که توسعه یافت شد 8 بیت پس این قابلیت ها وجود داره که تعداد بیت ها هم در تراشه های مختلفی که وجود داشته تنظیم بشه

Data Transfer Rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not necessarily equal. This is because baud rate is the modem terminology and is defined as the number of signal changes per second. In modems, sometimes a single change of signal transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K. Notice that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

در ارتباط سریال ما یک نرخ ارسال داده داریم که این معمولا با bits per second مشخص میشه و یک واژه دیگری هم وجود داره برای این مشخص کردن نرخ ارسال داده تحت عنوان baud rate که این برمی گرده به استفاده اون در مودم ها که اونجا نرخ سیگنال رو مشخص می کرده ولی توی این کارهایی که ما داریم انجام میدیم ما معمولا این ها رو با هم برابر در نظر میگیریم و این نرخ رو می تونیم به صورت baud rate هم بیان بکنیم
توی سیستم های قدیمی این baud rate چیزی بین 100 تا 9600 بوده ولی توی سیستم های جدید این نرخ ارسال به 56k هم می رسه که البته این با یکسری تکنیک هایی انجام میشه ولی اون baud rate که ما درباره اش صحبت کردیم نهایت به 100000bps محدود میشه

RS232 Standards

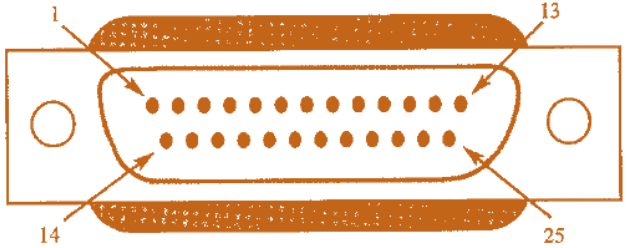
To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969,respectively.

RS232 is widely used serial I/O interfacing standards. This standard is used in PCs and numerous types of equipment. Because the standard was set long before the advent of the TTL logic family, however, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by –3 to –25 V, while a 0 bit is +3 to +25 volts, making –3 to +3 undefined. For this reason, to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage levels, and vice versa. MAX232 IC chips are commonly referred to as line drivers.

برای اینکه ابزارهای مختلفی که میخوان به صورت سریال با هم ارتباط برقرار کنن باید یکسری استانداردهایی رو رعایت بکنن تا حرف هم دیگر رو متوجه بشن
اما در حوزه ارتباط سریال ما یک استاندارد تحت عنوان RS232: یکسری استانداردهایی در خصوص ارتباط سریال مطرح میکنه که این یک استانداردی است که به صورت وسیعی در ارتباطات سریال داره استفاده میشه
با توجه به اینکه اون زمان TTL خیلی مطرح نشده بود این اون سطوح ولتاژی که در TTL وجود داشت رو پشتیبانی نمیکنه برای همین وقتی ما میخوایم ارتباط رو به صورت سریال برقرار بکنیم نیاز داریم یکسری درایورهایی اینجا اضافه بکنیم تا این سطوح ولتاژ رو برای ما تطبیق بده تا بتونیم در این استاندارد امکان ارتباط داشته باشیم یکی ازاین درایور ها تراشه ای هستش تحت عنوان MAX232 که این سطوح ولتاژ رو تطابق میده برای اینکه میکروکنترلر ما بتونه در قالب ارتباط سریال و برمبنای استاندارد RS232 ارتباط خودش رو برقرار بکنه

RS232 Pins

The Table shows the pins for the original RS232 cable and their labels, commonly referred to as the DB-25 connector. In labeling, DB-25P refers to the plug connector (male), and DB-25S is for the socket connector (female).

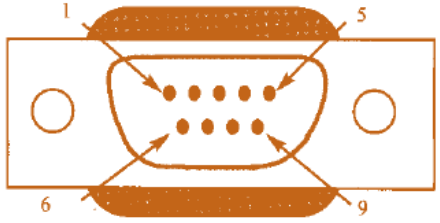


The Original RS232 Connector DB-25 (No longer in use)

Because not all the pins were used in PC cables, IBM introduced the DB-9 version of the serial I/O standard, which uses only 9 pins, as shown in

IBM PC DB-9 Signals

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)



9-Pin Connector for DB-9

RS232 Pins (DB-25)

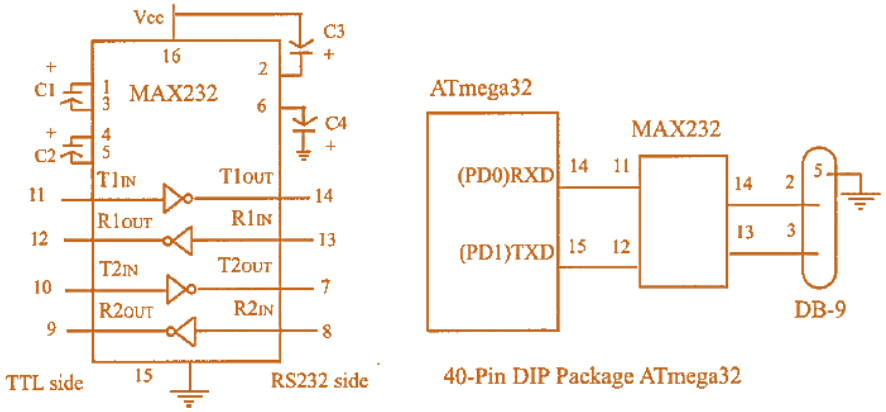
Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send (RTS)
5	Clear to send (CTS)
6	Data set ready (DSR)
7	Signal ground (GND)
8	Data carrier detect (DCD)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready (DTR)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

توی RS232 توی نسخه اولیه یک همچین connector داشتیم مثل شکل روبه رو که به DB-25 شناخته میشد ینی 25 پایه داشت که عملکرد پایه هاشو توی جدول وارده این connector تقریبا الان دیگه استفاده نمیشه
و الان یک connector دیگه ای جایگزین اون شد تحت عنوان DB-9 که 9 تا پایه داره

X86 PC COM Ports

The x86 PCs (based on 8086, 286, 386, 486, and all Pentium microprocessors) used to have two COM ports. Both COM ports were RS232-type connectors. The COM ports were designated as COM 1 and COM 2. In recent years, one of these has been replaced with the USB port, and COM 1 is the only serial port available, if any. We can connect the AVR serial port to the COM 1 port of a PC for serial communication experiments. In the absence of a COM port, we can use a COM-to-USB converter module.

The ATmega32 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TX and RX and are part of the Port D group (PD0 and PD1) of the 40-pin package. Pin 15 of the ATmega32 is assigned to TX and pin 14 is designated as RX. These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible. One such line driver is the MAX232 chip.



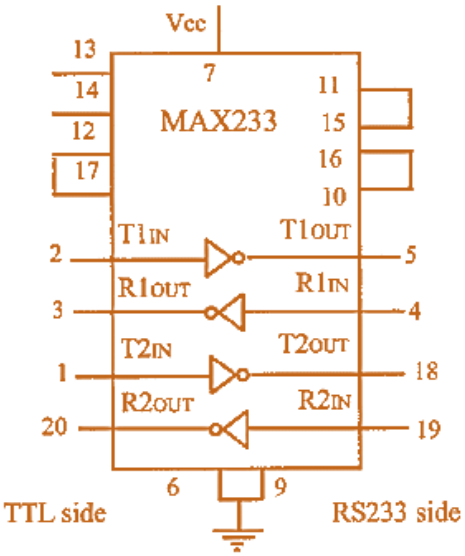
توی کامپیوترهای قدیمی ما دوتا پورت COM داشتیم ک این ها اون امکان ارتباط برمبنای استاندارد RS232 در اختیار ما قرار میدادن ولی الان کم کم تعداد این پورت ها کم شد الان توی اکثر سیستم های امروزی تقریبا پورت COM دیگه ما نمی بینیم و توی لپ تاپ ها هم اصلا وجود نداره الان که اینجا برای اینکه بتونیم اون ارتباط سریال رو داشته باشیم می تونیم از مبدل های COM-to-USB استفاده بکنیم --> ینی این ور لپ تاپ و سیستم رو به USB وصل میکنیم ولی اونور می تونیم اونو متصل بکنیم به IC MAX232 و اون ارتباطات برمبنای RS232 یا پورت های COM دیگری

