

# DJB-AtomPro FAQ

David J. Brown  
Created 3-15-06  
Last Updated 4-30-10

## Table of Contents

Integer vs. Floating Point Performance.....	2
Converting Potentiometer/Analog Inputs to Semi-Log Feel .....	2
Using Interrupts to Detect Narrow Triggers.....	4
PSIM Input to Output Voltage Conversion .....	5
MIDI Input & Output (e.g. hardware serial) .....	5
Timer A Interrupts .....	6
Timer V Interrupts .....	7
Timer W Interrupts .....	7
Serial Ouput Using s_out.....	7
Using MIDI input for RS-232.....	8
I2COUT.....	8
Miscellaneous Compiler Nuances.....	9
Code Space Limitations .....	9
AtomPro28 Nuances.....	9
BasicMicro Studio .....	10

## Integer vs. Floating Point Performance

I decided to run a test on my PSIM to verify the performance of integer vs. floating point. I used a line of my code that displays DAC voltages on the LCD as the test case. I need to divide by 102.3 and 10.23 so I had coded this using integers by first multiplying and then dividing (as  $*10/1023$  and  $*100/1023$ ).

I measured the execution speed with my scope and the integer calculation took 115.6  $\mu$ S.

```
i1 var word
i2 var word

let i1 = 8
let i2 = ((511-((i1*1023)/10))*100)/1023 ;executes in 115.6  $\mu$ S
```

Doing this same calculation using floating point variables took 50% longer.

```
f1 var float
f2 var float

let f1 = 8.0
let f2 = ((511.0-((f1*1023.0)/10.0))*100.0)/1023.0 ;executes in 176  $\mu$ S
```

However, with floating point I don't need to do the extra multiplications. Simplifying the statement to dividing by 102.3 and 10.23 resulted in code that was only 12% slower than using integer variables.

```
f1 var float
f2 var float

let f1 = 8.0
let f2 = (511.0-(f1/102.3))/10.23 ;executes in 130.0  $\mu$ S
```

There really isn't much penalty of floating point over integer other than increased ram storage.

## Converting Potentiometer/Analog Inputs to Semi-Log Feel

These transforms adjusts the range of my input potentiometers to give a more semi-log feel. These work well for controlling frequencies and delays where you need a fine control at low settings.

Let  $y = f(x)$  where  $x$  represents a 10 bit value corresponding to the input of

a linear control, and where  $y$  represents a transform of that value which is piecewise continuous and covers the 10 bit range

The simplest conversion is a two segment transform:

$$\begin{aligned} y &= x/2 && \text{for } x \text{ in the range of } 0 \text{ to } 682 \\ y &= 2*x - 1023 && \text{for } x \text{ in the range of } 683 \text{ to } 1023 \end{aligned}$$

Sample code for the two segment transform is:

```
adin pin_j1,in_j1           ;get input 10 bit value (in_j1)
if in_j1 > 681 then
    let in_j1 = 2*in_j1-1023 ;map 682-1023 to 341-1023
else
    let in_j1 = in_j1/2      ;map 0-681 to 0-340
endif
```

I have also used a three segment transform. This smooths the middle of the response with a linear segment. This is pretty simple to implement and feels quite well for time settings:

$$\begin{aligned} y &= x/2 && \text{for } x \text{ in the range of } 0 \text{ to } 511 \\ y &= x - 256 && \text{for } x \text{ in the range of } 512 \text{ to } 767 \\ y &= 2*x - 1023 && \text{for } x \text{ in the range of } 768 \text{ to } 1023 \end{aligned}$$

Sample code for the three segment transform is:

```
adin pin_j1,in_j1           ;get input 10 bit value (in_j1)
if in_j1 > 767 then
    let in_j1 = 2*in_j1-1023 ;map 768-1023 to 512-1023
elseif in_j1 > 511
    in_j1 = in_j1-256        ;map 512-767 to 256-511
else
    let in_j1 = in_j1/2      ;map 0-511 to 0-255
endif
```

I also used a four segment transform optimized for more control at low values:

$$\begin{aligned} y &= x/4 && \text{for } x \text{ in the range of } 0 \text{ to } 255 \\ y &= x/2 - 64 && \text{for } x \text{ in the range of } 256 \text{ to } 511 \\ y &= x - 320 && \text{for } x \text{ in the range of } 512 \text{ to } 863 \\ y &= 3*x - 2048 && \text{for } x \text{ in the range of } 864 \text{ to } 1023 \end{aligned}$$

