Optimization of a Narrow Band Filter

Implementation On A Low Cost Microcontroller

Issues and Performance

By Erick L. Oberstar

Table of Contents

7. Appendix
    7.1. C / ASM Source Code
    7.2. Compiler Multiplication Source Listings
    7.3. Matlab Souce Files
    7.4. Referenced Materials

1. Project Summary

The purpose of this project was to investigate the issues relating to the implementation optimizations of a digital filter on a low cost microcontroller platform rather than an expensive and special purpose Digital Signal Processor (DSP) system. In particular a 1KHz center frequency, 500 Hz bandwidth narrow band filter was implemented. Issues relating to algorithm optimization, fixed point mathematics, sampling rate, and signal reconstruction were observed and investigated. Overall system performance was also observed.

2. Atmel AVR Summary

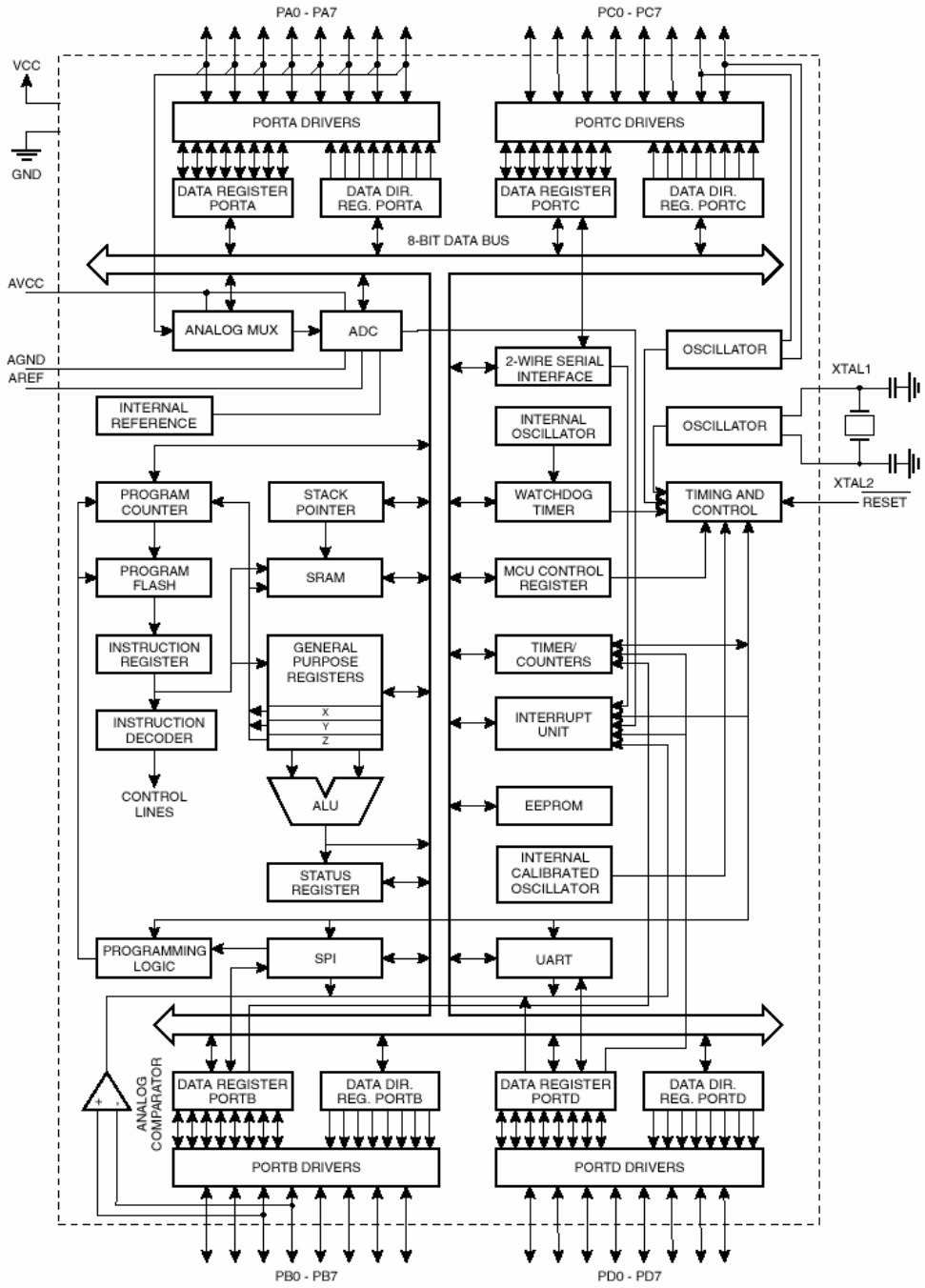
Atmel's AVR microcontrollers have a RISC core running single cycle instructions and a well-defined I/O structure that limits the need for external components. Internal oscillators, timers, UART, SPI, pull-up resistors, pulse width modulation, ADC, analog comparator and watch dog timers are some of the features in AVR devices.

AVR instructions are tuned to decrease the size of the program whether the code is written in C or Assembly. With on-chip in-system programmable Flash and EEPROM, the AVR is a reasonable choice to optimize for cost and get products to market quickly.