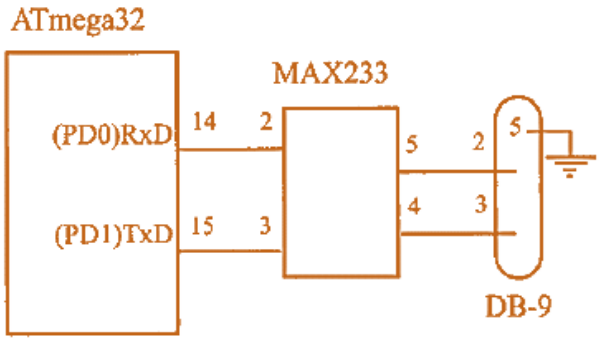
توی ATmega32 ما دوتا پایه توی پورت D داریم ینی پین شماره 0 و 1 پورت D که این ها امکان ارتباط سریال رو برای ما فراهم میکنه --> شکل سمت راست رو ببین

Line Driver (MAX232, MAX233)

MAX232 requires four capacitors ranging from 0.1 to 22 μF. The most widely used value for these capacitors is 22 μF.

To save board space, some designers use the MAX233 chip from Maxim. The MAX233 performs the same job as the MAX232 but eliminates the need for capacitors. However, the MAX233 chip is much more expensive than the MAX232. Notice that MAX233 and MAX232 are not pin compatible. You cannot take a MAX232 out of a board and replace it with a MAX233.





40-Pin DIP Package ATmega32

برای استفاده از MAX232 ما به چندتا خازن اینجا نیاز داریم که این ها یک رنجی بین 0.1 تا 22 میکروفاراد ظرفیت داره
اما وجود این خازن ها می تونه یک مقدار توی بعضی از طراحی ها دست و پا گیر باشه و برای اینکه بتونن توی فضای برد صرفه جویی بکنن معمولا توی جاهایی که فضا مهمه میاین از یک تراشه مشابه دیگری به اسم MAX233 استفاده می کنن که عملکرد این تراشه هم مثل همون 232 هستش منتها اینجا ما به خازن دیگه نیازی نداریم
این تراشه یک مقداری گرون تر است برای همین توی کارهای عادی خیلی از اون استفاده ای نمی کنن

نکته: اگر از MAX232 داریم استفاده میکنیم چینش پایه هاش به نحوی هست که نمی تونیم به راحتی اونو برداریم و MAX233 رو جایگزین کنیم و اینو باید از ابتدای طراحی توی اون ملاحظاتی که داریم ببینیم

AVR Serial Port Programming

The USART (universal synchronous asynchronous receiver/ transmitter) in the AVR has normal asynchronous, double-speed asynchronous, master synchronous, and slave synchronous mode features.

In he AVR microcontroller five registers are associated with the USART that we deal with in this part. They are UDR (USART Data Register), UCSRA, UCSRB, UCSRC (USART Control Status Register), and UBRR (USART Baud Rate Register).

به کار گیری تراشه USART:
از USART در میکروکنترلر های AVR می تونیم در یکی از این مدهای نرمال، double-speed و master و slave استفاده بکنیم
به صورت کلی برای اینکه بیایم و USART رو تنظیم بکنیم و کنترل هایی که در اختیار ما قرار میده متناسب با نیازهایی خودمون تنظیم بکنیم یکسری رجیسترهایی وجود داره مثل رجیسترهای کنترلی که در خصوص تایمرها داشتیم اینجا هم وجود داره و کافیه این ها رو بشناسیم و.. و به کار بگیریم

5 رجیستر برای تنظیمات USART وجود داره تحت عنوان: UCSRB , UCSRA , UDR , UCSRC , UBRR

UBRR Register and Baud Rate in the AVR

AVR transfers and receives data serially at many different baud rates. The baud rate in the AVR is programmable. This is done with the help of the 8-bit register called UBRR. For a given crystal frequency, the value loaded into the UBRR decides the baud rate. The relation between the value loaded into UBRR and Fosc is dictated by the following formula:

Desired Baud Rate = Fosc/ (16(X + 1))

UBRR Values for Various Baud Rates

Baud Rate	UBRR (Decimal Value)
38400	12
19200	25
9600	51
4800	103
2400	207
1200	415

Note: For Fosc = 8 MHz

Assuming that Fosc = 8 MHz, we have the following:

Desired Baud Rate = Fosc/ (16(X + 1)) = 8 MHz/16(X + 1) = 500 kHz/(X + 1)

X = (500 kHz/ Desired Baud Rate) – 1

رجیستر UBRR: امکان تنظیمات مرتبط با baud rate رو در اختیار ما قرارمیده این رجیستر این انتقال میتونه در baud rate های متفاوتی صورت بگیره AVR این امکان رو برای ما فراهم اورده که بتونیم baud rate رو تنظیم بکنیم این کار از طریق یک رجیستر 8 بیتی تحت عنوان UBRR انجام میشه و baud rate ما اینجا متناسب با اون فرکانس کاری که میکرو داره و اعمال کردیم به سیستم و مقداری که توی UBRR لود کردیم میاد و طبق فرمولی که گفته محاسبه میشه

مثال: اگر فرکانس کاری ما 8 مگاهرتز هستش --> baud rate ؟؟

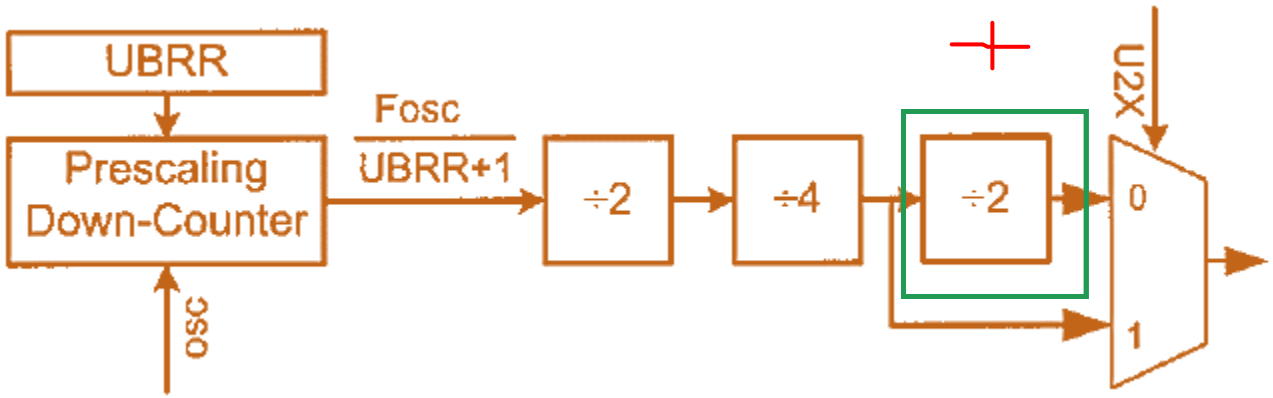
برای اینکه یک baud rate خاصی رو به دست بیاریم به چه فرکانس و مقداری نیاز داریم که توی این رجیستر بارگذاری بکنیم می تونیم از همین رابطه استفاده بکنیم مثلا توی همین مثال می تونیم X رو به دست بیاریم و بعد اونو توی رجیستر UBRRمون بارگذاری بکنیم

جدولی که توی اسلاید هست با این فرض به دست اومده که فرکانس کاری ما 8 مگاهرتز هستش مثلا برای اینکه baud rate --> 38400 داشته باشیم کافیه که عدد 12 رو به صورت دسیمال توی رجیستر UBRR بارگذاری بکینم
نکته: هرچی baud rate ما داره کم میشه این عدد داره بزرگتر میشه یک داستانی پشت این هست که توی صفحه بعدی...

UBRR Register and Baud Rate in the AVR

The UBRR is connected to a down-counter, which functions as a programmable prescaler to generate baud rate. The system clock (Fosc) is the clock input to the down-counter. The down-counter is loaded with the UBRR value each time it counts down to zero. When the counter reaches zero, a clock is generated. This makes a frequency divider that divides the OSC frequency by UBRR + 1. Then the frequency is divided by 2, 4, and 2.

we can choose to bypass the last divider and double the baud rate.