Sample code for the four segment transform is:

```
adin pin_j1,in_j1           ;get input 10 bit value (in_j1)
if in_j1 > 863 then
    let in_j1 = 3*in_j1-2048 ;map 864-1023 to 546-1023
elseif in_j1 > 511
```

```

        in_j1 = in_j1-320          ;map 512-863 to 192-543
elseif in_j1 > 255
        let in_j1 = in_j1/2-64    ;map 256-511 to 64-191
else
        let in_j1 = in_j1/4      ;map 0-255 to 0-63
endif

```

This can be simplified to a three segment transform by eliminating the second segment and extending the range of the neighboring segments. This provides more control at low values:

```

y = x/4          for x in the range of 0 to 426
y = x - 320     for x in the range of 427 to 863
y = 3*x - 2048  for x in the range of 864 to 1023

```

Sample code for the three segment transform with lower range is:

```

adin pin_j1,in_j1          ;get input 10 bit value (in_j1)
if in_j1 > 863 then
        let in_j1 = 3*in_j1-2048 ;map 864-1023 to 546-1023
elseif in_j1 > 426
        in_j1 = in_j1-320        ;map 427-863 to 107-543
else
        let in_j1 = in_j1/4      ;map 0-426 to 0-106
endif

```

## Using Interrupts to Detect Narrow Triggers

P8 can support edge interrupts. You have to reprogram P8 and enable IRQ1. The default is for falling-edge interrupts.

```

let pmr1 = pmr1|%00100000          ;set PMR to enable irq1 on P8

```

You can change this to rising-edge interrupts for a positive trigger. An active high signal which is normally low will cause an interrupt when enabled. This can be eliminated by clearing the pending interrupt prior to enabling interrupts. Add the next two lines to select rising-edge interrupts:

```

let ieqr1 = ieqr1|%00000010        ;set irq1 to rising edge
let irr1 = irr1&%11111101          ;clear any pending interrupt

```

Then enable the IRQ1 interrupts:

```

oninterrupt irq1int,aux_isr
enable irq1int                      ;enable edge interrupts

```

Remember to resume from your interrupt service routine.

```
aux_isr:  
    resume
```

## PSIM Input to Output Voltage Conversion

The PSIM inputs are 10 bits (e.g. 0 to 1023) referenced to 10 volts. The PSIM outputs are 12 bits (e.g. 0 to 4095) referenced to 10.666 volts.

To convert an input to an output you need to multiply by 4 (10 bit to 12 bit conversion) and multiply by 10/10.666 (reference conversion).

This equals 3.7502 which is reasonably close to 15/4 which keeps the math to integer for maximum speed.

## MIDI Input & Output (e.g. hardware serial)

The hserin and hserout commands are poorly documented. The enablehserial command enables an interrupt-driven serial input and output subsystem. Both input and output are supported with 128 byte circular buffers. These routines are very fast.

Sample code to initialize the subsystem for MIDI:

```
enablehserial  
sethserial h31200,h8databits,hnoparity,h1stopbits
```

*Do not set the serial transmit P15 direction to output.* Doing so will cause a glitch when the serial subsystem is initialized. This will cause one garbage character to be transmitted. Instead, set the serial transmit P15 direction to input. *(Note - setting P15 to an output in the latest BMIDE or Studio software inhibits all serial output. It must be set to an input in order to function correctly).* The enablehserial command will set the correct direction.

Sample code to send Midi output:

```
hserout [$90,64,64]
```

Sample code to check Midi input:

```
get_midi:
```

```

    hserin 0,no_midi,[midi_data]           ;see if data but don't wait
    if midi_data=$fe then get_midi        ;ignore active status
    let rcx_data_flg=1                    ;set data returned flag
    return
no_midi:
    let rcx_data_flg=0                    ;set empty flag
    return

```

Sample code to wait for Midi input:

```

wait_midi:
    hserin 0,wait_midi,[midi_data]       ;wait for data
    if midi_data=$fe then wait_midi      ;ignore active status
    return

```

There is also a way to get status of the hserial buffers. I have not validated the use of any of these commands:

```

hserstat 0,label                         ;clear input buffer, label not used
hserstat 1,label                         ;clear output buffer, label not used
hserstat 2,label                         ;clear both buffers, label not used
hserstat 3,label                         ;goto label if data is in input buffer
hserstat 4,label                         ;goto label if no data is in input buffer
hserstat 5,label                         ;goto label if data is in output buffer
hserstat 6,label                         ;goto label if no data is in output buffer

