AVR Helper Library 1.3

Dean Ferreyra
dean@octw.com
http://www.bourbonstreetsoftware.com/

November 11, 2008

Contents

Chapter 1

Introduction

The AVR Helper Library is a small collection of modules written in C that I have found useful while developing and prototyping AVR-based software. It includes a buffered USART module, buffered and unbuffered SPI modules, and some odds and ends like “gentle” EEPROM update functions, a thermistor temperature calculation function, and “endian” conversion functions.

I have released this code under the GNU Lesser General Public License with the hope that others might find it useful.

Chapter 2

Build and Installation

To build and install the AVR Helper Library, you need to have already installed the AVR GCC cross-compiler.

2.1 Build

To install the AVR Helper Library, unpack the source tree. If you downloaded the `*.tgz` version, you can do this with the following command:

```
tar -xzf helper-1.3-src.tgz
```

Next, change directory to the head of the source tree that was just unpacked and run the `make` command:

```
$ cd helper-1.3
$ make
```

This should build the AVR Helper Library for all the supported AVR microcontrollers.

2.2 Installation

Once the libraries have been successfully built, you need to install the library components to a location where the AVR GCC compiler can find them.

The default installation directory prefix is `/usr/local`. If you wish to change this, you will have to edit the `src/Makefile` file in the source tree. In the Makefile, you'll find a line that looks like this:

```
prefix = /usr/local
```

Change this line to desired directory prefix. For example, if you're using WinAVR, you will need to change this this point to where WinAVR has been installed. For example, if WinAVR has been installed in `c:\WinAVR`, then change the prefix line to this:

```
prefix = c:/WinAVR
```

Finally, from the head of the source tree, or from the `src` directory, run this:

```
$ make install
```

Chapter 3

EEPROM

This module provides "gentle" EEPROM writing routines to help extend the life of the on-chip EEPROM. Before each EEPROM data byte is written, these functions read the byte from the EEPROM and only write the new byte if the values are different.

3.1 Usage

To use the EEPROM writing functions, use this:

```
#include <eeprom-gentle.h>
```

3.2 Functions

`void gentle_eeprom_write_byte(uint8_t *addr, uint8_t val)` ***function***

Writes a single byte `val` to EEPROM address `addr`, but only if the byte in the EEPROM is different.

`void gentle_eeprom_write_word(uint16_t *addr, uint16_t val)` ***function***

Writes the bytes of a single 16 bit word `val` to EEPROM address `addr`, but only the bytes that are different than what is currently found in the EEPROM.

`void gentle_eeprom_write_dword(uint16_t *addr, uint32_t val)` ***function***

Writes the bytes of a single 32 bit word `val` to EEPROM address `addr`, but only the bytes that are different than what is currently found in the EEPROM.

`void gentle_eeprom_write_block(const void *buf, void *addr, size_t n)` ***function***

Writes `n` bytes of the block pointed to by `buf` to EEPROM address `addr`, but only the bytes that are different than what is currently found in the EEPROM.

Chapter 4

USART

This module provides basic buffered USART support for Atmel AVR microcontrollers. It support microcontrollers with 1 or 2 USART ports.

This module separates the actions of receive and transmit into independent parts. This way, if you will only be receiving or only be transmitting, you only need to link in and initialize the relevant part.

4.1 Usage

To use the USART functions for microcontrollers that only support one USART port, use this:

```
#include <uart.h>
```

and use the function names as shown below.

To use the USART functions for microcontrollers that support two USART ports, use this:

```
#include <uart0.h>
#include <uart1.h>
```

and append the digit `0` or `1` to the function names shown below to pick the appropriate port. For example, the following initializes the transmit functions of USART port 1 and transmits a single character:

```
uart_tx_init1();
uart_tx_char1('x');
```

4.2 USART Receive Functions

`void uart_rx_init(void)` ***function***

This function initializes the receive part of this module. This entails enabling the USART receive line and enabling the USART receive interrupt. Also, it automatically links in the default receive interrupt service routine from the library.