UBRR متصل است به یک شمارنده که این شمارنده به صورت کاهشی میاد و شمارش میکنه و باعث میشه که اون baud rate ما تنظیم بشه به چه صورت؟ این هر موقع که شمارش رو انجام میده و به صفر میرسه از طریق یکسری مدارات دیگه باعث تولید یک کلاک میشه که این کلاک کاری باعث انتقال بیت ها خواهد شد
شکل:
یک شمارنده داریم و مقدار از توی رجیستر UBRR بارگذاری میشه توی Prescaling.. و شروع میکنه به شمارش تا به صفر برسه و خود این prescaling.. وصل میشه به یکسری تقسیم کننده های دیگری

Example

With Fosc = 8 MHz, find the UBRR value needed to have the following baud rates:
(a) 9600 (b) 4800 (c) 2400 (d) 1200

Fosc = 8 MHz => X = (8 MHz/16(Desired Baud Rate)) – 1
=> X = (500 kHz/(Desired Baud Rate)) – 1

(a) (500 kHz/ 9600) – 1 = 52.08 – 1 = 51.08 = 51 = 33 (hex) is loaded into UBRR
(b) (500 kHz/ 4800) – 1 = 104.16 – 1 = 103.16 = 103 = 67 (hex) is loaded into UBRR
(c) (500 kHz/ 2400) – 1 = 208.33 – 1 = 207.33 = 207 = CF (hex) is loaded into UBRR
(d) (500 kHz/ 1200) – 1 = 416.66 – 1 = 415.66 = 415 = 19F (hex) is loaded into UBRR

مثال: فرض میکنیم فرکانس ما 8 مگاهرتز است میخوایم ببینیم چه عددی رو باید داخل رجیستر UBRR بارگذاری بکنیم تا این baud rate هایی که نوشته رو به دست بیاریم

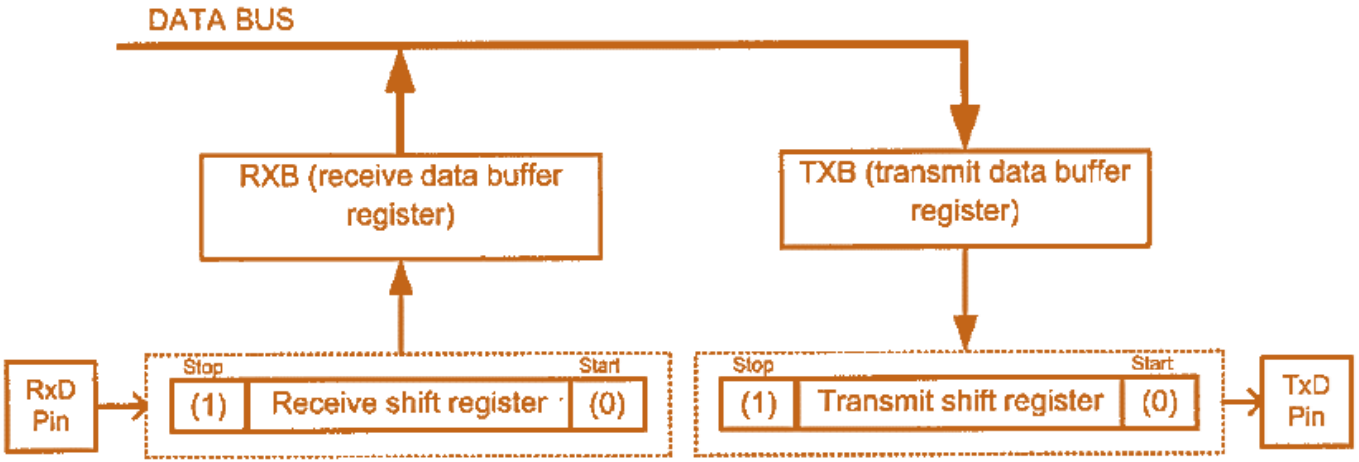
UBRR Register

As you see in the Figure , UBRR is a 16-bit register but only 12 bits of it are used to set the USART baud rate. Bit 15 is URSEL and, as we will see in the next section, selects between accessing the UBRRH or the UCSRC register. The other bits are reserved.

خود رجیستر UBRR یک رجیستر 16 بیتی هستش که فقط 12 بیت اون برای کنترل های مرتبط با USART استفاده شده و بیت 15 هم داره که امکان انتخاب بین رجیستر UBRRH,UCSRC برای ما فراهم میاره و مابقی بیت ها هم به صورت رزرو نگه داشته شده

UDR Registers and USART Data IO in the AVR

In the AVR, to provide a full-duplex serial communication, there are two shift registers referred to as *Transmit Shift Register* and *Receive Shift Register*. Each shift register has a buffer that is connected to it directly. These buffers are called *Transmit Data Buffer Register* and *Receive Data Buffer Register*. The USART Transmit Data Buffer Register and USART Receive Data Buffer Register share the same I/O address, which is called *USART Data Register* or *UDR*. When you write data to UDR, it will be transferred to the Transmit Data Buffer Register (TXB), and when you read data from UDR, it will return the contents of the Receive Data Buffer Register (RXB).



رجیستر UDR:
این رجیستر قراره یکسری داده رو برای ما توی انتقال های USART نگه داره
ساختار کلیش رو پایین کشیده:
یک شیفت رجیستر داریم که به صورت سریال توی قسمت receiveاش این داده رو دریافت می کنه و بعد به صورت موازی منتقل می کنه روی باس ما و در خصوص ارسال هم ینی transmit ما داده رو به صورت موازی دریافت میکنیم و به صورت سریال اینو ارسال می کنیم
برای اینکه بتونیم با این شیفت رجیسترها به درستی کار بکنیم یکسری بافرهایی قبل از این ها قرار داره که به این بافرها transmit data buffer , receive data buffer می شناسیم --> اینها در AVR و در استفاده از USART یک رجیستری رو تشکیل میده تحت عنوان UDR
UDR در فضای I/O ما یک ادرس برای این دوتا رجیستر ینی TXB , RXB داره ینی در حقیقت ما به این دوتا رجیستر می گیم UDR و با یک ادرس می تونیم اون ها رو ادرس دهی بکنیم
وقتی که قراره یک داده ای رو توی UDR بفرستیم و این از طریق واسط ما انتقال داده بشه میاد توی transmit data buffer ریخته میشه و وقتی که قراره یک داده ای رو بخونیم این داده میاد از قسمت receive data buffer خونده میشه --> پس این ها دو رجیستر مجزا هستند ولی یک ادرس دارند و به صورت اتوماتیک وقتی که میخوایم بنویسیم توی رجیستر TXB انجام میشه و خوندن توی رجیستر RXB

UCSR Registers and USART Configuration in the AVR

- UCSRs (USART Control Status Registers)
  - are 8-bit control registers
    - UCSRA
    - UCSRB
    - UCSRC
  - Used for controlling serial communication in AVR

رجیستر UCSR:
از طریق این رجیسترها ما می تونیم تنظیمات مرتبط با ارتباط سریال رو در تراشه AVR انجام بدیم
3 رجیستر 8 بیتی تحت این عنوان برای کنترل ارتباط سریال دیده شده:
UCSRA
UCSRB
UCSRC

UCSRA

RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM

RXC (Bit 7): USART Receive Complete
This flag bit is set when there are new data in the receive buffer that are not read yet. It is cleared when the receive buffer is empty. It also can be used to generate a receive complete interrupt.

TXC (Bit 6): USART Transmit Complete
This flag bit is set when the entire frame in the transmit shift register has been transmitted and there are no new data available in the transmit data buffer register (TXB). It can be cleared by writing a one to its bit location. Also it is automatically cleared when a transmit complete interrupt is executed. It can be used to generate a transmit complete interrupt.

UDRE (Bit 5): USART Data Register Empty
This flag is set when the transmit data buffer is empty and it is ready to receive new data. If this bit is cleared you should not write to UDR because it overrides your last data. The UDRE flag can generate a data register empty interrupt.

