Using the USART in AVR-GCC

Dean Camera

March 15, 2015

****

Contents

Chapter 1

Introduction

This tutorial will focus on setting up the serial USART on the AVR platform. Although other hardware AVR interfaces (eg, USI) can be configured for limited RS-232 serial transmission and reception such alternatives will not be covered.

1.1 What is the USART?

The vast majority of devices in the current AVR family lineup contain a USART hardware subsystem. The USART hardware allows the AVR to transmit and receive data serially to and from other devices — such as a computer or another AVR.

The USART transmission system differs to most other digital busses in that it does not utilize a separate pin for the serial clock. An agreed clock rate is preset into both devices, which is then used to sample the R_x/T_x lines at regular intervals. Because of this, the USART requires only three wires for bi-directional communication (R_x, T_x and G_{ND}).

1.2 The RS-232 Specification

The serial USART is frequently referred to as RS-232, which is a reference to the RS-232 specification which mandates the logic levels and control signals. While the AVR's normal logic levels are about 3-5V, RS-232 communication uses a low of +3V to +25V for a digital '0', and -3V to -25V for a digital '1'.

If the two USART devices are running at AVR logic levels, the T_x and R_x pins can be connected together directly. If the devices are using RS-232 specification voltages, a level converter circuit is needed.

This tutorial will assume the user is attempting to interface with a computer RS-232 port, which uses the proper RS-232 specification logic levels.

Chapter 2

Setting up the Hardware

To connect your AVR to a computer via the USART port, you will need to add a level converter to your existing circuit. The most common/popular serial level converter is the Maxim MAX232. This small and relatively inexpensive IC will convert your AVR's 5V logic levels to 10V RS-232 and vice versa.

Before you can hook up your MAX232, you need to first identify which pins of your AVR are responsible for serial communication. Looking in your AVR's datasheet, you should be able to identify the two serial port pins (named as alternative pin functions T_x and R_x).

You should wire up your MAX232 to reflect the schematic here. This will enable your AVR's USART hardware to interface with the computer's USART hardware.

Chapter 3

Deciding on your AVR's system clock frequency

As mentioned above, the USART does not use an external clock. To allow the two interfaced devices to communicate together, you need to decide on a baud rate for communication. Also due to the timing-critical nature of the USART, your system's clock should be stable and of a known frequency. The AVR's system clock frequency is very important. The USART clock is derived from the system clock, and it is used to sample the R_x line and set the T_x line at precise intervals in order to maintain the communication.

If the system clock frequency cannot be precisely divided down to a "magic" frequency for perfect communications, it will have a percentage error (where a byte fails to be read or written inside designated time frame). System clocks for perfect USART communications should be multiples of 1.8432MHz which when used will give a 0.00% error.

Other frequencies for the main AVR clock can be used, but as the frequency drifts from the "perfect" multiples of 1.8432MHz, the greater the percentage error will be (and thus the less reliable the serial communication will be). It is generally accepted that error percentages of less than $\pm$ 2% are acceptable.

Looking in your AVR's datasheet in the USART section you should see a table of common baud rates, system clocks and error percentages. You should find a combination which suits your project's constraints and has the lowest possible error percentage.

I will be basing the rest of this tutorial on an example setup; an ATMEGA16 running at 7.3728MHz and a baud rate of 9600bps (bits per second). This combination, because it uses a system clock which is a multiple of the magic 1.8432MHz, has an error percentage of 0.00%.

Chapter 4

Initializing the USART

Now you should have selected a baud rate, system clock and have your AVR set up with a RS-232 level converter — you're all set to go! All you need is the firmware to drive it.

First off, you need to enable both the USART's transmission and reception circuitry. For the ATMEGA16, these are bits named `RXEN` and `TXEN`, and they are located in the control register `UCSRB`. When set, these two bits turn on the serial buffers to allow for serial communications:

```
#include <avr/io.h>

int main (void)
{
   UCSRB = (1 << RXEN) | (1 << TXEN); // Turn on the transmission and reception circuitry
}
```

Next, we need to tell the AVR what type of serial format we're using. The USART can receive bytes of various sizes (from 5 to 9 bits) but for simplicity's sake we'll use the standard 8-bits (normal byte size for the AVR). Looking again at the ATMEGA16 datasheet, we can see that the bits responsible for the serial format are named `UCSZ0` to `UCSZ2`, and are located in the USART control register named `UCSRC`.

The datasheet very handily gives us a table showing which bits to set for each format. The standard 8-bit format is chosen by setting the `UCSZ0` and `UCSZ1` bits. Before we write to the `UCSRC` register however, note a curious peculiarity in our chosen ATMEGA16 AVR. To reduce the number of uniquely addressable registers in the I/O space, the AVR's `UCSRC` and `UBRRH` registers — the latter being explained later in this text) — both share the same physical address. To select between the two, you must also write the `URSEL` bit when writing to `UCSRC`. This isn't the case on all AVR models, so check your device's datasheet.

