Detailed Explanation of `Voltmeter.asm` and `VoltmeterTest.py` Code

In a previous note, entitled *Detailed Explanation of `BlinkingLEDUsingTimer0WithOVFInterrupt.asm` Code*, I described ATmega328P assembly code with an interrupt. We are now ready to look at the `Voltmeter.asm` code which returns voltage output readings to the Pi via the SPI interface. It involves two interrupts, one when the Pi requests a reading (`PCINT0`, pin-change interrupt 0), and one each time a byte is sent out via the SPI bus to the Pi (`SPI_STC`, serial peripheral interface serial transfer complete).

Communication via the SPI protocol is a bit tricky – it is two-way and always gives an answer even when the other end is not listening. Its two data lines are called MOSI (master out, slave in) and MISO (master in, slave out). There is also a CLK (clock) line to control the transfer process. If a master sends a message and the slave is not listening, there is no reliable indication that the reply message did not originate from the slave. This system works best when commands are sent and acknowledgments are returned. In this `Voltmeter.asm` program, four zero bytes are sent by the Pi and four bytes are returned. If the ATmega328P is not responding, four zero bytes appear to come back. If the ATmega328P is responding, the returning four bytes consist of a repeat of the first request byte, the two bytes with the voltage value (high byte first), and then finally a validity byte that is `0x01` if the voltage bytes are valid and `0x00` if not. Only readings with a `0x01` as the last byte are taken as valid. The receiving code at the Pi is the following Python program `VoltmeterTest.py`:

```
#! /usr/bin/env python3

import spidev
import RPi.GPIO
from time import time,sleep,asctime,clock
RPi.GPIO.setwarnings(False)
RPi.GPIO.cleanup()    # We leave the chip select pin high after programming
                      #    to let the ATmega328P run the program.
                      #    cleanup() is run to regain control without an warning.
sleep(0.02)
RPi.GPIO.setmode(RPi.GPIO.BOARD)
sleep(0.02)

chipSelect=0          # This is GPIO8 which is header pin 24

spiPort=spidev.SpiDev(0,chipSelect)   # The first parameter here is 0 because that
                                      #   is the first number when you do
                                      #   ls /dev/spidev*
                                      #   when the kernel module spi_bcmXXXX is loaded.
                                      # It has always been 0 for Raspberry Pi versions.

# According to https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md#driver
# According to http://www.takaitra.com/posts/492
# The allowable values for spiPort.max_speed_hz are
#       125000000
#       62500000
#       31200000
#       15600000
#       7800000
#       3900000
#       1953000
#       976000
#       488000
#       244000
#       122000
#       61000
#       30500
#       15200
#       7629
# setting a speed different from these will result in the next lower speed
# or if less than 7629, will set 7629 Hz.

# Different max speeds were tested for the time to get 500 voltages from the ATmega328P
# using the command
#   time sudo nice -n 20 ./VoltmeterTest.py > VoltmeterTestOut
# (The nice setting seems to have little effect whether is is -19 or +20 or not present.
# The sudo also appears to not matter.)
# Running it with different speeds gives the following results:
#  spiPort.max_speed_hz=7629    # 12.48 ms/reading
#  spiPort.max_speed_hz=15200   # 8.20 ms/reading
#  spiPort.max_speed_hz=30500   # 5.64 ms/reading
#  spiPort.max_speed_hz=61000   # 4.62 ms/reading  Best
#  spiPort.max_speed_hz=122000   # 6.44 s
#  spiPort.max_speed_hz=244000   # failed

spiPort.max_speed_hz=61000

t0 = time()
```

```
def getVoltage():
    global t0, start
    response=4*[0]
    count=0
    while count < 1000 and response[3]!=1:
        response=4*[0]
        response = spiPort.xfer2([0,0,0,0])
        sleep(0.000001)  # This 1 us of sleep seems to make a significant difference
                         # Try removing it and see what happens.
                         # This sensitivity to timing is typical for peripheral operations.
        count += 1
    t1=time()
    print('{0:8.2f}: {1:5d} [{2:3d}, {3:3d}, {4:3d}, {5:4d}]  {6:6.3f} V'\
          .format(1000*(t1-t0), count,response[0],response[1],response[2],response[3],\
          3.3*(response[1]*256+response[2])/1024.))
    t0 = t1
    return

for i in range(100):
    getVoltage()

spiPort.close()
```