FE (Bit 4): Frame Error
This bit is set if a frame error has occurred in receiving the next character in the receive buffer. A frame error is detected when the first stop bit of the next character in the receive buffer is zero.

DOR (Bit 3): Data OverRun
This bit is set if a data overrun is detected. A data overrun occurs when the receive data buffer and receive shift register are full, and a new start bit is detected.

PE (Bit 2): Parity Error
This bit is set if parity checking was enabled (UPM1 = 1) and the next character in the receive buffer had a parity error when received.

U2X (Bit 1): Double the USART Transmission Speed
Setting this bit will double the transfer rate for asynchronous communication.

MPCM (Bit 0): Multi-processor Communication Mode
This bit enables the multi-processor communication mode.

Notice that FE, DOR, and PE are valid until the receive buffer (UDR) is read. Always set these bits to zero when writing to UCSRA.

:UCSRA
این رجیستر 8 بیتی هستش
بیت 7 اون زمانی استفاده میشه که اون دریافت انتقال تکمیل شده باشه ینی یک فلگی هستش که اگر اون انتقال ما به صورت صحیحی به اتمام برسه این میاد و ست میشه
بیت 6 هم کاربرد مشابهی مثل 7 داره ولی برای زمانی که میخوایم انتقال بدیم
بیت 5 ما تحت عنوان UDRE شناخته میشه و یک فلگی است و زمانی ست میشه که اون transmit data buffer ما خالی باشه (زمانی که میخوایم داده ارسال بکنیم و اگر اون بافر ما خالی نباشه و بیایم درون اون رجیستر UDR داده ای بریزیم که هنوز انتقال اون اتمام پیدا نکرده این باعث میشه که اخرین دادمون رو تخریب بکنیم و به نوعی از دستش بدیم برای همین ما معمولا میایم این فلگ رو چک میکنیم تا زمانی که اون USART بیاد و داده قبلی ما رو انتقال بده و زمانی که این ست شد ینی بافر خالیه و ما می تونیم داده جدیدی رو ارسال بکنیم)
بیت 4 فلگی است تحت عنوان frame error و از اون بیت هایی کنترلی که نشون میده انتقال ما به صورت صحیحی انجام شده و این بیت در کنار بیت شماره 2 تحت عنوان parity error بیت های کنترلی هستن که صحت ارسال و دریافت داده رو دارن چک میکنن
بیت 3 یا DOR : این چک میکنه که overrun اتفاقی نیوفته
بیت 1: در ارتباطات asynchronous این امکان رو برای ما فراهم میاره که نرخ انتقال رو به 2 برابر افزایش بدیم --> توی صفحه 16 قسمت + : زمانی که این بیت فعال بشه دیوایدر اخر ینی + از بین می ره و عملا نرخ ارسال داده به 2 برابر افزایش پیدا خواهد کرد
بیت 0: میاد و ارتباط رو توی حالت مالتی پروسسور قرار میده

نکته: بیت های FE , PE , DOR تا زمانی که بافر دریافت رو نخونیم این ها معتبر هستن

UCSRB

RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

RXCIE (Bit 7): Receive Complete Interrupt Enable
To enable the interrupt on the RXC flag in UCSRA you should set this bit to one.

TXCIE (Bit 6): Transmit Complete Interrupt Enable
To enable the interrupt on the TXC flag in UCSRA you should set this bit to one.

UDRIE (Bit 5): USART Data Register Empty Interrupt Enable
To enable the interrupt on the UDRE flag in UCSRA you should set this bit to one.

RXEN (Bit 4): Receive Enable
To enable the USART receiver you should set this bit to one.

TXEN (Bit 3): Transmit Enable
To enable the USART transmitter you should set this bit to one.

UCSZ2 (Bit 2): Character Size
This bit combined with the UCSZ1:0 bits in UCSRC sets the number of data bits (character size) in a frame.

RXB8 (Bit 1): Receive data bit 8
This is the ninth data bit of the received character when using serial frames with nine data bits.

TXB8 (Bit 0): Transmit data bit 8
This is the ninth data bit of the transmitted character when using serial frames with nine data bits.

:UCSRB
یک رجیستر 8 بیتی است
بیت 7: برای فعال سازی وقفه مرتبط با دریافت استفاده میشه و اون فلگ PXC در رجیستر UCSRA تکمیل میشه میاد و یک وقفه صادر میکنه و ما می تونیم اون بحث پولینگ رو کنار بذاریم و این انتقال رو به صورت وقفه انجام بدیم
بیت 6: وقفه رو برای ارسال فعال میکنه و زمانی که ما میخوایم ارسال رو انجام بدیم از طریق این وقفه می تونیم اون سرکشی هایی که توی حالت پولینگ قراره انجام بدیم رو کنار بذاریم
بیت 5: توی صفحه قبل گفتیم یک بیت داریم تحت عنوان UDRE و اینو مرتب چک میکنیم که نیایم داده قبلی رو تخریب بکنیم و این وقفه مرتبط با همون قضیه رو برای ما فعال میکنه
بیت 4 و 3 : تنظیم می کنن USART ما قابلیت دریافت داشته باشم یا قابلیت ارسال داشته باشه
بیت 2: توی AVR این امکان وجود داره که ما مشخص بکنیم سایز اون داده ای که می خوایم ارسال بکنیم چقدر باشه --> سه بیت اینجا ما برای تعیین قضیه داریم یکی از این بیت ها تحت عنوان UCSZ2 توی این رجیستر 8 بیتی قراره داره و دو بیت دیگه اش توی رجیستر UCSRC هست
بیت1 و 0: زمانی که سایز اون کارکتری که میخوایم ارسال بکنیم 9 باشه یک بیت اضافه تر نیاز داریم که توی زمان انتقال ارسال و دریافت ازش کمک بگیریم

UCSRC

URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

URSEL (Bit 7): Register Select
This bit selects to access either the UCSRC or the UBRRH register

UMSEL (Bit 6): USART Mode Select
This bit selects to operate in either the asynchronous or synchronous mode of operation.
0 = Asynchronous operation
1 = Synchronous operation

UPM1:0 (Bit 5:4): Parity Mode
These bits disable or enable and set the type of parity generation and check.
00 = Disabled
01 = Reserved
10 = Even Parity
11 = Odd Parity

USBS (Bit 3): Stop Bit Select
This bit selects the number of stop bits to be transmitted.
0 = 1 bit
1 = 2 bits

UCSZ1:0 (Bit 2:1): Character Size
These bits combined with the UCSZ2 bit in UCSRB set the character size in a frame

UCPOL (Bit 0): Clock Polarity
This bit is used for synchronous mode only

UCSRC:
یک رجیستر 8 بیتی داریم
بیت 7: برای انتخاب بین UBRRH , UCSRC است --> این دو رجیستر یک ادرس مشترک در فضای I/O دارن و با این بیت ما انتخاب می کنیم که کدوم یکی از این دو رجیستر الان مد نظر ما هستش
اگر صفر باشه --> رجیستر UBRRH انتخاب میشه
اگر یک باشه --> UCSRC انتخاب میشه
بیت 6: برای انتخاب مد USART استفاده میشه:
اگر صفر باشه توی حالت asynchronous داریم عمل میکینم و اگر یک باشه به صورت synchronous عمل می کنه
بیت 4 و 5:
اگر 00 بود : parity ما غیر فعال میشه
01 : به صورت رزرو در نظر گرفته شده
10: میاد parity رو به صورت زوج در نظر میگیره
11: میاد parity رو به صورت فرد در نظر می گیره
بیت 3: انتخاب میکنه که الان توی نحوی ارسالی که ما در نظر گرفتیم یک بیت stop اضافه بشه به اون داده اصلی ما یا اینکه دو بیت stop:
0--> یک بیت
1--> دو بیت
بیت 2 و 1: سایز کارکتر
بیت 0: توی مد synchronous مورد استفاده قرار میگیره:
اگر یک باشد ینی لبه بالا رونده
اگر صفر باشد ینی لبه پایین رونده

Character Size

To set the number of data bits (character size) in a frame you must set the values of the UCSZ1 and USCZ0 bits in the UCSRB and UCSZ2 bits in UCSRC. The Table shows the values of UCSZ2, UCSZ1, and UCSZ0 for different character sizes.