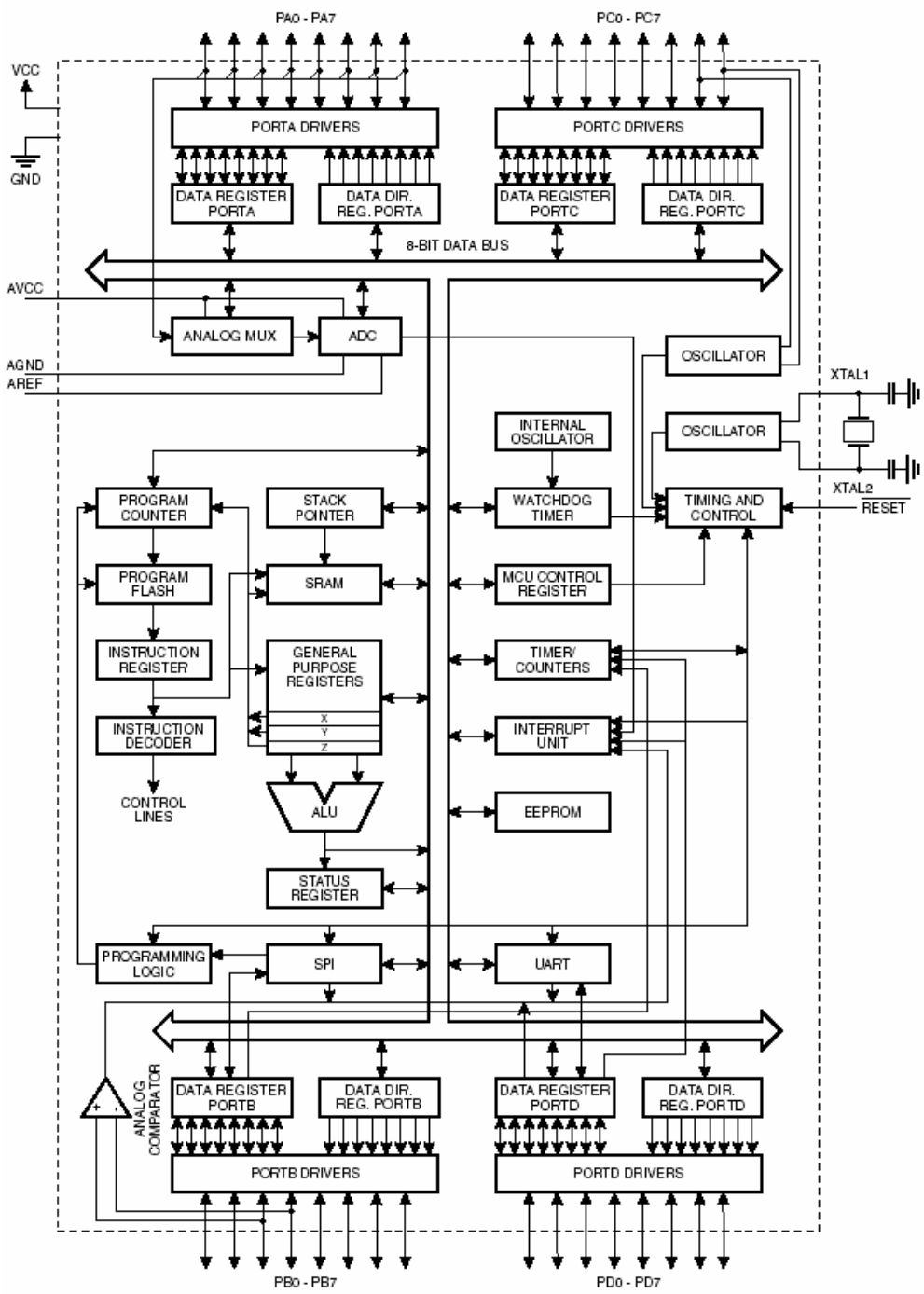
Two parts were selected to implement a simple 2^{nd} order digital filter on, the AT90S8535, and the ATmega163. The AT90S8535 and the ATmega163 belong to the Atmel AVR family of 8-bit RISC microcontrollers. Both parts run up to 8MHz. One has 16KB of FLASH and the other only 8K (AT90S8535) of Program memory. Both have 512 bytes of EEPROM (Nonvolatile Data Memory). The ATMEGA163 has 1KB of SRAM and the AT90S8535 only has 512 bytes of SRAM. Both have 32 I/O lines, 17 hardware interrupts, 3 counter timers, an 8 channel 10-bit A/D, and various other on chip peripheral modules. Key features used by the filter project were: one counter timer for interrupt service routine timing; A/D converter for sampling an analog waveform; an external D/A converter for analog output; a hardware integer multiply(on the Atmega163 only); and a “C friendly” instruction set.

Further feature specifications are available by following this link (ATmega163) or this link(AT90S8535) to the corresponding Atmel datasheets. The processor block diagram for the ATmega163 follows below:

Block Diagram



The processor block diagram for the AT90S8535 follows below:



For additional information follow this link to an Atmel Corporation internal AVR training Power Point presentation located on the accompanying CD.

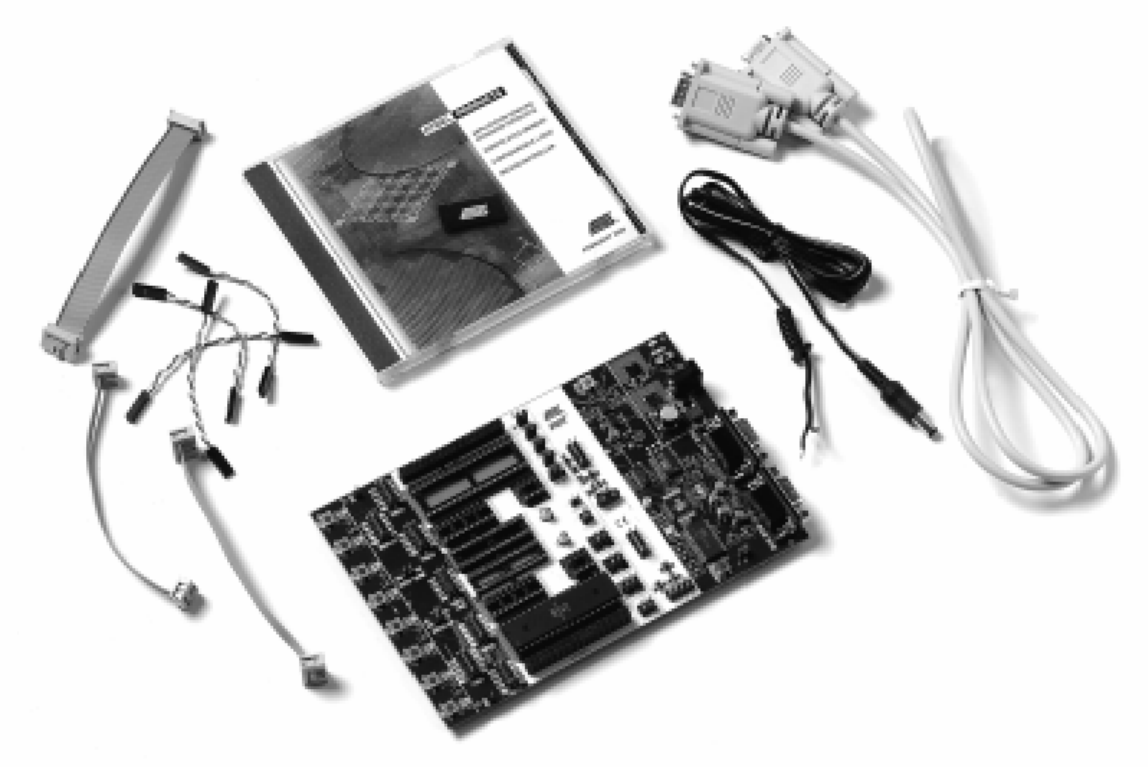
3. Development Systems

3.1 Hardware - Atmel STK500 Starter Kit

The Atmel STK500 is a complete starter kit and development system for the AVR Flash microcontroller from Atmel Corporation. It is manufactured and sold by Atmel for the purposes of part evaluation and prototype development. The entire STK500 users guide can be found on this cd by following this link. The features for the STK500 listed below are from the AVR STK500 User Manual.

Features

- **AVR Studio ® Compatible**
- **RS232 Interface to PC for Programming and Control**
- **Regulated Power Supply for 10 - 15V DC Power**
- **Sockets for 8-pin, 20-pin, 28-pin and 40-pin AVR Devices**
- **Parallel and Serial High-voltage Programming of AVR Parts**
- **Serial In-System Programming (ISP) of AVR Parts**
- **In-System Programmer for Programming AVR Parts in External Target System**
- **Reprogramming of AVR Parts**
- **8 Push Buttons for General Use**
- **8 LEDs for General Use**
- **All AVR I/O Ports Easily Accessible through Pin Header Connectors**
- **Additional RS232 Port for General Use**
- **Expansion Connectors for Plug-in Modules and Prototyping Area**
- **On-board 2-Mbit DataFlash ® for Nonvolatile Data Storage**



AVR Studio, version 3.53 or newer, supports the STK500. For up-to-date information on this and other AVR tool products, please read the document "avrtools.pdf" (AVR Studio Users Guide). The newest version of AVR Studio, "avrtools.pdf", and the user guide can be found in the AVR section of the Atmel Web site.

3.2 Software Development System

Software was developed using a pair of applications. Application source code was developed using a third party C compiler called CodeVisionAVR (CVAVR). Source code debugging was performed using Atmel's Integrated Development Environment (IDE) called AVR Studio. The hardware platform was programmed from the CVAVR compiler IDE using the AVR Studio programmer.