Walking through the assembly source code

The `Voltmeter.asm` code is in the directory `/home/pi/Programming/Assembly/Voltmeter/` . With labels are shown in bold face, it is:

```
.NOLIST
.INCLUDE "../m328Pdef.inc"
.LIST
.DEF    AD_Done       = R19  ; 0 if busy, 1 if data is ready
.DEF    bytesToGo     = R22
.DEF    statusFlags   = R23
.org    0x0000

; Connect voltage to be measured to pin PC1
; Connect pin PB2 to pin GP7 on header J1 for chip select of SPI

jmp     RESET
        reti            ; EXT_INT0
        nop
        reti            ; EXT_INT1
        nop
jmp     OUTPUT_REQUEST  ; PCINT0
        reti            ; PCINT1
        nop
        reti            ; PCINT2
        nop
        reti            ; WDT
        nop
        reti            ; TIM2_COMPA
        nop
        reti            ; TIM2_COMPB
        nop
        reti            ; TIM2_OVF
        nop
        reti            ; TIM1_CAPT
        nop
        reti            ; TIM1_COMPA
        nop
        reti            ; TIM1_COMPB
        nop
        reti            ; TIM1_OVF
        nop
        reti            ; TIM0_COMPA
        nop
        reti            ; TIM0_COMPB
        nop
        reti            ; TIM0_OVF
        nop
jmp     SPI_XFER_DONE   ; SPI_STC (SPI Transfer Complete)
        reti            ; USART_RXC
        nop
        reti            ; USART_UDRE
        nop
        reti            ; USART_TXC
        nop
        reti            ; ADC
        nop
        reti            ; EE_RDY
        nop
        reti            ; ANA_COMP
        nop
        reti            ; TWI
        nop
```

```
        reti            ; SPM_RDY
        nop

;*********************  Reset Handler  **********************

RESET:                          ; Program initialization
ldi     r16,high(RAMEND) ; Initialize stack pointer
out     SPH,r16
ldi     r16,low(RAMEND)
out     SPL,r16

lds     r16,MCUCR               ; Enable pull-ups for all input ports without changing other settings
andi    r16,~(1<<PUD)           ;    in the MCUCR (microcontroller unit control register).
sts     MCUCR,r16

cbi     DDRB,DDB2               ; Set PINB2 direction as input to receive the Chip Select signal
sbi     PORTB,PINB2             ; Set it to 1 initially.

sbi     DDRB,DDB4               ; Set SPI signal directions - PINB4 for MISO, PINB3 for MOSI
cbi     DDRB,DDB3

sbi     DDRB,DDB1               ; Set PINB1 is used to toggle an LED to signal activity.
cbi     PORTB,PORTB1            ; It will be turned on during data transmission to Raspberry Pi.

clr     AD_Done

; Set A/D reference to external of 3.3 V wired to VREF in Gertboard and set
;   input multiplexer to use PINC1 (ADC1).
ldi     r16,(0<<REFS1)|(0<<REFS0)|(0<<ADLAR)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(1<<MUX0)
sts     ADMUX,r16

ldi     r16,(1<<ADC1D)          ; Disable digital input for PINC1
sts     DIDR0,r16

; Enable power to the spi and D/A converter, and disable power to timer/counter-0, timer/counter-1,
;   timer/counter-2, two-wire interface, and USART.  Note: a 1 means disable power
ldi     r16,(1<<PRTWI)|(1<<PRTIM2)|(1<<PRTIM0)|(1<<PRTIM1)|(0<<PRSPI)|(1<<PRUSART0)|(0<<PRADC)
sts     PRR,r16

ldi     r16,(1<<PCINT2)         ; Enable pin change interrupt on PINB2 in pin change mask 0