Values of UCSZ2:0 for Different Character Sizes

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	1	1	9

Because of some technical considerations, the UCSRC register shares the same I/O location as the UBRRH, and therefore some care must be taken when accessing these I/O locations. When you write to UCSRC or UBRRH, the high bit of the written value (URSEL) controls which of the two registers will be the target of the write operation. If URSEL is zero during a write operation, the UBRRH value will be updated; otherwise, UCSRC will be updated.

سه بیت ما اینجا داریم:
یک بیتش در رجیستر UCSRB هستش و دو بیت دیگه توی رجیستر UCSRC هستن
این سه بیت تنظیم می کنه که سایز Character ما چی باشه
این سایز می تونه 5 و 6 و 7 و 8 و 9 باشه که عموما ما توی سایز 8 با اینها کار می کنیم
زمانی که حداکثر 8 هست ینی 5 و 6 و 7 و 8 همون رجیستر UDR جوابگوی این انتقالات هستش ولی اگر این به 9 بیت افزایش پیدا بکنه اون موقع با توجه به اینکه رجیستر UDR ما 8 بیتی هستش یک بیت کمکی نیاز داریم

Example

(a) What are the values of UCSRB and UCSRC needed to configure USART for asynchronous operating mode, 8 data bits (character size), no parity, and 1 stop bit? Enable both receive and transmit.

(b) Write a program for the AVR to set the values of UCSRB and UCSRC for this configuration.

(a) RXEN and TXEN have to be 1 to enable receive and transmit. UCSZ2:0 should be 011 for 8-bit data, UMSEL should be 0 for asynchronous operating mode, UPM1:0 have to be 00 for no parity, and USBS should be 0 for one stop bit.

(b)

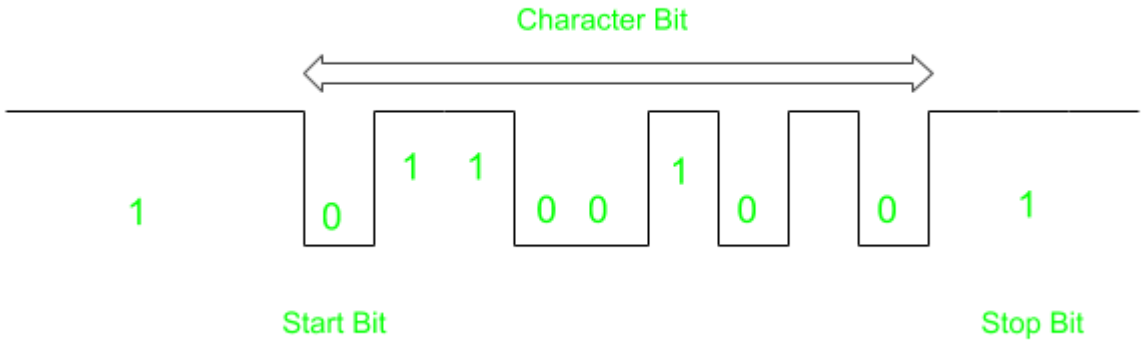
```
.INCLUDE "M32DEF.INC"

     LDI   R16,(1<<RXEN)|(1<<TXEN)
     OUT   UCSRB, R16
;In the next line URSEL = 1 to access UCSRC. Note that instead
;of using shift operator, you can write "LDI R16, 0b10000110"
     LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL)
     OUT   UCSRC, R16
```

مثال:
می خوایم USART رو به نحوی تنظیم بکنیم که در مد asynchronous عمل بکنه و کارکتر سایزش 8 بیت باشه و یک بیت stop داشته باشیم و همزمان می خوایم هم امکان دریافت و هم امکان ارسال فعال باشه --> رجیستر UCSRC , UCSRB هم اینجا داریم : الف
ب: کدش

الف) با توجه به جدول هایی که توی این اسلاید داشتیم می بینیم برای اینکه بیایم فعال بکنیم هم دریافت و هم ارسال رو کافیه که دو بیت PXEN , TXEN رو یک بکنیم
و چون داده 8 بیت هست باید اون بیت های مرتبط با سایز رو تنظیم بکنیم طبق صفحه قبلی
و توی مد asynchronous هم میخوایم باشیم پس باید UMSEL رو هم صفر بذاریم و چون parity هم نداریم 00 رو برای بیت های parity باید انتخاب بکنیم
و توی قسمت stop bit هم چون خواسته شده که یک بیت داشته باشیم کافیه که USBS رو صفر بذاریم

FE and PE Flag Bits

When the AVR USART receives a byte, we can check the parity bit and stop bit. If the parity bit is not correct, the AVR will set PE to one, indicating that an parity error has occurred. We can also check the stop bit. As we mentioned before, the stop bit must be one, otherwise the AVR would generate a stop bit error and set the FE flag bit to one, indicating that a stop bit error has occurred. We can check these flags to see if the received data is valid and correct. Notice that FE and PE are valid until the receive buffer (UDR) is read. So we have to read FE and PE bits before reading UDR.



این دو بیت برای بررسی صحت ارسال انتقال داده مورد استفاده قرار میگیره
اگر فلگ FE فعال باشه ینی ما یک خطایی توی اون بیت stop داشتیم
اگر فلگ PE فعال باشه ینی داده های ما تخریب شده

Programming the AVR to transfer data Serially

In programming the AVR to transfer character bytes serially, the following steps must be taken:

1. The UCSRB register is loaded with the value 08H, enabling the USART transmitter. The transmitter will override normal port operation for the TxD pin when enabled.
2. The UCSRC register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR is loaded with one of the values in the Table (if Fosc = 8 MHz) to set the baud rate for serial data transfer.
4. The character byte to be transmitted serially is written into the UDR register.
5. Monitor the UDRE bit of the UCSRA register to make sure UDR is ready for the next byte.
6. To transmit the next character, go to Step 4.

چجوری AVR رو تنظیم بکنیم که برای ما یک داده رو به صورت سریال ارسال بکنه؟
برای اینکه بیایم و AVR رو تنظیم بکنیم به این صورت می ریم سراغ رجیستر UCSRB و فرض می کنیم که می خوایم این قضیه به صورت ارسال باشه
1- 08 هگز کافیه توی رجیستر UCSRB بریزیم و این باعث میشه که اون پایه های مرتبط با TxD که روی تراشه در نظر گرفته این ها برای انتقال تنظیم بشه
2- رجیستر UCSRC با فرض اینکه بخوایم به صورت asynchronous از USARTمون استفاده بکنیم و در سایز 8 بیت و parity هم در نظر گرفته نشده و بیت stop هم یک در نظر می گیریم --> کافیه که 06 هگز رو برای انجام این تنظیمات توی UCSRC بریزیم
3- رجیستر UBRR : تنظیم میکنیم با توجه به اون فرکانسی که داریم و baud rateمون
4- داده ای که قراره ارسال بکنیم رو روی UDR قرار میدیم
5- چک میکنیم اون داده ای که قرار دادیم --> اول ببینیم اون داده های قبلی که داریم ارسالشون تموم شده باشه تا داده های قبلی رو تخریب نکنیم و زمانی که این داده رو می فرستیم برای ارسال داده بعدی باز نیازه که چک بکنیم که ارسال داده جاری هم تموم شده باشه
6- و بعد دوباره این مراحل رو تکرار میکینم ینی داده جاری رو که فرستادیم برمیگردیم به مرحله 4 و داده جدید رو توی رجیستر UDR قرار میدیم به شرط اینکه همین کنترل هایی که دربارشون صحبت کردیم رو چک کرده باشیم

Example

Write a program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz.

```
.INCLUDE "M32DEF.INC"
      LDI   R16,(1<<TXEN)                        ;enable transmitter
      OUT   UCSRB, R16
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);8-bit data
      OUT   UCSRC, R16                           ;no parity, 1 stop bit
      LDI   R16,0x33                             ;9600 baud rate
      OUT   UBRRL,R16                            ;for XTAL = 8 MHz
AGAIN:
      SBIS  UCSRA,UDRE                           ;is UDR empty
      RJMP  AGAIN                                ;wait more
      LDI   R16,'G'                              ;send 'G'
      OUT   UDR,R16                              ;to UDR
      RJMP  AGAIN                                ;do it again
```

By monitoring the UDRE flag, we make sure that we are not overloading the UDR register. If we write another byte into the UDR register before it is empty, the old byte could be lost before it is transmitted.