3.2.1 CodeVisionAVR (CVAVR) C Compiler

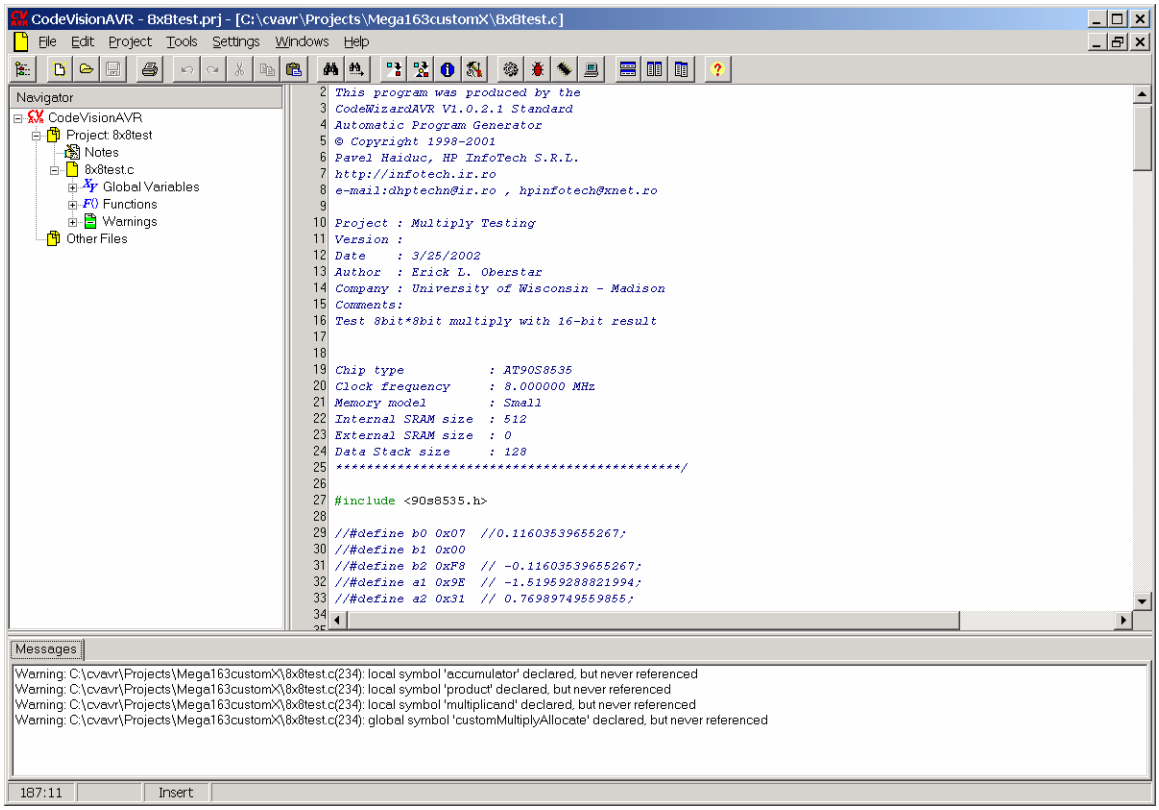
CodeVisionAVR is a third party C compiler targeted at the Atmel AVR line of RISC microcontrollers. It is build by HP InfoTech, a Romanian one-man company. CVAVR version 1.0.2.2e was used for source code development. A list of CVAVR features copied from the HP InfoTech website (http://www.hpinfotech.ro/) is included below:

Features

- 32 bit application, runs under Windows 95, 98, NT 4.0, 2000, XP
- Easy to use **Integrated Development Environment** and **C Compiler**
- Editor with auto indentation and keywords highlighting
- Supported data types: **bit**, **char**, **int**, **short**, **long**, **float**, **double**
- AVR specific extensions for:
  - Accessing the EEPROM & FLASH memory areas
  - Bit level access to I/O registers
  - Interrupt support
- Compiler optimizations:
  - Peep-hole optimizer
  - Advanced variables to register allocator, allows very efficient use of the AVR architecture
  - Common Block Soubroutine Packing, replaces repetitive code sequences with calls to subroutines
  - Loop optimization
  - Branch optimization
  - Subroutine call optimization
  - Cross-jumping optimization
  - Constant folding
  - Store-copy optimization
  - Dead code removing optimization
  - Two memory models: **TINY** (8 bit data pointers for chips with up to 256 bytes of RAM) and **SMALL** (16 bit data pointers for chips with more than 256 bytes of RAM) for better code efficiency
  - User selectable optimization for **Size** or **Speed**
- Possibility to insert assembler code directly in the C source file
- VERY EFFICIENT USE OF RAM: Constant character strings are stored only in FLASH memory and aren't copied to RAM, like in other compilers for the AVR
- C Source level debugging, with COFF symbol file generation, allows variable watching and the use of the Terminal I/O in Atmel's AVR Studio 3.53 Debugger
- Fully compatible with Atmel's In-Circuit Emulators
- Supported chips:
  - ATtiny22
  - AT90S2313
  - AT90S2323/2343
  - AT90S2333/4433
  - AT90S4414/8515
  - AT90S4434/8535
  - AT90S8534
  - ATmega603/103
  - ATmega64/128
  - ATmega161
  - ATmega163
  - ATmega323 (ATmega32)
  - ATmega8/16
  - FPSLIC AT94K10/20/40
  - AT43USB355
- Supplementary libraries for:
  - Alphanumeric LCD modules for up to 4x40 characters
  - Philips I²C Bus
  - National Semiconductor LM75 Temperature Sensor
  - Dallas DS1621 Thermometer/Thermostat
  - Philips PCF8563 and PCF8583 Real Time Clocks
  - Dallas DS1302 and DS1307 Real Time Clocks
  - Dallas 1 Wire protocol
  - Dallas DS1820/DS1822 Temperature Sensors

    - SPI
    - Power management
    - Delays
- Built-in **CodeWizardAVR Automatic Program Generator**, allows you to write in a matter of minutes all the code needed for implementing the following functions:
    - External memory access setup
    - Chip reset source identification
    - Input/Output Port initialization
    - External Interrupts initialization
    - Timers/Counters initialization
    - Watchdog Timer initialization
    - UART initialization and interrupt driven buffered serial communication with the following parameters: 7N2, 7E1, 7O1, 8N1, 8N2, 8E1 and 8O1
    - Analog Comparator initialization
    - ADC initialization
    - SPI Interface initialization
    - I²C Bus, LM75 Temperature Sensor, DS1621 Thermometer/Thermostat, PCF8563, PCF8583, DS1302 and DS1307 Real Time Clocks initialization
    - 1 Wire Bus and DS1820/DS1822 Temperature Sensors initialization
    - LCD module initialization
- Built-in **Serial Communication Terminal** for debugging RS232, RS422, RS485
- Built-in **In-System AVR Chip Programmer**, compatible with the Atmel **STK500**, Kanda Systems **STK200** and **STK300** development boards, Vogel Elektronic **VTEC-ISP**, Dontronics **DT006**, with automatic programming after successful compilation.
- Supported chips:
    - ATtiny22
    - AT90S2313
    - AT90S2323/2343
    - AT90S2333/4433
    - AT90S4414/8515
    - 90S4434/8535
    - AT90S8534
    - ATmega603/103
    - ATmega64/128
    - ATmega161
    - ATmega163
    - ATmega323 (ATmega32)
    - ATmega8/16

A screen grab of the compiler is shown below:



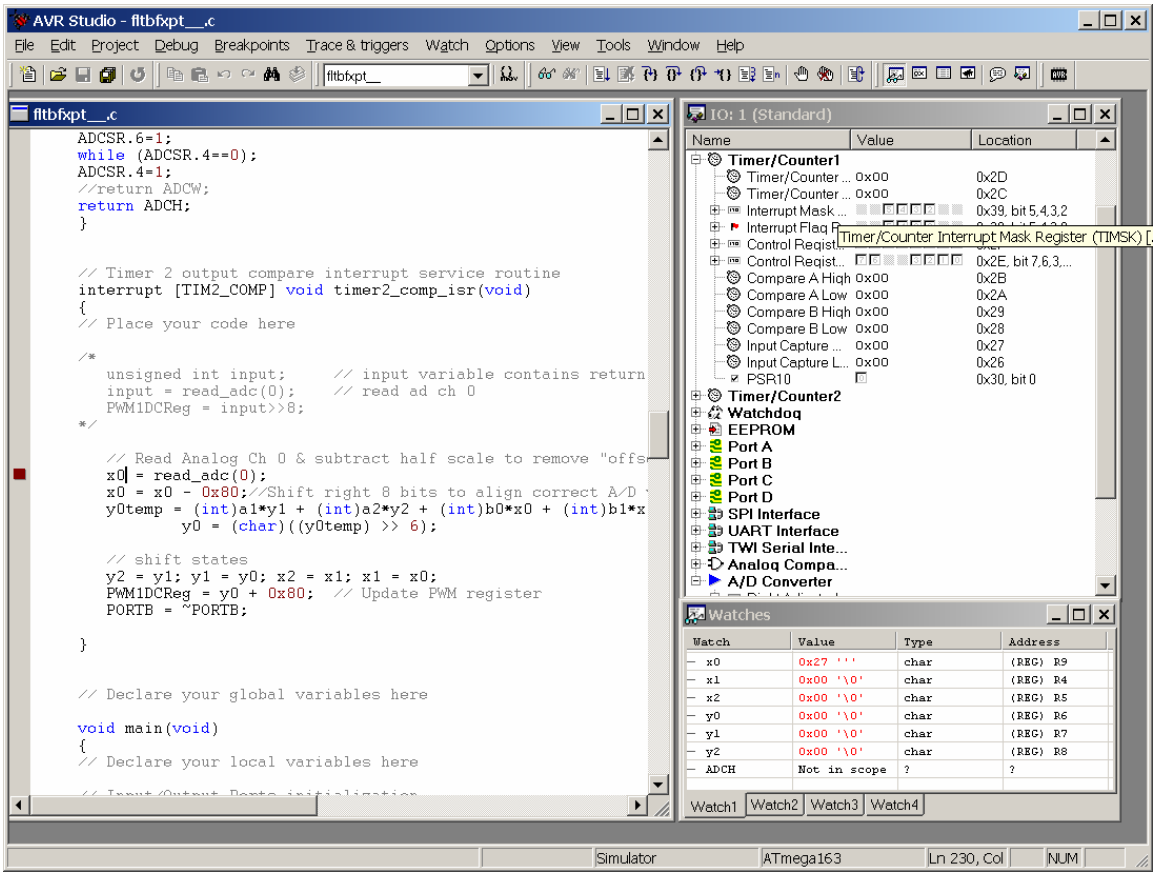
The full manual for the CVAVR compiler can be found online (http://www.hpinfotech.ro/).

3.2.2 Atmel AVR Studio

AVR Studio is a Win32 application built by Atmel to provide an IDE for assembly language code generation and simulation. It also provides an interface to Atmel's STK500 AVR®Flash MCU Starter Kit for part programming and hardware testing.

For the purpose of this project, AVR Studio was used primarily for C level source code simulation/debugging. The AVR Studio IDE can be used to step through code, watch variables, and stimulate registers as well as I/O pins in both assembly and C.

A screen shot of the AVR Studio IDE in a C source level debugging configuration is shown below:



The entire AVR Studio 3 User Guide can be found on the Atmel web site.

4. System Configuration

4.1 Analog Input

Both the ATmega163 and AT90S8535 feature a 10-bit successive approximation analog to digital converter. There are two modes of operation: (1) Free Running (2) Single Conversion. In Single Conversion Mode, the user must initiate each conversion. In Free Running Mode, the ADC is constantly sampling and updating the ADC Data Register. The fifth bit of the ADCSR register selects between these modes. Any pin of PORTA may be chosen as the ADC input via a multiplexer.

The ADC supports single ended conversion between 0 and 5 volts. The conversion time ranges between 65 and 260 microseconds. The ADC may operate up to 15kSPS at maximum resolution and up to 76kSPS at 8-bit resolution. The successive approximation circuitry requires an input clock frequency that may be chosen from pre-scaled system clock values in the ADC module. According to ATmega163 Data Sheet p.98, the ADC clock must be less than or equal to 1MHz to provide 8 bit resolution.

A worse case conversion rate is 25 ADC clock cycles. The conversion time must obviously be shorter than the sampling interval to produce relevant digital values. So 25 times the ADC clock period must be less than the sampling period or equivalently, the ADC clock frequency must be 25 times greater than the sampling rate. For a 12.048 kHz sampling rate, the ADC clock frequency must be greater than approximately .3012 MHz. An ADC clock frequency of 1MHz satisfies the upper boundary of the preceding paragraph and the lower boundary just mentioned. The A/D Specs are identical for the AT90S8535.

4.2 Analog Output

A common and inexpensive external quad D/A converter was used for analog reconstruction. Specifically the Maxim MAX506ACPP was used. The MAX506ACPP is a quad channel 8-bit D/A converter single supply part with ±1 lsb accuracy .The datasheet can be found on the Maxim web site at http://pdfserv.maxim-ic.com/arpdf/MAX505-MAX506.pdf.

4.3 Sampling Rate

The system utilized one of the counter timers with interrupt support on the both processors for deterministic input signal sampling, input signal filtering, and output signal updating. Counter Timer 2 was configured to use the full 8 MHz system clock in an output compare mode with the output disconnected. The timer was configured to clear it self and generate an interrupt on a compare match. The Timer 2 compare register was loaded with a hex value that results in a 12.048 KHz interrupt rate with an 8 MHz system clock.

5. DSP

5.1 Filter Design

In order to build a narrow-band digital filter, start with a simple second order continuous-time band pass filter:

$$H(s) = \frac{sG}{(s - s0)(s - s0^*)}$$

As a narrow-band approximation (B<<f0) place the poles as shown below:

$$s0 = -\pi B + j2\pi f0$$
$$s0 = -\pi B - j2\pi f0$$

where:

B = Desired Band Width (based on 3dB attenuation)
f0 =Desired Center frequency
G = scale factor to control over all filter gain

$$H(s) = \frac{sG}{s^2 - s(s0 + s0*) + s0s0*}$$

Let p = $s0 + s0* = -2\pi B$

Let q = $s0 * s0* = \pi^2 B^2 + 4\pi^2 f0^2$

$$H(s) = \frac{sG}{s^2 - ps + q}$$

Next apply the bilinear transform:

$$s = \frac{2}{T}\frac{(1 - z^{-1})}{(1 + z^{-1})}$$

T = sampling interval

$$H\left(\frac{2}{T}\frac{(1-z^{-1})}{(1+z^{-1})}\right) = \frac{\frac{2G}{T(\frac{4}{T^2} + \frac{2p}{T} + q)} - \frac{2G}{T(\frac{4}{T^2} + \frac{2p}{T} + q)} z^{-2}}{1 + \frac{2q - \frac{8}{T^2}}{(\frac{4}{T^2} + \frac{2p}{T} + q)} z^{-1} + \frac{\frac{4}{T^2} - \frac{2p}{T} + q}{(\frac{4}{T^2} + \frac{2p}{T} + q)} z^{-2}}$$