sts     PCMSK0,r16
ldi     r16,(0<<PCIE2)|(0<<PCIE1)|(1<<PCIE0)
sts     PCICR,r16

; Enable SPI interface by setting SPI-done interrupt, SPIE, and
;   set as slave with ordinary protocol defaults.
;   Also set the clock frequency to fosc/4.
ldi     r16,(1<<SPIE)|(1<<SPE)|(0<<DORD)|(0<<MSTR)|(0<<CPOL)|(0<<CPHA)|(0<<SPR1)|(0<<SPR0)
out     SPCR,r16
clr     r16
out     SPSR,r16

ldi     r16,(0<<SM2)|(0<<SM1)|(0<<SM0)|(1<<SE)    ; Go to ADC noise reduction sleep while waiting for interrupts.
sts     SMCR,r16

sei                             ; Enable interrupts globally.
GOODNIGHT:
sleep
rjmp    GOODNIGHT


;************  Output-request Interrupt Handler  **************
; The following routine is run when PINB2 changes.

OUTPUT_REQUEST:                 ; Chip select signal on PINB2 has changed.
in      statusFlags,SREG ; Save status flags

sbi     PINB,PINB1              ; Change state of LED

in      r16,SPDR

in      r16,PINB               ; Check state of SPI chip select pin
andi    r16,(1<<PINB2)         ; Enable SPI interrupts if low,
breq    END_OUTPUT_REQUEST

lds     r16,ADCSRA              ; Check if D/A has is enabled, if not enable it
andi    r16,1<<ADEN
brne    DA_ALREADY_ENABLED

ldi     r16,(0<<ACME) ; Set ADC for single-conversion mode
sts     ADCSRB,r16

; Enable A/D, start a single conversion, without interrupt, and set frequency division factor to 128.
ldi     r16,(1<<ADEN)|(1<<ADSC)|(0<<ADATE)|(0<<ADIF)|(0<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
sts     ADCSRA,r16
rjmp    DA_BUSY

DA_ALREADY_ENABLED:
```

```
lds     r16,ADCSRA              ; Check if D/A conversion is still in progress
andi    r16,1<<ADIF
breq    DA_BUSY

ldi     AD_Done,0x01            ; D/A conversion is complete
ldi     bytesToGo,3
rjmp    END_OUTPUT_REQUEST

DA_BUSY:
ldi     AD_Done,0x00            ; D/A conversion is still in progress

END_OUTPUT_REQUEST:
out     SREG,statusFlags ; Restore status flags
reti


;***********  SPI-Data-Sent Interrupt Handler  ***************

SPI_XFER_DONE:                  ; SPI transfer complete, ready for next byte
in      statusFlags,SREG ; Save status flags

in      r16,SPDR
cpi     bytesToGo,3
brne    SEND_HI_V_BYTE
dec     bytesToGo
lds     r17,ADCL         ; The ADCL and ADCH registers must be read
lds     r16,ADCH         ;   together in this order!

out     SPDR,r16         ; Send voltage low byte first.
rjmp    END_SPI_XFER

SEND_HI_V_BYTE:
cpi     bytesToGo,2
brne    SEND_LAST_BYTE
dec     bytesToGo
out     SPDR,r17         ; Send voltage high byte last.
rjmp    END_SPI_XFER

SEND_LAST_BYTE:
cpi     bytesToGo,1
brne    END_SPI_XFER
out     SPDR,AD_Done
cpi     AD_Done,0x00
breq    END_SPI_XFER            ; If not yet finished skip cleaning up.

ldi     r16,(0<<ADEN)|(0<<ADSC)|(0<<ADATE)|(1<<ADIF)|(0<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
sts     ADCSRA,r16              ; Disable D/A converter

END_SPI_XFER:
out     SREG,statusFlags ; Restore status flags
reti
```

RESET Routine

The interrupt jump table now has two active entries besides `RESET`, one for when `PINB2` changes from “lo” to “hi” or from “hi” to “lo” (`PCINT0`) which jumps to `OUTPUT_REQUEST`, and the other when a byte has been sent out the SPI bus (`SPI_STC`) which jumps to `SPI_XFER_DONE`.

