Interrupt Driven USART in AVR-GCC

Dean Camera

February 4, 2013

****

Contents

1 Introduction **3**

1.1 What are Interrupts? 3

2 A recap: The simple echo program **4**

3 Populating the ISR **6**

4 Enabling the USART Receive Interrupt **7**

5 Putting it all together **9**

Chapter 1

Introduction

This tutorial is an extension of my previous tutorial, "Using the USART in AVR-GCC". This tutorial will teach the basics for creating interrupt-driven USART communications. It assumes that the reader has both read and fully understood my previous tutorial on basic serial communications.

1.1 What are Interrupts?

AVRs — and almost all microcontrollers — contain a feature known as interrupts. Interrupts, as their name implies, allow for external events (such as inputs from the user or AVR peripheral) to momentarily pause the main microcontroller program and execute an "Interrupt Service Routine" (often shortened to the abbreviation "ISR") before resuming the main program where it left off. Interrupts are extremely useful for dealing with irregular external input (such as pin changes or the arrival of a serial byte), as well as for processing "background tasks" like togling a LED each time a timer overflows.

In this tutorial, we are going to make use of the AVR's USART peripheral interrupts.

Chapter 2

A recap: The simple echo program

From the last serial tutorial, we have created a simple program from scratch which will echo bytes received on the AVR's USART interface. The full program listing is as follows:

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

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 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
   }
}
```

Readers should be able to fully understand this code — if you cannot please re-read the previous tutorial on basic serial communication.

Now, we want to extend this code so that the serial data is echoed back when received in an interrupt, rather than our main program loop. To do this, first we need to include the avr-libc standard library header, `<avr/interrupt.h>`. This file contains library functions and macros which relate to the interrupt functionality of the AVR. We'll add this to the top of our code, below the standard device header, `<avr/io.h>` include:

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

Once included, we now have a way to make our ISR to deal with the serial reception. To make an ISR, we use the syntax:

```
ISR({Vector Name}, {Options})
{
   // Code to be executed when ISR fires
}
```

And place it in our program as if it was a normal function. To add one to deal with the reception of a byte via the USART, we need to look for the appropriate name in our AVR's datasheet. In the datasheet for our example AVR, the ATMEGA16, we see that the name of the interrupt for when a byte is received is "USART_RXC". The standard avr-libc device header file <avr/io.h> — included in our program as well as any other AVR-GCC program involving the AVR's I/O functionality — defines the vector names for us.

The avr-libc symbolic names for each of the interrupt vectors is identical to the datasheet, with the addition of a "_vect" suffix to denote that it is a vector name. So, as our datasheet listed "USART_RXC" as the vector name, the syntax for our program is:

```
ISR(USART_RXC_vect, ISR_BLOCK)
{
   // Code to be executed when the USART receives a byte here
}
```

Which we'll place at the end of our program, after our main function. Note that here I've used the ISR option of ISR_BLOCK, which is what most interrupt service routines should use except in special situations. That's outside the scope of this tutorial, but please read through the avr-libc manual if you wish to know more about this.

The new program code looks like this:

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

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 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
   }
}

ISR(USART_RXC_vect)
{
   // Code to be executed when the USART receives a byte here
}
```

Chapter 3

Populating the ISR

At the moment our new USART reception ISR doesn't actually do anything — we've just defined it. We want it to echo back the byte that is sent, so we'll move our main loop code:

```
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
```

Over to it. However, we can now remove the two while loops — since the ISR only fires when a byte is received, and only one byte is sent after each reception, we can guarantee that both checks are now redundant. When the ISR fires we know that there is both a byte received in the USART input buffer, as well as nothing in the output buffer. Using this knowledge, we can simplify our ISR code to the following:

```
ISR(USART_RXC_vect)
{
   char ReceivedByte;
   ReceivedByte = UDR; // Fetch the received byte value into the variable "ByteReceived"
   UDR = ReceivedByte; // Echo back the received byte back to the computer
}
```

Note that I've also moved the variable declaration of the variable `ReceivedByte` over to the ISR, as that is now where it is actually used.