`void uart_rx_init_no_isr(void)` ***inline***

This function is identical to `uart_rx_init()`, but it does *not* link in the default receive interrupt service routine. In this case, you must provide your own interrupt service routine.

`void uart_rx_clear(void)` ***function***

This function clears the receive buffer.

`int uart_rx_char(void)` ***function***

This function takes a character from the receive buffer and returns it. If the receive buffer is empty, this function returns `-1`.

4.3 USART Transmit Functions

`void uart_tx_init(void)` ***function***

This function initializes the transmit part of this module. This entails enabling the USART transmit line. Also, it automatically links in the default transmit interrupt service routine from the library.

`void uart_tx_init_no_isr(void)` ***inline***

This function is identical to `uart_tx_init()`, but it does *not* link in the default transmit interrupt service routine. In this case, you must provide your own interrupt service routine.

`void uart_tx_clear(void)` ***function***

This function clears the transmit buffer.

`bool uart_tx_char(char c)` ***function***

If there is room in the transmit buffer, this function places the given character, `c`, into the transmit buffer. If the transmit interrupt is disabled, it enables the interrupt which starts the transmission process. If the transmit interrupt is already enabled, a transmission is already in progress and the character will eventually be transmitted. The function then returns `true`.

If the transmit buffer is full when this function is called, it does nothing and returns `false`.

`bool uart_tx_str(const char* str)` ***function***

If there is room in the transmit buffer, this function places the given string, `str`, into the transmit buffer. If the transmit interrupt is disabled, it enables the interrupt which starts the transmission process. If the transmit interrupt is already enabled, a transmission is already in progress and the character will eventually be transmitted. The function then returns `true`.

If the transmit buffer is full when this function is called, it does nothing and returns `false`.

`bool uart_tx_str_P(PGM_P str)` ***function***

This function is analogous to `uart_tx_str()`, except that instead of taking a string in RAM, `str` points to a string stored in the program space.

`const char* uart_tx_str_partial(const char* str)` ***function***

This function places into the transmit buffer as many characters from the given string, `str`, as will fit.

If any characters were placed into the transmit buffer, and the transmit interrupt is disabled, it enables the interrupt which starts the transmission process. If the transmit interrupt is already enabled, a transmission is already in progress and any characters placed into the transmit buffer will eventually be transmitted.

If all the characters in `str` were placed into the transmit buffer, then this function returns `0`. Otherwise, it returns a pointer to the characters in the string that were not placed into the transmit buffer. This way, you can use the returned string to try again latter.

`PGM_P uart_tx_str_partial_P(PGM_P str)` ***function***

This function is analogous to `uart_tx_str_partial()`, except that instead of taking a string in RAM, `str` points to a string stored in the program space. Also, it returns a pointer to the remaining characters stored in the program space.

`bool uart_tx_data(const char* data, uint8_t count)` ***function***

This function is analogous to `uart_tx_str()`, except that instead of taking `'\0'` terminated string, it takes a character pointer to data, `data`, and the number of characters to transmit, `count`. If all `count` characters will not fit in the transmit buffer, this function returns `false`. Otherwise, it returns `true`.

`uint16_t uart_generate_ubrr(uint32_t fosc, uint32_t baud)` ***function***

This function calculates the closest `UBRR` value for the given oscillator frequency, `fosc`, and the desired baud rate, `baud`.

Chapter 5

Software UART Transmission

This module provides buffered, software-based, transmit-only UART functionality. TIMER0 is the baud-rate generator. By default, this module twiddles `PORTE` bit 1 or `PORTD` bit 1. By redefining a couple of functions you can make any output bit the transmit line.

5.1 Usage

To use the transmit-only UART functions, place this include in your source file:

```
#include <tx-only.h>
```

5.2 Functions

`void tx_only_init(uint8_t timer0_mode, uint8_t timer0_count)` ***function***

This function initializes TIMER0 by placing it in the given mode, `timer0_mode`, and setting the counter value to `timer0_count`. Finally the `OCIE0` interrupt is enabled.