To start off the RESET routine, the stack pointer is set:

```
ldi     r16,high(RAMEND) ; Initialize stack pointer
out     SPH,r16
ldi     r16,low(RAMEND)
out     SPL,r16
```

We need to have `PINB2` set as input with a pull-up resistor keeping it at “1” unless pulled down by the `GPIO07` pin on the Pi to which it is jumpered. This involves two steps near the start of the `RESET` code.

First, the `MCUCR` register listed on page 624 of the ATmega328P datasheet at I/O port `0x35` (memory address `0x55`) is set with details described on pages 44, 69, and 92. The three lines of code

```
lds     r16,MCUCR               ; Enable pull-ups for all input ports without changing other settings
andi    r16,~(1<<PUD)           ;    in the MCUCR (microcontroller unit control register).
sts     MCUCR,r16
```

keep the existing bits of the `MCUCR` register the same except for the `PUD` bit which is forced to be “0”. This enables all pull-ups on pins that are (soon to be) specifically configured as inputs.

Next, the four SPI lines need to be set up. They are alternate functions of `PINB2` ($\overline{\texttt{SS}}$, SPI select when pulled low), `PINB3` (`MOSI`), `PINB4` (`MISO`), and `PINB5` (`SCK`, SPI clock) as explained on page 84. The instructions

```
cbi     DDRB,DDB2               ; Set PINB2 direction as input to receive the Chip Select signal
sbi     PORTB,PINB2             ; Set it to 1 initially.
```

set PINB2 for input with its pull-up enabled as described in the second row of Table 14-1 of section 14.2.3 (page 78). The instructions

```
sbi     DDRB,DDB4               ; Set SPI signal directions - PINB4 for MISO, PINB3 for MOSI
cbi     DDRB,DDB3
```

set the MISO and MOSI pin directions. (Some of these are automatically when SPI is enabled. See page 164 top.)

PINB1 is used to control an LED and is set for output starting with a “0” value.

```
sbi     DDRB,DDB1              ; Set PINB1 to be used to toggle an LED to signal activity.
cbi     PORTB,PORTB1           ; It will be turned on during data transmission to Raspberry Pi.
```

Initialize the AD_Done register to zero.

```
clr     AD_Done
```

The A/D converter which monitors the voltage of interest with its reference set to external and set the input multiplexer to use PINC1 (ADC1).

```
ldi     r16,(0<<REFS1)|(0<<REFS0)|(0<<ADLAR)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(1<<MUX0)
sts     ADMUX,r16
```

Register ADMUX shown on page 623 at address 0x7c is set according to the detailed description on pages 254-5. This enables ADC1 with AREF (pin 21 on the ATmega328P chip) as the reference voltage. In Table 14-6 on page 86, we see that ADC1 is an alternate function of pin PC1 (pin 24 on the ATmega328P chip). We connected PC1 to the midpoint of our voltage divider and wired AREF to the 3.3 V supply. We also must disable PC1 from being a digital input pin. That is done by the instructions

```
ldi     r16,(1<<ADC1D)          ; Disable digital input for PINC1
sts     DIDR0,r16
```

This sets the ADC1D bit of register DIDR0 as shown on page 623 at address 0x7e. Details are given in Section 24.9.5 (page 257-8).