Equate the coefficients with the general second order digital infinite impulse response filter:

$$H(z) = \frac{b0 + b1z^{-1} + b2z^{-2}}{1 + a1z^{-1} + a2z^{-2}}$$

$$b0 = \frac{2G}{T(\frac{4}{T^2} + \frac{2p}{T} + q)} \qquad a1 = \frac{2q - \frac{8}{T^2}}{(\frac{4}{T^2} + \frac{2p}{T} + q)}$$

$$b1 = 0$$

$$b2 = \frac{-2G}{T(\frac{4}{T^2} + \frac{2p}{T} + q)} \qquad a2 = \frac{\frac{4}{T^2} - \frac{2p}{T} + q}{(\frac{4}{T^2} + \frac{2p}{T} + q)}$$

Prewarping:

$$s = \frac{2}{T}\frac{(1 - z^{-1})}{(1 + z^{-1})}$$

The bilinear transform does not produce a perfectly linear relationship between analog and digital frequencies especially for higher frequencies:

$$j\omega a = \frac{2}{T}\frac{(1 - e^{-j\omega d})}{(1 + e^{-j\omega d})}, \qquad \begin{aligned} s &= j\omega a \\ z &= e^{j\omega d} \end{aligned}$$

$$fa = \frac{\tan(\pi fdT)}{\pi T}$$

Therefore, in order to design for a center frequency (fd) and bandwidth (fd_max-fd_min) similar to that of the analog filter, the digital filter coefficients must be calculated with
f0 = fa and B=Ba = fa_max – fa_min.

Unity gain desired at center frequency:

$$\left|H(j2\pi fa)\right| = \frac{2\pi faG}{\pi Ba\sqrt{\pi^2 Ba^2 + 16\pi^2 fa^2}}$$

$$G = \frac{Ba}{2fa}\sqrt{\pi^2 Ba^2 + 16\pi^2 fa^2}$$

Difference Equation:

$$y[n] = -a1\,y[n-1] - a2\,y[n-2] + b0\,x[n] + b1\,x[n-1] + b2\,x[n-2]$$

The floating-point representation for the filter coefficients for a 500Hz wide 1KHz center frequency filter were calculated using the above equations in Matlab to be:

b0 = 0.11603539655267;
b1 = 0;
b2 = -0.11603539655267;
a1 = -1.51959288821994;
a2 = 0.76989749559855;

5.2 Fixed-point Representation

To more accurately construct the digital filter, floating point data and coefficient values should be used. However there is significant processor overhead required to perform floating-point calculations. Floating point overhead limits the effective sampling rate of the filter because interrupt service routine (ISR) takes extra time to execute thus lowering the highest possible ISR execution rate.

To improve mathematical throughput and increase the ISR execution rate (i.e. increase sampling rate) calculations are performed using two's complement signed fixed point representations. Q7 and Q6 8-bit fixed point representations for input samples and filter coefficients respectively, were chosen because the low cost microcontroller selected is natively 8-bit. Q7 numbers can represent fixed-point numbers ranging from –1 to 0.9921875 in increments 0.0078125 (-1 to 1 - 1/128). The 8-bit Q7 number bit weighting is shown below. The decimal place is between bits 6 and 7. Input samples are in a Q7 format.

s.	x	x	x	x	x	x	x
-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128

Eight bit Q6 numbers can represent fixed-point numbers ranging from –2 to 1.984375 in increments 0.015625 (-2 to 2 - 1/64). The Q6 representation bit weighting is shown below. The decimal place is between bits 5 and 6. Filter coefficients are in a Q6 format.

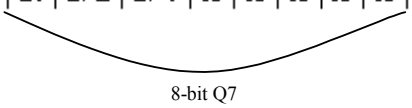
s	x.	x	x	x	x	x	x
-2	1	1/2	1/4	1/8	1/16	1/32	1/64

When a Q7 and Q6 number are multiplied (both 8 bit numbers) the result is a 16-bit Q13 number. Q13 numbers range from –4 to 3.9998779296875 in increments of 0.0001220703125 (-4 to 4 – 1/8192). The Q13 representation bit weighting is shown below.

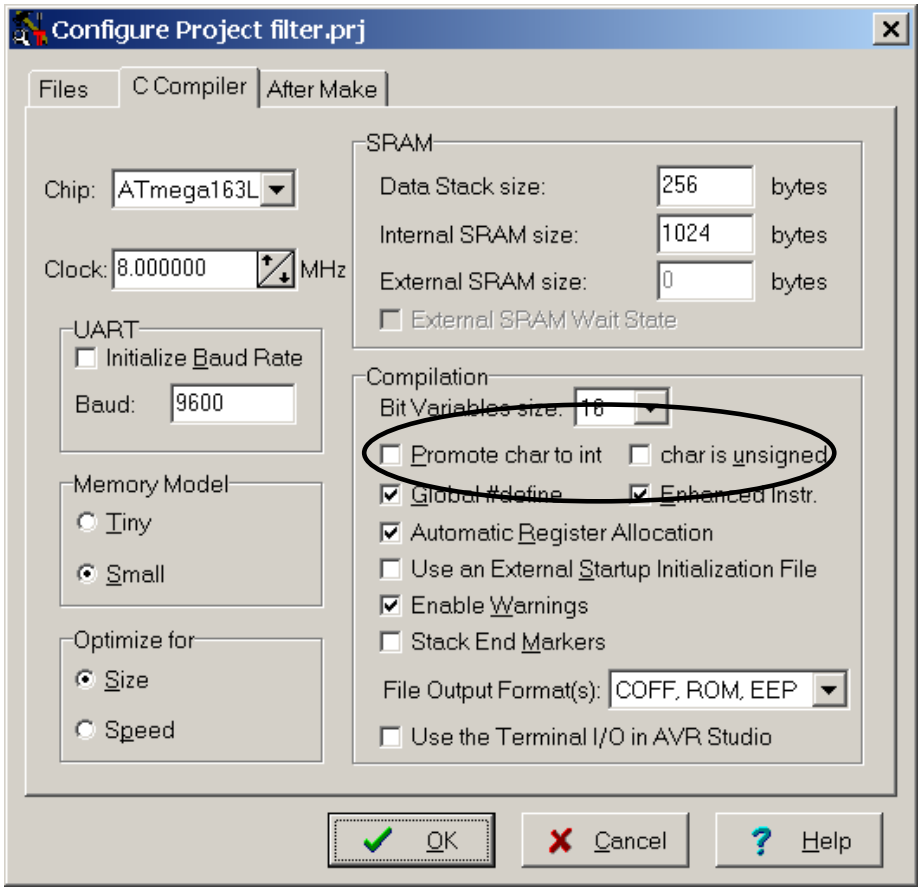
s	x	x.	x	x	x	x	x	x	x	x	x	x	x
-4	2	1.	1/2	1/4	x	x	x	x	x	x	x	x	x	1/8192

The 16-bit Q13 number is scaled back to a Q7 representation for the digital filter's output. This Q7 number is contained in the bit just left of the decimal place and the seven bits just below the decimal place. These Q7 bits are extracted by shifting the 16-bit Q13 number right six bits and selecting only the low byte of the 16-bit value. The resulting 8-bit Q7 number is shown below.

s	x	x.	x	x	x	x	x	x	x	x	x	x	x
-4	2	1.	1/2	1/4	x	x	x	x	x	x	x	x	x	1/8192



A critical detail is that the 8-bit Q7 & Q6 numbers must be a signed data type in C. This is important because when a product of a Q7 & Q6 number is calculated the 8-bit values must be sign extended to 16-bits to do the multiplication correctly. A point of caution: The CodeVision compiler contains switches to make "char" data types unsigned by default as well as to allow automatic promotion of "char" types to "int". See the below project configuration block showing these switches and the correct settings.