Only the three low-order bits of the `timer0_mode` argument are used. Register `TCCR0` is updated with these bits, plus bit `CTC0`. Register `OCR0` is updated with `timer0_count`.

For example, to simulate a 9600 baud UART transmission line on an ATmega103 running at 4 MHz, call `tx_only_init()` like this:

```
tx_only_init(0x02, 4000000 / 8 / 9600);
```

`void tx_only_clear()` ***function***

This function clears the transmit buffer.

`bool tx_only_chr(uint8_t c)` ***function***

If there is room in the transmit buffer, this function places the given character, `c`, into the transmit buffer and the function then returns `true`. Otherwise, it returns `false`.

`bool tx_only_str(const uint8_t* str)` ***function***

If there is room in the transmit buffer, this function places the given string, `str`, into the transmit buffer and returns `true`. Otherwise, it returns `false`.

`bool tx_only_str_P(PGM_P str)` ***function***

This function is analogous to `tx_only_str()`, except that instead of taking a string in RAM, `str` points to a string stored in the program space.

`void tx_only_port_low()` ***optional user-defined function***

`void tx_only_port_high()` ***optional user-defined function***

These two functions are provided by the programmer to twiddle the bits of the transmission line. The "low" and "high" that appear in their names are in reference to the signal level, where "low" means a negative voltage, and "high" means a positive voltage.

These functions are optional. If you don't provide them, the library supplies its own version that twiddles `PORTE` bit 1 (or if the microcontroller does not have `PORTE`, then `PORTD` bit 1).

5.3 Caveats

As written, this module has many limitations:

- It supports only one UART.
- It is hard-coded to use TIMER0.
- All of the functions are linked into the image, even if they're not used.
- Because the interrupt service routine calls two possibly user-defined functions, it is especially slow.

Chapter 6

Basic SPI

This module provides a basic interface to the microcontroller SPI in master mode. It utilizes an interrupt service routine to clock out the SPI data asynchronously.

6.1 Usage

To use the SPI module, place the following in your source files:

```
#include <spi2.h>
```

6.2 Functions

`void spi2_init(void)` ***function***

This function places the SPI into "master" mode, enables the SPI, and enables the SPI interrupt. The library provides the required interrupt service routine.

`bool spi2_busy(void)` ***function***

This function returns `true` if the SPI module is busy clocking SPI data. While the SPI module is busy, `spi2_send_byte()` and `spi2_send_data()` will do nothing and will return `false`.

`bool spi2_send_byte(uint8_t* b, volatile bool* done_flag)` ***function***

This function begins an asynchronous SPI exchange of one byte. The data to transmit is the byte pointed to by `b`. If `done_flag` is not `NULL`, when the exchange is complete, the interrupt service routine sets this flag to `true`. After the exchange, the byte pointed to by `b` will contain the received byte.

While the module is busy exchanging the byte, it is important that the location pointed to by `b` remain valid. For example, it would be an error if `b` points to a location on the stack that goes out of scope while the exchange is taking place.

`bool spi2_send_data(uint8_t* d, uint8_t s, volatile bool* done_flag)` ***function***

This function begins an asynchronous SPI exchange of `s` bytes of the data pointed to by `d`. If `done_flag` is not `NULL`, when the exchange is complete, the interrupt service routine sets this flag to `true`. After the exchange, the data pointed to by `d` will contain the received data.

While the module is busy exchanging the byte, it is important that the location pointed to by `d` remain valid. For example, it would be an error if `d` points location on the stack that goes out of scope while the exchange is taking place.

Chapter 7

Buffered SPI

This module provides a buffered interface to the microcontroller SPI in master mode. It utilizes an interrupt service routine to clock out the SPI data asynchronously.