It's worth mentioning at this point a small section on the datasheet about the RXC interrupt:

> When interrupt-driven data reception is used, the receive complete routine must read the received data from UDR in order to clear the RXC Flag, otherwise a new interrupt will occur once the interrupt routine terminates.

That's important for us to remember: if you are using the USART RXC interrupt, you must read a byte from the `UDR` register to clear the interrupt flag. We do that in our above code, but keep it in mind for your future projects!

Chapter 4

Enabling the USART Receive Interrupt

Let's take a look at the latest incarnation of our test code:

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

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 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
   {
         // Do nothing - echoing is handled by the ISR instead of in the main loop
   }
}

ISR(USART_RXC_vect)
{
   char ReceivedByte;
   ReceivedByte = UDR; // Fetch the received byte value into the variable "ByteReceived"
   UDR = ReceivedByte; // Echo back the received byte back to the computer
}
```

If you compile and run this, you'll notice that nothing happens — no characters are echoed back to the connected computer. This is because although we've defined and populated the ISR, we haven't enabled it. To do so, we need to do two things:

1. Turn on global interrupts
2. Enable the USART Byte Received interrupt

Item one is simple, so we'll do that first. The AVR microcontrollers contain a global flag which can be set or cleared to enable or disable the handling of interrupts. Note that setting this flag doesn't enable all interrupts, it only allows for the possibility of running them. If the *Global Interrupt Enable* flag is disabled, all interrupts will be ignored, even if they are enabled (more on that later).

To turn on the *Global Interrupt Enable* flag, we can use the macro `sei()` which the avr-libc header file `<avr/interrupt.h>` helpfully defines for us. This is so named as it generates a "`SEI`" assembly instruction in the final code listing, which the AVR interprets as an order to set the *Global Interrupt Enable* flag. The compliment of `sei()` is `cli()` (to turn off the handling of interrupts) however we will not be using that macro in this tutorial.

We'll add our `sei()` instruction to our main routine, after configuring the USART registers:

```
// ...
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

sei(); // Enable the Global Interrupt Enable flag so that interrupts can be processed

for (;;) // Loop forever
// ...
```

Now for item two on our list, which needs to be performed before the interrupt will be enabled. We need to specifically enable the USART *Receive Complete* interrupt, which we can do by setting the appropriate flag in the USART control register.

In the ATMEGA16, this bit is called `RXCIE` (short for "**R**ecieve **C**omplete **I**nterrupt **E**nable") and is part of the register `UCSRB`. Setting this bit enables the handling of the `USART_RXC` event vector:

```
UCSRB |= (1 << RXCIE);
```

We'll add this to our main routine, before our new "`sei();`" command.

Chapter 5

Putting it all together

Now we have a working interrupt driven serial example:

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

#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 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

   UCSRB |= (1 << RXCIE); // Enable the USART Recieve Complete interrupt (USART_RXC)
   sei(); // Enable the Global Interrupt Enable flag so that interrupts can be processed

   for (;;) // Loop forever
   {
         // Do nothing - echoing is handled by the ISR instead of in the main loop
   }
}

ISR(USART_RXC_vect)
{
   char ReceivedByte;
   ReceivedByte = UDR; // Fetch the received byte value into the variable "ByteReceived"
   UDR = ReceivedByte; // Echo back the received byte back to the computer
}
```

Which, like out original program, will echo characters received via the USART. However, because our program is now interrupt driven, we can add in code into the main loop which will be executed when data is not received — such as flashing a LED.

Interrupts allow for infrequent “background” tasks to be executed when they occur, without posing a run-time penalty of having to poll the hardware until the even occurs. This frees up our main loop to take care of the critical code, with the interrupt code pausing the main code to execute when the event of interest ocurs.

Interrupts should be made to be as short as possible in execution time. This is because while one ISR is executing, others are blocked and thus if another ISR condition occurs while one ISR is executing, that ISR event will be missed or delayed.

Because of this, communication ISRs are generally made short by receiving in characters via an ISR, and placing them into a buffer which the main code may read at its leisure. This ensures that received data will not be missed, while giving the main program time to complete what it is currently doing before having to process the input. Similarly, long transmissions may be made by placing the data to be sent into a buffer, and have an interrupt send the data as the hardware is ready while the main program performs other tasks.