مثال:
فرض میکنیم می خوایم کارکتر G رو با 9600 <-- baud rate و فرض اینکه فرکانس ما 8 مگاهرتز هست ارسال بکنیم:

تنظیم میکنیم که اول اون USART ما برای انتقال اماده باشه

Example

Write a program to transmit the message "YES " serially at 9600 baud, 8-bit data, and 1 stop bit. Do this forever.

```
        .INCLUDE "M32DEF.INC"

        LDI   R21,HIGH(RAMEND)          ;initialize high
        OUT   SPH,R21                   ;byte of SP
        LDI   R21,LOW(RAMEND)           ;initialize low
        OUT   SPL,R21                   ;byte of SP

        LDI   R16,(1<<TXEN)             ;enable transmitter
        OUT   UCSRB, R16
        LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); 8-bit data
        OUT   UCSRC, R16                ;no parity, 1 stop bit
        LDI   R16,0x33                  ;9600 baud rate
        OUT   UBRRL,R16
AGAIN:
        LDI   R17,'Y'                   ;move 'Y' to R17
        CALL  TRNSMT                    ;transmit r17 to TxD
        LDI   R17,'E'                   ;move 'E' to R17
        CALL  TRNSMT                    ;transmit r17 to TxD
        LDI   R17,'S'                   ;move 'S' to R17
        CALL  TRNSMT                    ;transmit r17 to TxD
        LDI   R17,' '                   ;move ' ' to R17
        CALL  TRNSMT                    ;transmit space to TxD
        RJMP  AGAIN                     ;do it again
TRNSMT:
        SBIS  UCSRA,UDRE                ;is UDR empty?
        RJMP  TRNSMT                    ;wait more
        OUT   UDR,R17                   ;send R17 to UDR
        RET
```

مثال:
می خوایم پیام YES رو با 9600 <-- baud rate و 8 بیت داده و یک بیت stop ارسال بکنیم

Programming the AVR to receive data Serially

In programming the AVR to receive character bytes serially, the following steps must be taken:

1. The UCSRB register is loaded with the value 10H, enabling the USART receiver. The receiver will override normal port operation for the RxD pin when enabled.
2. The UCSRC register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR is loaded with one of the values in the Table (if Fosc = 8 MHz) to set the baud rate for serial data transfer.
4. The RXC flag bit of the UCSRA register is monitored for a HIGH to see if an entire character has been received yet.
5. When RXC is raised, the UDR register has the byte. Its contents are moved into a safe place.
6. To receive the next character, go to Step 5.

دریافت داده:
1- برای دریافت داده کافیه که اون USART رو به نحوی تنظیم بکنیم که امکان دریافتش فعال بشه --> فرض میکنیم فقط میخواد دریافت رو انجام بده : 10 هگز رو در رجیستر UCSRB قرار میدیم و این باعث میشه که اون پایه RxD رو هم برای دریافت تنظیم میکنیم که فعال بشه
2- همون تنظیمات قبلی که گفتیم اینجا هم هست که عرفش ایناست تقریبا --> حالت asynchronous و یک بیت stop و بدون parity و 8 بیت --> کافیه که 06 هگز رو توی UCSRC قرار بدیم
3- با توجه به baud rate مورد نظر بیایم و محتوای UBRR رو پر بکنیم
4- و مرتب چک میکنیم فلگ PXC رو که دادمون رسیده باشه و زمانی که این فلگ یک بشه ینی داده رسیده و ما می تونیم از اون استفاده بکنیم
5- می ریم داده رو از داخل UDR برمی داریم و توی رجیستر خاص یا هرجایی که مدنظر داریم قرار میدیم تا ازش استفاده بکنیم
6- و برای تکرار این فرایند کافیه که ما همین مرحله 5 رو تکرار بکنیم تا کارکترهای جدیدی رو دریافت بکنیم

Example

Program the ATmega32 to receive bytes of data serially and put them on Port B. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

```
.INCLUDE "M32DEF.INC"
      LDI   R16,(1<<RXEN)                    ;enable receiver
      OUT   UCSRB, R16
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);8-bit data
      OUT   UCSRC, R16                       ;no parity, 1 stop bit
      LDI   R16,0x33                         ;9600 baud rate
      OUT   UBRRL,R16
      LDI   R16,0xFF                         ;Port B is output
      OUT   DDRB,R16
RCVE:
      SBIS  UCSRA,RXC                        ;is any byte in UDR?
      RJMP  RCVE                             ;wait more
      IN    R17,UDR                          ;send UDR to R17
      OUT   PORTB,R17                        ;send R17 to PORTB
      RJMP  RCVE                             ;do it again
```

مثال:
قصد داریم توی تراشه atmega32 یک بایت داده رو به صورت سریال دریافت بکنیم و اونو روی پورت B نمایش بدیم
baud rate رو 9600 در نظر میگیریم و 8 بیت دیتا و یک بیت stop

Example

Write an AVR program with the following parts: (a) send the message "YES" once to the PC screen, (b) get data from switches on Port A and transmit it via the serial port to the PC's screen, and (c) receive any key press sent by HyperTerminal and put it on LEDs. The programs must do parts (b) and (c) repeatedly.

```
        LDI   R21,0x00
        OUT   DDRA,R21                      ;Port A is input
        LDI   R21,0xFF
        OUT   DDRB,R21                      ;Port B is output
        LDI   R21,HIGH(RAMEND)              ;initialize high
        OUT   SPH,R21                       ;byte of SP
        LDI   R21,LOW(RAMEND)               ;initialize low
        OUT   SPL,R21                       ;byte of SP
        LDI   R16,(1<<TXEN)|(1<<RXEN)       ;enable transmitter
        OUT   UCSRB, R16                    ;and receiver
        LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL) ;8-bit data
        OUT   UCSRC, R16                    ;no parity, 1 stop bit
        LDI   R16,0x33                      ;9600 baud rate
        OUT   UBRRL,R16
        LDI   R17,'Y'                       ;move 'Y' to R17
        CALL  TRNSMT                        ;transmit r17 to TxD
        LDI   R17,'E'                       ;move 'E' to R17
        CALL  TRNSMT                        ;transmit r17 to TxD
        LDI   R17,'S'                       ;move 'S' to R17
        CALL  TRNSMT                        ;transmit r17 to TxD
AGAIN:
        SBIS  UCSRA,RXC                     ;is there new data?
        RJMP  SKIP_RX                       ;skip receive cmnds
        IN    R17,UDR                       ;move UDR to R17
        OUT   PORTB,R17                     ;move R17 TO PORTB
SKIP_RX:
        SBIS  UCSRA,UDRE                    ;is UDR empty?
        RJMP  SKIP_TX                       ;skip transmit cmnds
        IN    R17,PINA                      ;move Port A to R17
        OUT   UDR,R17                       ;send R17 to UDR
SKIP_TX:
        RJMP  AGAIN                         ;do it again
TRNSMT:
        SBIS  UCSRA,UDRE                    ;is UDR empty?
        RJMP  TRNSMT                        ;wait more
        OUT   UDR,R17                       ;send R17 to UDR
        RET
```

مثال:
می خوایم یک برنامه ای بنویسیم که ابتدا یک مسیجی رو برای یک مرتبه بیاد و ارسال بکنه روی پورت سریال و اونجا فرض کنید که این پورت سریال وصله به پورت COM یک PC و اونجا ما یک ترمینال داریم و این داده هایی که روی پورت سریال ارسال میکنیم می تونیم اونجا ببینیم
الف: برای بار اول قراره مسیج YES رو ارسال بکنیم و روی کامپیوتر مشاهده بکنیم
ب: از طریق یک سری سوییچ هایی که به پورت A متصل هست یک داده ای رو میخونیم و از طریق پورت سریال ارسال میکنیم و دوباره می خوایم روی اون PC ک داشتیم این داده رو ببینیم
ج: قصد داریم یک داده از طریق اون ترمینالی که روی اون PC ران هستش دریافت بکنیم و روی یک سری LED نمایش بدیم
قسمت ب و ج به صورت تکراری قراره انجام بشه