Each data transfer request can include execution of a user-defined function just before the transmission begins and just after the transmission completes. The user-defined functions can do things like enable and disable chip-select lines and change SPI clocking frequencies to allow you to buffer SPI transmissions for multiple SPI devices.

7.1 Usage

To use the SPI module, place the following in your source files:

```
#include <spi3.h>
```

7.2 Functions

`void spi3_init(void)` ***function***

This function places the SPI into "master" mode, enables the SPI, and enables the SPI interrupt. The library provides the required interrupt service routine.

`bool spi3_clock_data(uint8_t* data, uint8_t size, void (*pre)(void), void (*post)(void), volatile bool* done)` ***function***

This function buffers an SPI data transfer request. A data transfer request consists of:

`data` Pointer to the data to exchange.

`size` Number of bytes to exchange.

`pre` Thunk called by the interrupt service routine before the exchange begins.

`post` Thunk called by the interrupt service routine after the exchange ends.

`done` Pointer to a Boolean set by the interrupt service routine to `true` once the exchange ends and the call to `post` completes.

If there is room in the buffer, the request is buffered and `spi3_clock_data()` returns `true`. Otherwise, `spi3_clock_data()` returns `false`. The data is clocked onto the SPI bus once all previously buffered requests are completed. Any pending requests placed in the buffer by calls to `spi3_pend_data()` will no longer be considered pending and they will all eventually be clocked onto the SPI.

The arguments `pre`, `post`, and `done` are optional; passing a `NULL` causes `spi3_clock_data()` to ignore that argument.

Note that the memory pointed to by `data` and `done` must remain valid until the exchange is complete. For example, it would be an error if `data` were to point to a location on the stack and that location were to go out of scope while the exchange was taking place.

Note also that the `pre` and `post` thunks are called directory from the interrupt service routine.

`bool spi3_pend_data(uint8_t* data, uint8_t size, void (*pre)(void), void (*post)(void), volatile bool* done)` ***function***

This functions is similar to `spi3_clock_data()`, except that the data transfer request is held pending in the buffer without being clocked onto the SPI bus. This function can be called repeatedly to buffer more requests. Pending transfer requests will begin being clocked onto the SPI the next time `spi3_clock_data()` is called.

This function is useful when a multi-byte exchange needs to take place without interruption, but when it is more convenient to build the multi-byte exchange in pieces.

`void spi3_clear_pending(void)` ***function***

This function clears any pending requests that have been placed in the buffer by `spi3_pend_data()`.

7.3 Unbuffered Interface

There is also an *unbuffered* version of this interface that does not use an interrupt service routine. To use the unbuffered version, place the following in your source files:

```
#include <spi3-no-isr.h>
```

Every function from the buffered interface is available, but each function is prefixed with `spi3_no_isr` instead of `spi3`.

Because the data is not buffered, `spi3_no_isr_clock_data()` and `spi3_no_isr_pend_data()` both immediately clock out the data to the SPI bus.

`void spi3_no_isr_wait(void)` ***user-defined function***

This user defined function is called by `spi3_no_isr_clock_data()` and `spi3_no_isr_pend_data()` before placing the next byte into `SPDR`. At the very least, it should wait for the `SPIF` flag in `SPSR` to be set, like this:

```
void spi3_no_isr_wait(void)
{
    while (! (SPSR & _BV(SPIF)))
        ;
}
```

By having the wait function be a user-defined function, you can program in an escape or a timeout check, something like this:

```
void spi3_no_isr_wait(void)
{
    while (! (SPSR & _BV(SPIF)))
        check_timeout();
}
```

Chapter 8

Resistor Divider Networks

This module provides functions to calculate values from resistor divider networks.

8.1 Usage

To use the resistor divider network functions, use this:

```
  #include <resistor-network.h>
```

8.2 Functions

`uint16_t divider_network_lower_resistor(uint16_t v_ad_max, uint16_t v_ad_counts, uint16_t r_upper)` ***function***

This function calculates the resistance of the lower resistor of a resistor divider network, given the maximum A/D counts `v_ad_max`, the measured A/D counts `v_ad_counts` for the divider network, and the known resistance `r_upper` in ohms of the upper resistor. It uses the equation

$$R_{\text{lower}} = R_{\text{upper}} \times \frac{V_{\text{measured}}}{V_{\text{max}} - V_{\text{measured}}}$$

to arrive at the result.