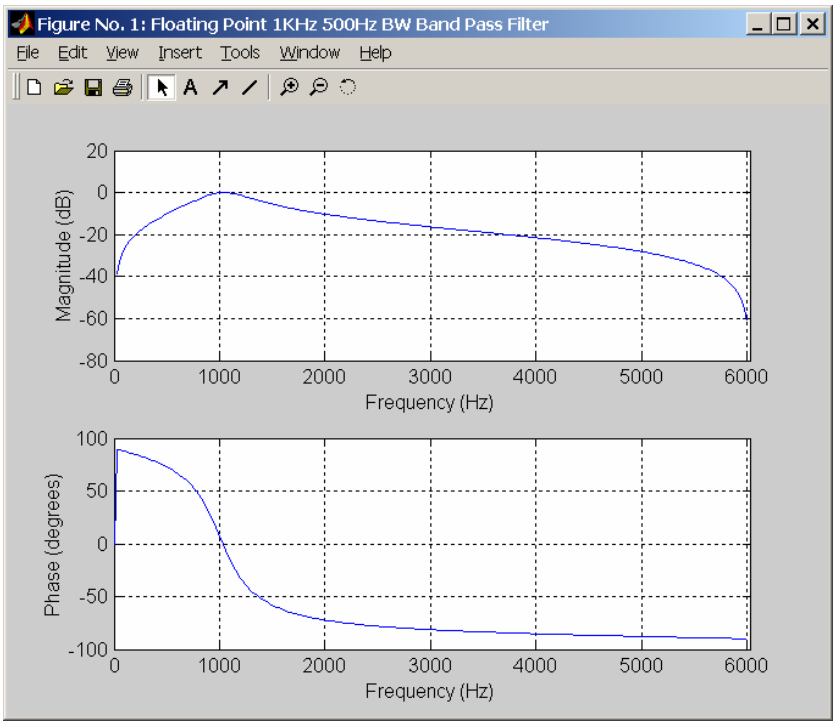


The Q6 fixed-point representation for the filter coefficients for a 500Hz wide 1KHz center frequency filter were calculated to be:

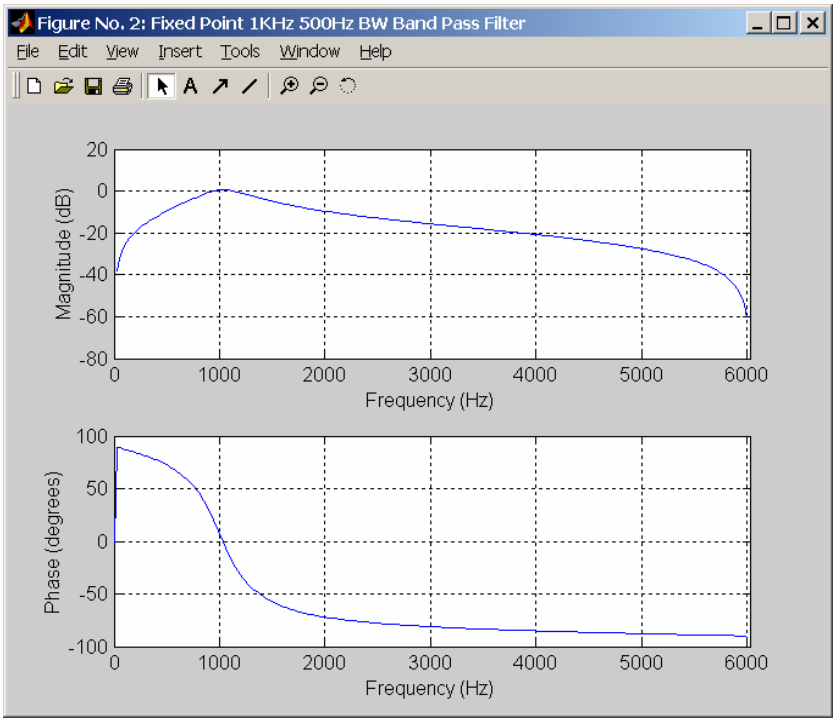
b0 = 0x07 = .125 //0.11603539655267;
b1 = 0x00 = 0;
b2 = 0xF8 = -.125 // -0.11603539655267;
a1 = 0x9E = -1.515625 // -1.51959288821994;
a2 = 0x31 = .765625 // 0.76989749559855;

5.3 Fixed-point Effects

Fixed-point representation affects several areas of the filter design/response. Since the filter coefficients are represented by 8-bit Q7 and Q6 fixed point format, the coefficients can only have quantum values. The worst-case 8-bit Q7 coefficients can be off by ±1/256. The worst-case 8-bit Q6 coefficients can be off by ±1/128. As the center frequency or bandwidth of the filter is lowered the fixed-point effects on the filter coefficients become more pronounced. Below is a Matlab plot showing a 1KHz center frequency, 500Hz bandwidth band pass filter whose frequency response was generated using floating-point coefficients.



Below is a Matlab plot showing a 1KHz center frequency, 500Hz bandwidth band pass filter who's frequency response was generated using the fixed point coefficients closest to the designed floating point filter coefficients. There are three apparent effects: amplitude scaling, a slight shift in center frequency, and a slight difference in the filter's bandwidth.



There may also be fixed point errors that are induced in the phase of the filtered signal.

6. Performance / Optimizations / Tradeoffs / Conclusion

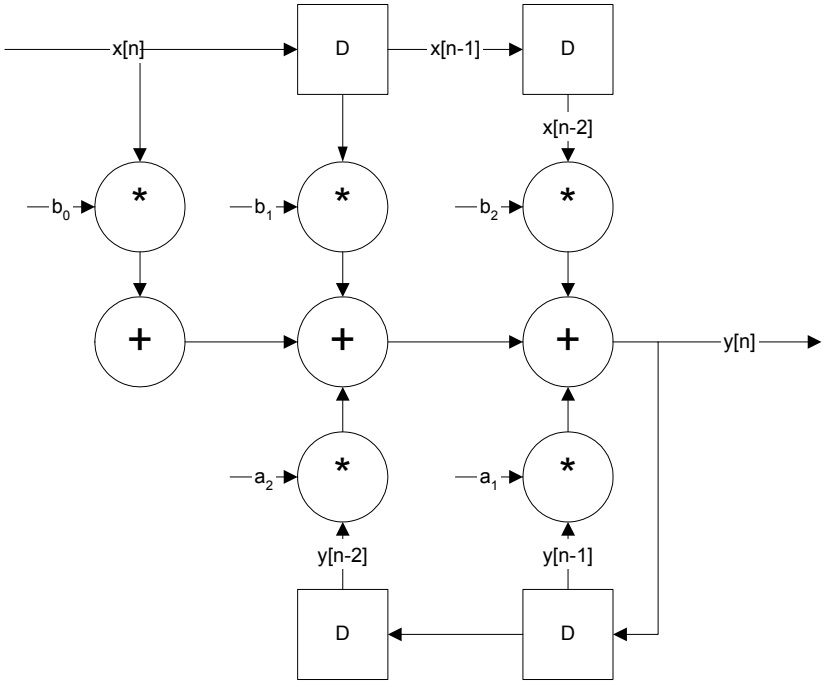
6.1. Algorithm Transformations - General Form V.S. Data Broadcast With/Without Hardware Multiplier