Power is then enabled to the SPI and A/D system and disabled to other systems by doing

```
ldi     r16,(1<<PRTWI)|(1<<PRTIM2)|(1<<PRTIM0)|(1<<PRTIM1)|(0<<PRSPI)|(1<<PRUSART0)|(0<<PRADC)
sts     PRR,r16
```

Next, PINB2 is setup to allow the Pi GPIO08 pin connected to PINB2 to cause a pin-change interrupt and thereby start a measurement.

```
ldi     r16,(1<<PCINT2)         ; Enable pin change interrupt on PINB2 in pin change mask 0
sts     PCMSK0,r16
ldi     r16,(0<<PCIE2)|(0<<PCIE1)|(1<<PCIE0)
sts     PCICR,r16
```

There are only three pin change interrupts so a mask PCMSK0 (page 624, address 0x6b, details on page 75) is used to specify that the PCINT2 alternate function of PINB2 is going to cause interrupt PCINT0 to trigger.

Finally, the SPI system must be configured by doing

```
ldi     r16,(1<<SPIE)|(1<<SPE)|(0<<DORD)|(0<<MSTR)|(0<<CPOL)|(0<<CPHA)|(0<<SPR1)|(0<<SPR0)
out     SPCR,r16
clr     r16
out     SPSR,r16
```

This enables the SPI system, enables it to produce interrupts, and sets its clock frequency, polarity, and phase. SPCR and SPSR are listed on page 624 at addresses 0x2c and 0x2d with details given in Section 19.5 on pages 169-171.

The RESET instructions then sett the sleep mode to ADC noise reduction

```
ldi     r16,(0<<SM2)|(0<<SM1)|(0<<SM0)|(1<<SE)  ; Go to ADC noise reduction sleep while waiting for interrupts.
sts     SMCR,r16
```

Globally enable interrupts

```
sei                             ; Enable interrupts globally.
```

And go into a loop waiting for the interrupts

```
GOODNIGHT:
sleep
rjmp    GOODNIGHT
```

OUTPUT_REQUEST Routine

When PINB2 is changed by GPIO07 of the Pi, the PCINT0 interrupt is triggered, the ATmega328P sleep is interrupted, and execution jumps to the OUTPUT_REQUEST routine. There, the status flags are saved by

```
in      statusFlags,SREG
```

PINB1 is toggled to change an LED

```
sbi     PINB,PINB1              ; Change state of LED
```

empty the SPI data register

```
in      r16,SPDR
```

PINB2 is then checked to see if it changed to "1" which should cause a start of D/A conversion.

```
in      r16,PINB                ; Check state of SPI chip select pin
andi    r16,(1<<PINB2)          ; Enable SPI interrupts if low,
breq    END_OUTPUT_REQUEST
```

If it has not changed to "1", the output request ends. If it has changed to "1", it is necessary to start a D/A conversion or if one is started but not yet completed, to end the output request.

```
lds     r16,ADCSRA              ; Check if D/A has is enabled, if not enable it
andi    r16,1<<ADEN
brne    DA_ALREADY_ENABLED
```

ADCSRA and ADCSRB are on page 623 at addresses 0x74 and 0x74 with details on pages 255-258. The D/A is enabled by setting bit ADEN to "1" in ADCSRA. The ACME bit in ADCSRB is explained on page 239. It needs to be "0" for D/A operation. It, in fact, defaults to "0" so this step is not required.

```
ldi     r16,(0<<ACME)           ; Set ADC for single-conversion mode
sts     ADCSRB,r16
```

To enable the D/A, use its single conversion mode, not enable D/A complete interrupt (we will check its flag, ADIF instead), and to set a frequency division factor of 128 for the ADC prescaler, we do the following:

```
; Enable A/D, start a single conversion, without interrupt, and set frequency division factor to 128.
ldi     r16,(1<<ADEN)|(1<<ADSC)|(0<<ADATE)|(0<<ADIF)|(0<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
sts     ADCSRA,r16
rjmp    DA_BUSY
```