`uint16_t divider_network_voltage(uint16_t v_ad_counts, uint16_t r_upper, uint16_t r_lower)` ***function***

This function calculates the voltage feeding into a resistor network given the measured A/D counts `v_ad_counts`, and the the known resistances in ohms of the upper resistor `r_upper` and the lower resistor `r_lower`. It uses the equation

$$V_{\text{network}} = V_{\text{measured}} \times \frac{R_{\text{upper}} + R_{\text{lower}}}{R_{\text{lower}}}$$

to arrive at the result.

Chapter 9

Thermistor

This module provides a function to calculate the temperature of a thermistor using an exponential model for NTC (negative temperature coefficient) thermistors.

9.1 Usage

To use the thermistor function, use this:

```
#include <thermistor.h>
```

9.2 Functions

`float thermistor_t2_at_r2(float beta, float t1, float r_t1, float r_t2)` ***function***

This function uses an exponential model for NTC thermistors to approximate the thermistor's temperature from its resistance. This function returns a thermistor's temperature given the thermistor's β value, `beta`, the nominal resistance, `r_t1`, at the nominal temperature, `t1`, and the measured resistance, `r_t2`. The temperatures must be in an absolute temperature scale; e.g., Kelvin. The equation used is:

$$T = \left[\frac{\ln (R/R_0)}{\beta} + \frac{1}{T_0} \right]^{-1}$$

9.3 Caveat

Generally, this model is not as accurate as a table lookup can be. The measured temperature should fall within the range that the β value was calculated for, and if you have a choice of β values, you should choose the β value calculated over the smallest range that still suits your expected temperature range.

Also, this module uses floating calculations and `log()` to perform its calculation. The floating-point library is large and floating point calculations can be relatively slow.

Chapter 10

Network Order

This module provides functions to convert from the native byte order on the AVR microcontroller to a big-endian network order like the order used by the Internet Protocol.

10.1 Usage

To use the network order module, place the following in your source files:

```
#include <network.h>
```

10.2 Functions

`uint16_t htons(uint16_t host)` ***function***

This function takes a 16 bit word in host order, `host`, and returns it in network order.

`uint32_t htonl(uint32_t host)` ***function***

This function takes a 32 bit word in host order, `host`, and returns it in network order.

`uint16_t ntohs(uint16_t network)` ***function***

This function takes a 16 bit word in network order, `network`, and returns it in host order.

`uint32_t ntohl(uint32_t network)` ***function***

This function takes a 32 bit word in network order, `network`, and returns it in host order.

`uint16_t ntohb2(const uint8_t* network2)` ***function***

This function takes a pointer to two bytes in network order, `network2`, and returns its value in host order.

`void htonb2(uint16_t host, uint8_t* network2)` ***function***

This functions takes a 16 bit word in host order, `host`, and writes its value in network order at the two bytes pointed to by `network2`.

`uint32_t ntohb3(const uint8_t* network3)` ***function***

This function takes a pointer to three bytes in network order, `network3`, and returns its value in host order.

`void htonb3(uint32_t host, uint8_t* network3)` ***function***

This functions takes a 32 bit word in host order, `host`, and writes its value in network order at the three bytes pointed to by `network3`. Only the least significant 24 bits of `host` are used in this conversion.

`uint32_t ntohb4(const uint8_t* network4)` ***function***

This function takes a pointer to four bytes in network order, `network4`, and returns its value in host order.

`void htonb4(uint32_t host, uint8_t* network4)` ***function***

This functions takes a 32 bit word in host order, `host`, and writes its value in network order at the four bytes pointed to by `network4`.

Index