Let's look at how our code looks right now.

```
#include <avr/io.h>

int main (void)
{
   UCSRB = (1 << RXEN) | (1 << TXEN);   // Turn on the transmission and reception circuitry
   UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character sizes - URSEL bit
         set to select the UCRSC register
}
```

There, we're almost done setting up the USART already! The last thing to set for basic serial communications is the baud rate register. This register sets the clock divider for the USART which is used as a timebase to sample the incoming data at the correct frequency. It also gives the timebase for sending data, so it's vital for RS-232 serial communications.

The baud rate register (**UBRR**) is 16-bit, split into two 8-bit registers as is the case with all 16-bit registers in the AVR device family. To set our baud rate prescaler value, we first need to determine

it. Note that the baud rate register value is **not** the same as the baud rate you wish to use — this is a common point of failure amongst those new to the serial peripheral. Instead, the value must be derived from the following formula (taken from a random AVR8 device datasheet from Atmel):

$$\text{Baud Value} = \frac{\text{F}_{\text{cpu}}}{16 \times \text{BAUD}}$$

Where F_{cpu} is your AVR's system clock frequency (in Hz), and $BAUD$ is the desired communication baud rate in bit-per-second.

Given my example project using a system clock of 7372800Hz and a baud rate of 9600, our formula gives:

$$\begin{aligned}
\text{Baud Value} &= \frac{\text{F}_{\text{cpu}}}{16 \times \text{BAUD}} \\
\text{Baud Value} &= \frac{7372800}{9600 \times 16} - 1) \\
\text{Baud Value} &= \frac{7372800}{153600} - 1 \\
\text{Baud Value} &= 48 - 1 \\
\text{Baud Value} &= 47
\end{aligned}$$

However, it just so happens that the values I chose give a nice, round number result. This isn't always the case; sometimes you'll end up with a required baud rate of 7.5 or similar (for example, if you have a F_{cpu} of 16MHz, and wanted a baud of 115200, giving a required **UBRR** of 138.88). Until such time that Atmel makes a USART that accepts floating point numbers, we need to round to the nearest integer.

To make our life easier, we can turn this formula into a set of handy macros. The system clock frequency F_{cpu} is usually defined as a macro called `F_CPU` in AVR-GCC via your makefile, so all that is needed is the baud rate.

```
#define USART_BAUDRATE 9600

// Naiive Version - has rounding bugs!
//#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

// Better Version - see text below
#define BAUD_PRESCALE ((((F_CPU / 16) + (USART_BAUDRATE / 2)) / (USART_BAUDRATE)) - 1)
```

This avoids hard-coded "magic numbers" (unexplained constants) in our source code, and makes changing the baud rate later on very easy — just change the `USART_BAUDRATE` macro value and recompile your code. You'll notice that the above macro doesn't correspond 1-to-1 with the original formula from the datasheet; the adjusted macro is designed to perform the rounding required to ensure the closest fit to the target baud value, via some algebra trickery and a little knowledge of how integer rounding works.

Now, we need to load this into the baud rate registers, named `UBRRH` (for the high byte) and `UBRRL` (for the low byte). This is simple via a simple bit-shift to grab the upper eight bits of the `BAUD_PRESCALE` constant:

```
// Load upper 8-bits of the baud rate value into the high byte of the UBRR register
UBRRH = (BAUD_PRESCALE >> 8);

// Load lower 8-bits of the baud rate value into the low byte of the UBRR register
UBRRL = BAUD_PRESCALE;
```

Now we're ready to rock and roll!

Chapter 5

Sending and receiving data

Once initialized, we're ready to send and receive data. We do this by the special register named `UDR` — short for "**U**SART I/O **D**ata **R**egister". This is special for two reasons; one, it behaves differently when it is read or written to, and two, it is double buffered in hardware.

By assigning a byte to the `UDR` register, that byte is sent out via the AVR's T_x line. This process is automatic — just assign a value and wait for the transmission to complete. We can tell when the transmission is complete by looking at the transmission completion flag inside the USART control registers.

On the ATMEGA16, the USART *Transmission Complete* flag is located in the control register `UCSRA`, and it is named `TXC`. Using this information we can construct a simple wait loop which will prevent data from being written to the `UDR` register until the current transmission is complete:

```
UDR = ByteToSend; // Send out the byte value in the variable "ByteToSend"
while ((UCSRA & (1 << TXC)) == 0) {}; // Do nothing until transmission complete flag set
```

However, note that this approach is non-optimal. We spend time waiting after each byte which could be better spent performing other tasks, when it is far better to check before a transmission to see if the `UDR` register is ready for data. We can do this by checking the USART *Data Register Empty* flag called `UDRE` instead, also located in the `UCSRA` control register of the ATMEGA16:

```
while ((UCSRA & (1 << UDRE)) == 0) {}; // Do nothing until UDR is ready for more data to be
    written to it
UDR = ByteToSend; // Send out the byte value in the variable "ByteToSend"
```

This is much better, as now time is only wasted before a transmission if the `UDR` register is full. After a transmission we can immediately continue program execution while the hardware sends out the byte stored in `UDR`, saving time.