```

## Timer A Interrupts

Timer A has rather limited prescalar selections. The AtomPro2X uses a 16 MHz clock that is based on a resonator, not a crystal. These times are approximate as the resonator has a wide tolerance. This sample code will initialize Timer A for 512 uS interrupts:

```

    let tma=%00010110                    ;set Timer Mode Register
                                           ; prescalar S /32 = 500 KHz clock
                                           ; 256 counts = 1953 Hz for 512 uS
                                           ;measured at 511.25 uS

    oninterrupt timeraint, tmr_a_isr
    enable timeraint                       ;enable timer interrupt

```

Remember to resume from your interrupt service routine.

```

tmr_a_isr:
    resume

```

## Timer V Interrupts

The `oninterrupt timervint` command does not function in releases prior to BMIDE 8.0.1.7. It is functional in BMIDE 8.0.1.7 and Studio. The AtomPro2X uses a 16 MHz clock so this is my test code to generate 1 mS interrupts:

```
let tcrv0=%00001011           ;set Timer Control Registers
                                ;clear on compare match A
let tcrv1=%00000001           ;prescalar S /128 = 125 KHz clock
let tcora=125                  ;count to 125 for 1 mS
                                ;124 measured 998 uS on system

oninterrupt timervint_cmea, tmr_v_isr
enable timervint_cmea         ;this command generates errors
```

Remember to resume from your interrupt service routine.

```
tmr_v_isr:
    resume
```

## Timer W Interrupts

I generally use Timer W for interrupts. It has the most flexibility. The AtomPro2X uses a 16 MHz clock so this sample code will generate 1 mS interrupts:

```
let tmrw=%10001000           ;set Timer Mode Register to enable
let tcrw=%10110000           ;set Timer Control Register
                                ;clear on compare match A
                                ;16 MHz clock /8 prescalar S = 2 MHz
let gra=2000                  ;2 MHz / 2000 = 1 mS
                                ;2003 measured more accurate

oninterrupt timerwint_imiea, tm_isr
enable timerwint_imiea       ;enable timer interrupt
```

Remember to resume from your interrupt service routine.

```
tm_isr:
    resume
```

## Serial Output Using `s_out`

The serial programming port may be used as an additional serial input or output pin. You have to use the `serout` command which requires interrupts be disabled for correct timing. The `s_out` signal is connected to pin 2 of the RS-232 connector. The internal circuitry is designed for +/- 5 volts levels to be compatible with RS-232 levels. The -5 volts is derived from RS-232 data on pin 3. With no RS-232 cable connected, the `s_out` pin is not referenced to ground. Adding a 2K resistor between pin 2 (`s_out`) and pin 5 (gnd) on the RS-232 connector will provide a ground reference. There is no current specification on this pin so I was unable to validate the minimum resistor value. 2K allows reasonable signal integrity at 57,600 baud. `S_out` can be used for software serial communications with the following command:

```
serout s_out,i57600,[data]
```

I have not validated software serial input on `s_in`. I have also been unable to get `s_in` to function as a general digital input pin.

## Using MIDI input for RS-232

I needed a way to send RS-232 data to the AtomPro2X from my computer. I wanted to use interrupts so I did not use `s_in`. I made an adapter cable that would interface RS-232 from my computer to the MIDI input. This allows me to use the `hserial` system for interrupt input. Remember to modify the `sethserial` command for the correct baud rate.

The adapter cable is made to convert +12 volts (RS-232 levels on my computer) to 5 mA current loop. Wire up the RS-232 cable adapter as follows:

1. Wire the DB-9 pin 3 to 1.5K resistor to a diode to MIDI-In pin 4 (the cathode end connects to MIDI-In pin 4)
2. Wire the DB-9 pin 5 to MIDI-In pin 5 (ground)
3. Wire the DB-9 pin 7 to DB9 pin 8 (flow control loopback)
4. Wire the DB-9 pin 1 to DB-9 pin 4 and pin 6 (status loopback)

## I2COUT

For `BMIDE`, the command syntax is:

```
i2cout datapin,clockpin,{ErrLabel},Control,{Address},[data ... data]
```

I have found some peculiarities with this command. If an address is not specified, there will be 1 start bit and the first data byte (shifted left 1 bit) will be used for the address (the control byte is not used). If an address is specified, there will be a repeated start (e.g. 2 start bits) and the control byte will be used for the address. The address (second byte) will be used as the first byte of data.

```
i2cout p6,p7,i2cerr,($23<<1),$14,["h"] ;8 bit adr=$23, two start bits
```

```
i2cout p6,p7,i2cerr,($23<<1),$14,["hello"] ;8 bit adr=$23, two start bits
i2cout p6,p7,i2cerr,($23<<1),["h"] ;8 bit adr=$34 (1/2 of "h"), one start bit
i2cout p6,p7,i2cerr,($23<<1),["hello"] ;8 bit adr=$34 (1/2 of "h"), one start bit
```