Doubling the Baud Rate in the AVR

There are two ways to increase the baud rate of data transfer in the AVR:

1. Use a higher-frequency crystal.
2. Change a bit in the UCSRA register, as shown below.

Option 1 is not feasible in many situations because the system crystal is fixed. Therefore, we will explore option 2. There is a software way to double the baud rate of the AVR while the crystal frequency stays the same. This is done with the U2X bit of the UCSRA register. When the AVR is powered up, the U2X bit of the UCSRA register is zero. We can set it to high by software and thereby double the baud rate. If we set the U2X bit to HIGH, the third frequency divider will be bypassed. In the case of XTAL = 8 MHz and U2X bit set to HIGH, we would have:

Desired Baud Rate = Fosc / (8 (X + 1)) = 8 MHz / 8 (X + 1) = 1 MHz / (X + 1)

UBRR Values for Various Baud Rates (XTAL = 8 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**UBR (HEX)**	**UBRR**	**UBR (HEX)**
38400	12	C	25	19
19200	25	19	51	33
9,600	51	33	103	67
4,800	103	67	207	CF
UBRR = (500 kHz / Baud rate) – 1			*UBRR = (1 kHz / Baud rate) – 1*	

یکی ازامکانات خوب USART : وجود یک بیتی هستش تحت عنوان U2X که امکان تغییر در Baud Rate به صورت نرم افزاری در AVR فراهم می کنه
به صورت کلی از دو طریق می تونیم Baud Rate مرتبط با AVR رو کنترل بکنیم:
1- اون فرکانس oscillator که داره سیستم ما رو تحریک میکنه بیاد و تغییر بدیم و متناسب با اون چیزی که نیاز هست اینو تنظیم بکنیم و این نیازمند اینه که ما oscillator متفاوتی جایگزین oscillator فعلی بکنیم
2- بیایم و به صورت نرم افزاری این کار رو بکنیم و این با توجه به وجود بیت U2X در رجیستر UCSRA امکان پذیر است نحوی کار به این صورته : که ما یک بیت داریم تحت عنوان U2X در رجیستر UCSRA و اگر ما اینو یک بکنیم اخرین دیوایدری که در اون سیستم کلاک منتهی به شیفت رجیستر وجود داشت از بین می ره و حذف میشه و کلاک ما سریعتر پالس تولید میکنه و این باعث میشه که نرخ انتقال ما افزایش پیدا کنه --> توی این حالت اون فرمول قبلی تغییر می کنه و اون 16 که اونجا داشتیم اینجا میشه 8
توی جدول این اسلاید:
داریم نرخ Baud Rate و اون محتوایی که قراره بیاد توی رجیستر UBRR بارگذاری بشه رو می بینیم برای دو حالت: حالتی که U2X صفره و حالتی که یکه
با یک شدن دیوایدر اخر حذف میشه
نکته: برای اینکه بخوایم نرخ Baud Rate رو 2 برابر بکنیم می تونیم از بیت U2X استفاده بکنیم و با فعال شدن این بیت این اتفاق می افته

Baud Rate Error Calculation

In calculating the baud rate we have used the integer number for the UBRR register values because AVR microcontrollers can only use integer values. By dropping the decimal portion of the calculated values we run the risk of introducing error into the baud rate. There are several ways to calculate this error. One way would be to use the following formula.

Error = (Calculated value for the UBRR – Integer part) / Integer part

For example, with XTAL = 8 MHz and U2X = 0 we have the following for the 9600 baud rate:

UBRR value = (500,000/ 9600) – 1 = 52.08 – 1 = 51.08 = 51
Error = (51.08 – 51)/ 51 = 0.16%

UBRR Values for Various Baud Rates (XTAL = 8 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**Error**	**UBRR**	**Error**
38400	12	0.2%	25	0.2%
19200	25	0.2%	51	0.2%
9,600	51	0.2%	103	0.2%
4,800	103	0.2%	207	0.2%
UBRR = (500,000 / Baud rate) – 1			*UBRR = (1,000,000 / Baud rate) – 1*	

وقتی ما اومدیم طبق اون فرمولی که داشتیم می خواستیم اون مقدار رو حساب بکنیم معمولا توی محاسباتمون یک مقداری اعشار هم می اوردیم که ما این اعشار رو در نظر نمی گرفتیم و با توجه به این یک خطایی اینجا می تونه اتفاق بیوفته که ناشی از این نادیده گرفتن مقدار اعشاره
روش های مختلفی برای محاسبه این خطا وجود داره:
یکی از این روش ها--> می تونیم اون نرخی که به صورت صحیح داریم در رجیستر UBRRمون قرار میدیم بیام و این رو از مقدار واقعی اش ینی بدون کسر اون قسمت اعشار کم بکنیم و تقسیم بر قسمت صحیحش بکنیم و این یک خطایی به ما میده که میشه همون Error
برای اینکه Baud Rate ما 9600 باشه اون مقداری که قراره توی UBRR بارگذاری بکنیم طبق اون فرمولی که قبلا گفتیم برابره با 51.08 خواهد بود و با توجه به اینکه ما فقط داریم قسمت صحیح اون رو در UBRR هشت بیتی بارگذاری می کنیم ینی فقط 51 رو وارد UBRR میکنیم و اون قسمت اعشارش نادیده گرفته میشه --> پس اینجا تقریبا 0.16 درصد ما خطا داریم

پس انتظار داریم با توجه به اینکه ما داریم اون قسمت اعشاری رو داریم حذف میکنیم و قسمت صحیح رو برای تنظیم Baud Rate در رجیستر UBRR قرار میدیم یک مقدار در اون Baud Rate که انتظاری داریم خطا به وجود میاد

Baud Rate Error Calculation

In some applications we need very accurate baud rate generation. In these cases we can use a 7.3728 MHz or 11.0592 MHz crystal. As you can see in the table , the error is 0% if we use a 7.3728 MHz crystal. In the table there are values of UBRR for different baud rates for U2X = 0 and U2X = 1.

UBRR Values for Various Baud Rates (XTAL = 7.3728 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**Error**	**UBRR**	**Error**
38400	11	0%	23	0%
19200	23	0%	47	0%
9,600	47	0%	95	0%
4,800	95	0%	191	0%
UBRR = (460,800 / Baud rate) – 1			*UBRR = (921,600 / Baud rate) – 1*	

برای زمان هایی که واقعا این دقت مهم است و این مقدار خطا می تونه مشکل ایجاد بکنه توی سیستم ما یکسری راه حل هایی داریم و یکی از این راه حل ها اینه که ما بیایم واقعا از یک oscillator استفاده بکنیم که توی اون تقسیمی که ما داریم توی اون فرمول انجام میدیم این باعث تولید یک عدد صحیحی بشه که اون دقت کار ما رو ببره بالا
مثال:
برای اینکه بیایم و اون خطاها رو کاهش بدیم می تونیم از یک oscillator --> 7.3728 یا یک oscillator --> 11.0592 استفاده بکنیم

توی این اسلاید یک جدول اورده شده که این خطا رو برای oscillator --> 7.3728 حساب کرده --> توی این حالت ما هیچ خطایی نداریم و نزدیک به صفر است پس اون Baud Rate دقیقی که مد نظر بوده تا حد بسیار قابل قبولی می تونیم بهش دست پیدا بکنیم

AVR Serial Port Programming in C

all the special function registers of the AVR are accessible directly in C compilers by using the appropriate header file.