Now we can move on to receiving data. As mentioned before, the `UDR` register behaves differently between read and write operations. If data is written to it it is sent out via the AVR's T_x pin, however when data is received by the R_x pin of the AVR, it may be read out of the `UDR` register. Before reading the `UDR` register however, we need to check to see if we have received a byte.

To do this, we can check the USART *Receive Complete* flag `RXC` to see if it is set. Again, this is located in the `UCSRA` control register of the ATMEGA16:

```
while ((UCSRA & (1 << RXC)) == 0) {}; // Do nothing until data have been received and is ready
    to be read from UDR
ReceivedByte = UDR; // Fetch the received byte value into the variable "ReceivedByte"
```

Chapter 6

Putting it all together

Right! Now it's time to put everything together! Let's make a simple program that echoes the received characters back to the computer. First, the basic program structure:

```
#include <avr/io.h>

int main (void)
{

}
```

Next, our USART initializing sequence:

```
#include <avr/io.h>

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((((F_CPU / 16) + (USART_BAUDRATE / 2)) / (USART_BAUDRATE)) - 1)

int main (void)
{
   UCSRB = (1 << RXEN) | (1 << TXEN);   // Turn on the transmission and reception circuitry
   UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character sizes

   UBRRH = (BAUD_PRESCALE >> 8); // Load upper 8-bits of the baud rate value into the high byte
        of the UBRR register
   UBRRL = BAUD_PRESCALE; // Load lower 8-bits of the baud rate value into the low byte of the
       UBRR register
}
```

Now, an infinite loop to contain our echo code:

```
#include <avr/io.h>

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((((F_CPU / 16) + (USART_BAUDRATE / 2)) / (USART_BAUDRATE)) - 1)

int main (void)
{
   UCSRB = (1 << RXEN) | (1 << TXEN);   // Turn on the transmission and reception circuitry
   UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character sizes

   UBRRH = (BAUD_PRESCALE >> 8); // Load upper 8-bits of the baud rate value into the high byte
        of the UBRR register
   UBRRL = BAUD_PRESCALE; // Load lower 8-bits of the baud rate value into the low byte of the
       UBRR register

   for (;;) // Loop forever
   {

   }
}
```

And some echo code to complete our example. We'll add in a local variable called `ReceivedByte`, which will always hold the last received character from the computer:

```
#include <avr/io.h>

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((((F_CPU / 16) + (USART_BAUDRATE / 2)) / (USART_BAUDRATE)) - 1)

int main (void)
{
   char ReceivedByte;

   UCSRB = (1 << RXEN) | (1 << TXEN);   // Turn on the transmission and reception circuitry
   UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Use 8-bit character sizes

   UBRRH = (BAUD_PRESCALE >> 8); // Load upper 8-bits of the baud rate value into the high byte
         of the UBRR register
   UBRRL = BAUD_PRESCALE; // Load lower 8-bits of the baud rate value into the low byte of the
        UBRR register

   for (;;) // Loop forever
   {
      while ((UCSRA & (1 << RXC)) == 0) {}; // Do nothing until data have been received and is
           ready to be read from UDR
      ReceivedByte = UDR; // Fetch the received byte value into the variable "ByteReceived"

      while ((UCSRA & (1 << UDRE)) == 0) {}; // Do nothing until UDR is ready for more data to
           be written to it
      UDR = ReceivedByte; // Echo back the received byte back to the computer
   }
}
```

And voila, we have a working serial example! Connect to this project via a serial terminal on your computer, using 8-bit, no parity 9600 baud communication settings and it will echo back anything you send to it.

Chapter 7

Potential Pitfalls

There are several "gotchas" when dealing with the USART. First, make sure your AVR is running of a reliable, known clock source with a low error percentage at your chosen baud rate. New AVRs default to their internal oscillators until their fuses are changed. The internal RC oscillator is of too low a tolerance for use with the USART without external calibration.

Check your datasheet to check for the locations of the different flags. Different AVRs have the control flags located in different control registers (`UCSRA`, `UCSRB` and `UCSRC`).

Make sure your computer terminal is connected at the exact settings your AVR application is designed for. Different settings will cause corrupt data.

Cross your R_x and T_x wires. Your PC's R_x line should be connected to your AVR's T_x line and vice-versa. This is because the R_x/T_x lines are labeled relative to the device they are marked on.

Some AVRs do not have the `URSEL` bit to differentiate between the `UCSRC` and `UBRRH` registers — check your datasheet.

Chapter 8

Advanced serial communication

A more advanced method of serial communication involves the use of the USART interrupts and is covered in another of my AVR tutorials.