Use this command format to send a single byte of data:

```
i2cout p6,p7,i2cerr,($adr<<1),[$adr<<1,"h"] ;duplicate address in data field
```

Use this command format to send multiple bytes of data:

```
i2cout p6,p7,i2cerr,($adr<<1),[$adr<<1,"hello"] ;duplicate address in data field
```

This has been corrected in Studio. See the Studio section for correct syntax.

## Miscellaneous BMIDE Nuances

1. Bytetable lengths must be an even number of bytes. If the length is odd the compiler will add an extra byte.
2. | (vertical pipe) will continue a long command to the next line as in this example:

```
p_len      bytetable  off_0,off_1,off_2,off_3,off_4,off_5,off_6,off_7,|
                                     off_8,off_9,off_A,off_B,off_C,off_D,off_E,off_F
```

3. The include command does not function. I have tried a variety of alternatives, none successful.

## Code Space Limitations

1. One of my large programs started to get near the 32K limit. I found that I got a verify error at address \$7C00. I can only assume that there is really only 31,744 bytes of code space and the last 1,024 bytes are reserved. BMIDE indicates free space based on a total code size of 32,768 bytes.

## AtomPro28 Nuances

1. The AtomPro28 contains a Power-On-Reset chip that will be active low ~360 mS at power-on. Unfortunately, it also drives the reset line active high so current limiting is required if this signal is grounded through a switch. A series resistor in the range of 22R~47R work well.
2. The additional four I/O pins have different numbers depending on whether they are analog inputs or digital (*Note: the pin numbers changed in BMIDE 8.0.1.7*)

For digital inputs/outputs, the additional pins are:

Pin No.	Digital I/O <i>BMIDE 8.0.1.0 and previous</i>	Digital I/O <i>BMIDE 8.0.1.7</i>
13	P16 (e.g. High P16)	(not verified)
14	P17	(not verified)
15	P18	(not verified)
16	P19	(not verified)

For analog inputs, the additional pins are:

Pin No.	Analog in <i>BMIDE 8.0.1.0 &amp; previous</i>	Analog in <i>BMIDE 8.0.1.7</i>
13 / AN7	P19 (e.g. Adin P19,var)	P16
14 / AN6	P18	P17
15 / AN5	P17	P19
16 / AN4	P16	P18

Note that the analog pin designations in BMIDE 8.0.1.7 now almost match the datasheet and the digital pin designations (AN5 and AN6 are swapped). There must be at least one analog input enabled at all times.

Note that if you force the BMIDE setting to AtomPro24, you cannot use P16-19 as digital inputs/outputs. You can however use the additional pins as analog inputs by using the pin designators P4-P7 in the Adin command (e.g. Adin P7,var will read pin 13). (*Note: this may also have changed in BMIDE 8.0.1.7.*)

3. BMIDE must be set to Auto or AtomPro28 to use P16-19.

## BasicMicro Studio

BMIDE 8.0.1.7 has been superseded by Basic Micro Studio which supports all products. The current release as of this document was beta 1.0.0.25.

The fastmsbpre (e.g. 4) function for shiftout was fixed in Studio 1.0.0.25. Releases previous to this will not function correctly.

BasicMicro changed both the i2cout and the i2cin command syntax in Studio and they behave much more intuitive. In addition, these commands now hang if there was no I2C hardware present (they used to fail and continue). You now need to use the error label for them to fail and continue.

The syntax for each command is now identical:

```
i2cin i2c_data,i2c_clk,control,[data]
```

```
i2cout i2c_data,i2c_clk,control,[data]
```

This means the i2cin command no longer sends an address. You must use an i2cout command to send the address followed by the i2cin command to read the data.

For example, the BMIDE command

```
i2cin p6,p7,($4c<<1),$00,[step_max,step_max]
```

needs to be modified for Studio to

```
i2cout p6,p7,($4c<<1),[$00]  
i2cin p6,p7,($4c<<1),[step_max,step_max]
```

For example, the BMIDE command

```
i2cout p6,p7,i2cerr,($adr<<1),[( $adr<<1),"h"]
```

needs to be modified to

```
i2cout p6,p7,i2cerr,($adr<<1),["h"]
```