Example

Write a C function to initialize the USART to work at 9600 baud, 8-bit data, and 1 stop bit. Assume XTAL = 8 MHz.

```
void usart_init (void)
{
   UCSRB = (1<<TXEN);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
}
```

در زبان سی:
نکته: special function registers ینی همون رجیسترهایی که ما قراره برای تنظیمات این ها استفاده بکنیم عموما در زبان سی قابل دسترس است و فقط کافیه که ما اون فایل های header متناسب با این ها رو به فایل برناممون اضافه بکنیم

مثال:
تابعی بنویسیم که بیاد از USART با Baud Rate -->9600 استفاده بکنه و اون تنظیماتی که گفتیم عرف است ینی 8 بیت دیتا و یک بیت stop رو هم داشته باشیم و فرکانس کاری ما 8 مگاهرتز است

Example

Write a C program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Use 8-bit data and 1 stop bit. Assume XTAL = 8 MHz.

```
#include <avr/io.h>                          //standard AVR header
void usart_init (void)
{
   UCSRB = (1<<TXEN);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
}
void usart_send (unsigned char ch)
{                                            //wait until UDR
   while (! (UCSRA & (1<<UDRE)));            //is empty
   UDR = ch;                                 //transmit 'G'
}

int main (void)
{
   usart_init();                             //initialize the USART
   while(1)                                  //do forever
     usart_send ('G');                       //transmit 'G' letter
   return 0;
}
```

مثال:
قصد داریم کارکتر G رو به صورت سریال با 9600<-- Baud Rate بفرستیم و اون تنظیماتی که قبلا گفتیم اینجا داریم و نرخ فرکانس هم 8 مگاهرتز است

Example

Write a program to send the message "The Earth is but One Country. " to the serial port continuously. Using the settings in the last example.

```
#include <avr/io.h>                          //standard AVR header

void usart_init (void)
{  UCSRB = (1<<TXEN);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
}
void usart_send (unsigned char ch)
{  while (! (UCSRA & (1<<UDRE)));
   UDR = ch;
}
int main (void)
{  unsigned char str[ 30] = "The Earth is but One Country. ";
   unsigned char strLenght = 30;
   unsigned char i = 0;
   usart_init();
   while(1)
   {
      usart_send(str[ i++] );
      if (i >= strLenght)
        i = 0;
   }
   return 0;
}
```

مثال:
می خوایم یک پیامی رو از طریق پورت سریال با همون تنظیماتی که همیشه درباره اش گفتیم ارسال بکنیم

AVR Serial Port Programming using Interrupts

- Polling TXIF and RXIF is Waste of ‘s μC time
  - Instead, we can use interrupts

- Using interrupts
  - Interrupt Enable bit(s) become HIGH and Force the CPU to Jump to the interrupt service routine
    - Upon completion of receive
    - When UDR is ready to accept new data

روش انتقال داده ها به صورت سریال رو تا اینجا به صورت پولینگ بررسی کردیم ینی مرتب میایم فلگ UDRE رو چک میکنیم که داده ما ارسال شده باشه یا زمانی که داشتیم داده رو دریافت می کردیم این چک رو انجام می دادیم --> اینجوری عملا داریم وقت CPU رو تلف میکنیم برای اینکه این کار به صورت موثرتری انجام بشه ما می تونیم از وقفه استفاده بکنیم

زمانی که USART رو فعال میکنیم که ارسال بکنه و زمانی که ارسالش به اتمام رسید پردازنده رو باخبر می کنه تا پردازنده بتونه مابقی کارهایی که با اون بخش داره رو ادامه بده

کافیه که بیت وقفه مرتبط با این امکان رو فعال بکنیم و به صورت کلی وقفه ها رو فعال بکنیم تا بتونیم زمانی که یک دریافت کامل میشه یا زمانی که اماده ارسال هستیم پردازنده رو باخبر بکنیم که روند کار رو جلو ببره

Example (interrupt-based data receive)

Program the ATmega32 to receive bytes of data serially and put them on Port B. Set the baud rate at 9600, 8-bit data, and 1 stop bit. Use Receive Complete Interrupt instead of the polling method.

```
.INCLUDE "M32DEF.INC"
.CSEG                                   ;put in code segment
     RJMP  MAIN                         ;jump main after reset
.ORG  URXCaddr                          ;int-vector of URXC int.
     RJMP  URXC_INT_HANDLER             ;jump to URXC_INT_HANDLER
.ORG  40                                ;start main after
                                        ;interrupt vector
MAIN: LDI   R16, HIGH(RAMEND)           ;initialize high byte of
      OUT   SPH, R16                    ;stack pointer
      LDI   R16, LOW(RAMEND)            ;initialize low byte of
      OUT   SPL,R16                     ;stack pointer
      LDI   R16,(1<<RXEN)|(1<<RXCIE)    ;enable receiver
      OUT   UCSRB, R16                  ;and RXC interrupt
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);sync,8-bit data
      OUT   UCSRC, R16                  ;no parity, 1 stop bit
      LDI   R16,0x33                    ;9600 baud rate
      OUT   UBRRL,R16
      LDI   R16,0xFF                    ;set Port B as an
      OUT   DDRB,R16                    ;input
      SEI                               ;enable interrupts
WAIT_HERE:
      RJMP  WAIT_HERE                   ;stay here
URXC_INT_HANDLER:
      IN    R17,UDR                     ;send UDR to R17
      OUT   PORTB,R17                   ;send R17 to PORTB
      RETI
```

مثال:
می خوایم از atmega32 استفاده بکنیم و یکسری داده رو به صورت سریال دریافت بکنیم و روی پورت B نمایش بدیم
Baud Rate --> 9600 و داده ها 8 بیتی و یک بیت stop هم داریم

روش کار ما اینجا به صورت وقفه است

Example (interrupt-based data transmit)

Write a program for the AVR to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz. Use interrupts instead of the polling method.

```
.INCLUDE "M32DEF.INC"

.CSEG                                   ;put in code segment
     RJMP  MAIN                         ;jump main after reset
.ORG  UDREaddr                          ;int. vector of UDRE int.
     RJMP  UDRE_INT_HANDLER             ;jump to UDRE_INT_HANDLER
.ORG  40                                ;start main after
                                        ;interrupt vector
;*****************************
MAIN:
     LDI   R16, HIGH(RAMEND)            ;initialize high byte of
     OUT   SPH, R16                     ;stack pointer
     LDI   R16, LOW(RAMEND)             ;initialize low byte of
     OUT   SPL,R16                      ;stack pointer
     LDI   R16,(1<<TXEN)|(1<<UDRIE)     ;enable transmitter
     OUT   UCSRB, R16                   ;and UDRE interrupt
     LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); sync., 8-bit
     OUT   UCSRC, R16                ;data no parity, 1 stop bit
     LDI   R16,0x33                     ;9600 baud rate
     OUT   UBRRL,R16
     SEI                                ;enable interrupts
WAIT_HERE:
     RJMP  WAIT_HERE                    ;stay here
;*****************************
UDRE_INT_HANDLER:
     LDI   R26,'G'                      ;send 'G'
     OUT   UDR,R26                      ;to UDR
     RETI
```

مثال:
کارکتر G رو میخوایم با فرکانس 8 مگاهرتز و 9600 <-- Baud Rate ارسال بکنیم

Example (Programming in C)

Write a C program to receive bytes of data serially and put them on Port B. Use Receive Complete Interrupt instead of the polling method.

```
#include <avr\io.h>
#include <avr\interrupt.h>

ISR(USART_RXC_vect)
{
   PORTB = UDR;
}

int main (void)
{
   DDRB = 0xFF;                          //make Port B an output
   UCSRB = (1<<RXEN)|(1<<RXCIE);         //enable receive and RXC int.
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
   sei();                                //enable interrupts
   while (1);                            //wait forever
   return 0;
}
```

وقفه در زبان سی:
می خوایم از بیت وقفه مرتبط با دریافت ارسال بکنیم و یک عملیات ارسال رو داشته باشیم

Example (Programming in C)

Write a C program to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz. Use interrupts instead of the polling method.

```
#include <avr\io.h>
#include <avr\interrupt.h>
ISR(USART_UDRE_vect)
{
    UDR = 'G';
}

int main (void)
{
   UCSRB = (1<<TXEN)|(1<<UDRIE);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;

   sei();                          //enable interrupts
   while (1);                      //wait forever
   return 0;
}
```

مثال:
می خوایم کارکتر G رو ارسال بکنیم با استفاده از مکانیزم وقفه

پایان

موفق و پیروز باشید