The base algorithm for the IIR filter implementation follows the form:

For the transfer function, $H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$,of a second order IIR digital filter, the base difference equation to implement follows the form:

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + y(n) + a_1 y(n-1) + a_2 y(n-2)$$

The "general form" for the signal flow graph (SFG) of a 2nd order IIR is as follows:



This dependence graph was implemented in "C" code as in the "general form" in single assignment form with fixed point coefficients:

```
// The 1KHz 500Hz bandwidth, filter Coefficients in Q6 fixed point format.
b0 = 0x07;      //0.11603539655267;
b1 = 0x00;      // 0.0
```

```
b2 = 0xF8;      // -0.11603539655267;
a1 = 0x9E;      // -1.51959288821994;
a2 = 0x31;      // 0.76989749559855;

//State initialization
x0 = 0x00; x1 = 0x00; x2 = 0x00;
y0 = 0x00; y1 = 0x00; y2 = 0x00;
y0temp = 0x0000;

while (1)
{
        x0 = read_adc(0);       // Sample signal from A/D Channel 0;

        x0 = x0 - 0x80;         // Subtract off DC offset

        // Filter Multiply and accumulate to Q13 Number
        y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;
        y0 = (char)((y0temp) >> 8);

        // shift states
        y2 = y1; y1 = y0; x2 = x1; x1 = x0;

        WRSingleDAC(y0 + 0x80, 0);          // Add back DC offset and put on D/A Channel 0
}
```

Where a1, a2, b0, b1, b2 are the signed 8-bit Q6 format filter coefficients. The values y0temp, y0, y1, y2, x1, and x2 are filter state variables containing current and past filter input and output values.

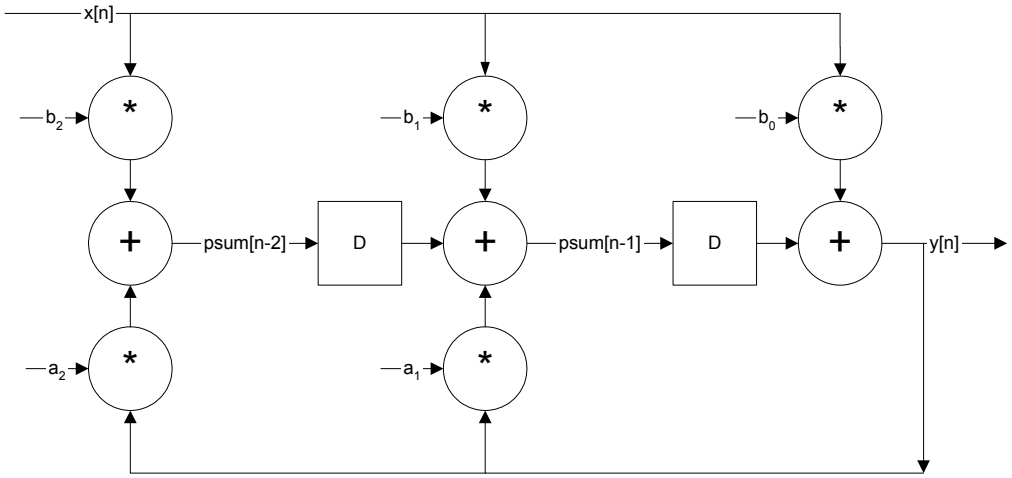
This general form of the IIR filter algorithm was compiled for both the ATmega163 and the AT90S8535 with the compiler option set to “Optimize for Speed” and again with “Optimize for Size” for comparison. The summary for this general form algorithm follows:

Optimize for Speed	Code Size (Words)	Critical Path (Cycles)
AT90S8535	133	1138
ATmega163	126	408

Optimize for Size	Code Size (Words)	Critical Path (Cycles)
AT90S8535	123	1173
ATmega163	126	408

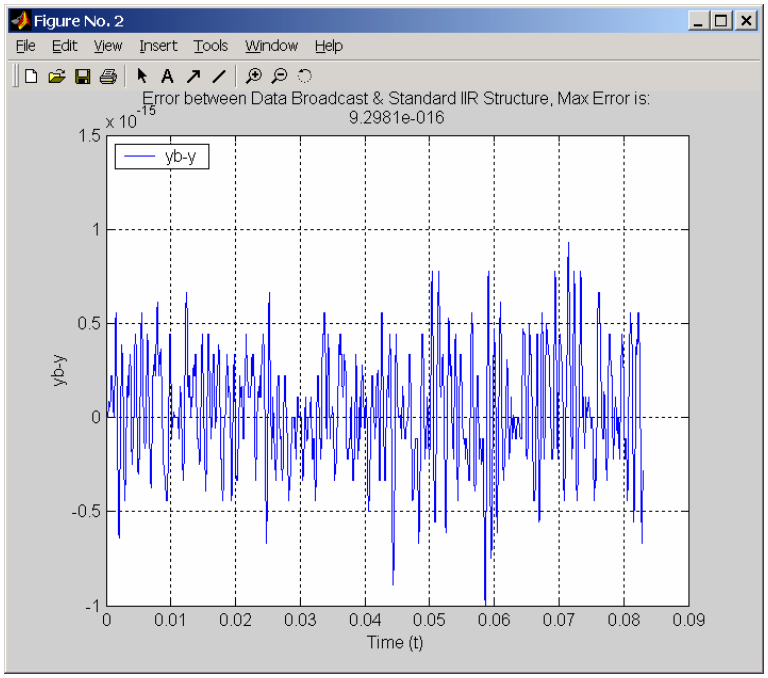
The source code for the general form algorithm can be found in the Appendix section 7.1.

However this is only one formulation of the SFG for this algorithm. An alternate form is the data broadcasting structure. The SFG for the data broadcast structure is shown as follows:

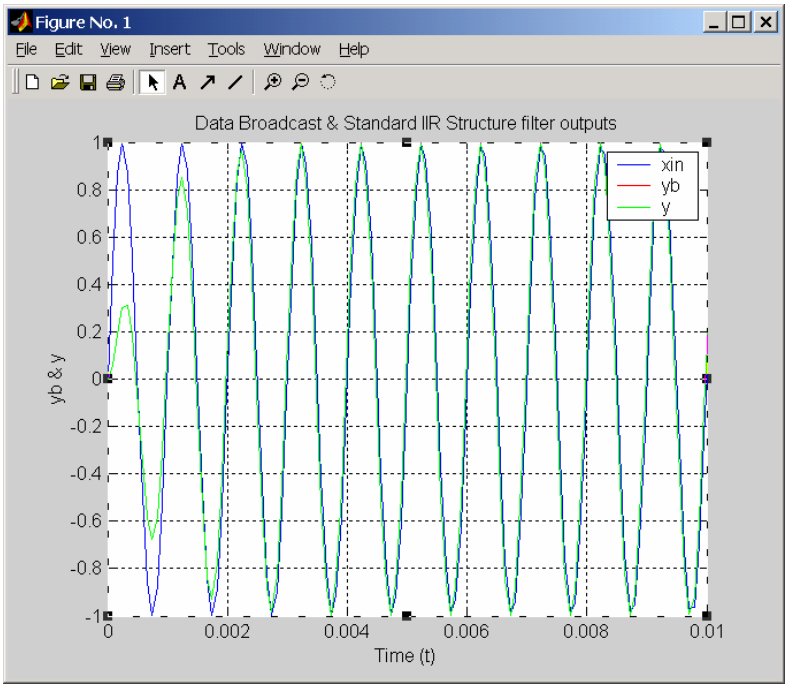


The most obvious difference in this SFG is the broadcasting of the same input to all the multiply operations and the associated reduction in registers by a factor of two.