When the D/A has already been started, we need to check if it has completed by looking at the ADIF flag of the ADCSRA register, being careful to not change any of the bits of ADCSRA:

```
DA_ALREADY_ENABLED:
lds     r16,ADCSRA              ; Check if D/A conversion is still in progress
andi    r16,1<<ADIF
breq    DA_BUSY
```

If the D/A has completed its conversion, we set our AD_DONE flag to 0x01 and set the bytesToGo value to 3:

```
ldi     AD_Done,0x01            ; D/A conversion is complete
ldi     bytesToGo,3
rjmp    END_OUTPUT_REQUEST
```

Otherwise, we end and wait for another output request by restoring the processor flags:

```
DA_BUSY:
ldi     AD_Done,0x00            ; D/A conversion is still in progress

END_OUTPUT_REQUEST:
out     SREG,statusFlags ; Restore status flags
reti
```

SPI_XFER_DONE Routine

Like always, this interrupt routine must store the current processor flags

```
in      statusFlags,SREG
```

Since SPI interrupts were enabled in the RESET routine, every transfer attempt from the Pi will cause the OUTPUT_REQUEST routine to be run. This interrupt handler will be run after each byte pair is transferred, one byte coming from the Pi and simultaneously a byte going out to the Pi. The incoming byte from the Pi that triggered this interrupt can be placed in the r16 register with the instruction

```
in      r16,SPDR
```

The register SPDR is shown on page 624 at I/O port 0x2e (memory location 0x4e) with additional details in Section 19 (pages 162-171) and in particular on page 171. For this program, however, this step is unnecessary since we are not interested in the incoming byte; we are not looking for commands from the Pi. If we were, this would be parsed to find out what the Pi is requesting.

The next instructions send out bytes, one for each time this interrupt routine is called. The data appear in the ADCH and ADCL registers shown on page 624 at memory addresses 0x79 and 0x78 with details on page 256 and more explanation throughout section 24 of the ATmega328P datasheet.

An initial byte went out from the ATmega328P when the Pi first sent a byte to it, the next bytes to be sent come from ADCH (A/D conversion high byte), ADCL (A/D conversion low byte), and finally the value of AD_Done is sent to certify that the preceding bytes were from a valid voltage conversion. The AD_Done byte is then reset to 0x00. It will be set to 0x01 once again when the next A/D conversion is completed:

```
cpi     bytesToGo,3
brne    SEND_HI_V_BYTE
dec     bytesToGo
lds     r17,ADCL         ; The ADCL and ADCH registers must be read
lds     r16,ADCH         ;   together in this order!

out     SPDR,r16         ; Send voltage low byte first.
rjmp    END_SPI_XFER

SEND_HI_V_BYTE:
cpi     bytesToGo,2
brne    SEND_LAST_BYTE
dec     bytesToGo
out     SPDR,r17         ; Send voltage high byte last.
rjmp    END_SPI_XFER

SEND_LAST_BYTE:
cpi     bytesToGo,1
brne    END_SPI_XFER
out     SPDR,AD_Done
cpi     AD_Done,0x00
breq    END_SPI_XFER
```

The program is set up to only make a conversion each time the Pi initiates the transfer of a block of bytes, i.e. each time it pulls down the PINB2 line of the ATmega328P. So while waiting for the next request, the A/D converter is shut down after the AD_Done byte has been sent.

```
ldi     r16,(0<<ADEN)|(0<<ADSC)|(0<<ADATE)|(1<<ADIF)|(0<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
sts     ADCSRA,r16               ; Disable D/A converter
```

Unfortunately, it appears that the A/D converter has already sampled the current voltage and will provide that (usually) stale value at the next request! So if the voltage is changed between requests from the Pi, the first reading provided afterwards will be an old value. A variation of this program would be to set the A/D into free-running mode instead of single conversion mode. Then the value would be very close to current value since each conversion happens very quickly.

The processor flags must be restored and followed by a return from this interrupt routine.

```
END_SPI_XFER:
out     SREG,statusFlags        ; Restore status flags
reti
```