The validity of this transformation was verified with Matlab by comparing the outputs for the given filter structures. This is shown in the below figures. Note the scale is on the order of zero given Matlab's precision. The Matlab script file that was written for this validation is found in the Appendix section 7.3.



IIR Form Error (Broadcast – General)



Filter output at 1KHz for both Broadcast and General forms

Since the "results" of Matlab simulation validated the algorithm transformation, C source level software was written implementing the transformed algorithm.

The C source implementation for the data broadcast structured IIR filter takes the form:

```
while (1)
{
        x0 = read_adc(0);

        x0 = x0 - 0x80;          // Subtract off DC offset

        // the current x0 and y0temp are broadcast to all required computations
        // Calculate the actual value for output based on current input and partial sum
        y0temp  = (int)b0*x0 + psum_n_m_1;
        // Calculate 1st partial sum
        psum_n_m_1 = (int)b1*x0 - (int)a1*y0temp + (int)psum_n_m_2;
        // Calculate 2nd partial sum
        psum_n_m_2 = (int)b2*x0 - a2*y0temp;

        y0 = (char)((y0temp) >> 8);   // Scale back down to 8 bits

        WRSingleDAC(y0 + 0x80, 0);          // Add back DC offset and put on D/A Ch 0
}
```

Where a1, a2, b0, b1, b2 are the signed 8-bit Q6 format filter coefficients. The 16-bit Q13 format values psum_n_m_1 and psum_n_m_2 are partial sum state variables containing past partial products.

This data broadcast form of the IIR filter algorithm was compiled for both the ATmega163 and the AT90S8535 with the compiler option set to "Optimize for Speed" and again with "Optimize for Size" for comparison. The summary for this data broadcast algorithm follows:

Optimize for Speed	Code Size (Words)	Critical Path (Cycles)
AT90S8535	132	1153
ATmega163	129	420

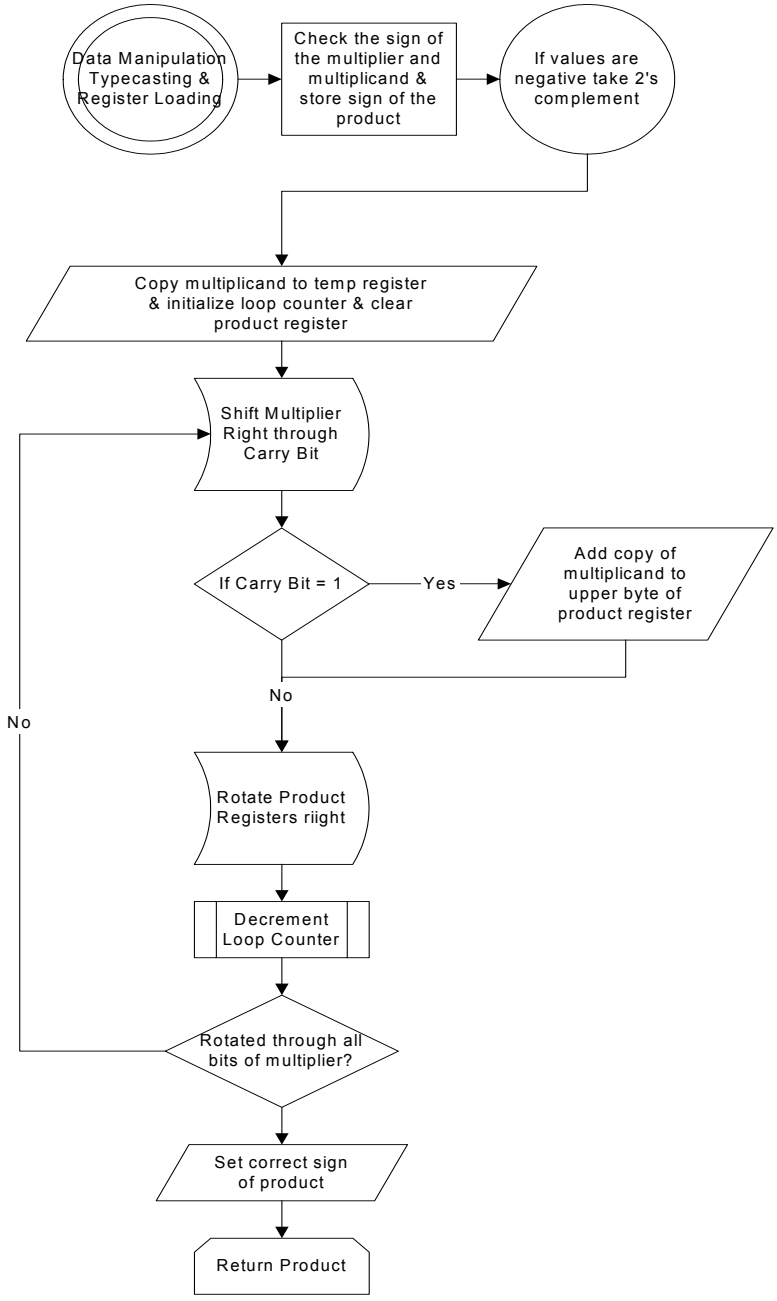
Optimize for Size	Code Size (Words)	Critical Path (Cycles)
AT90S8535	115	1202
ATmega163	113	441

The C source code for each compilation scheme is found in the appendix section 7.1.

6.2 Analysis of Compiler Generated Multiplication Routines

6.2.1 Software based multiplication

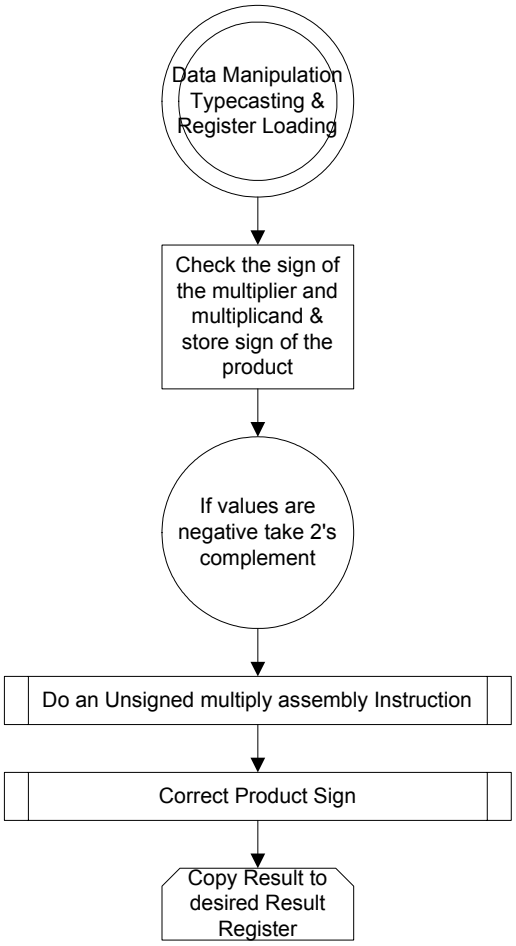
The primary source of critical path delays for the code running on the AT90S8535 is the multiplication overhead. The general-purpose multiplication subroutine that is generated by the compiler has a critical path of 213 cycles including the data manipulation (10 cycles), branch, and return overhead of 8-10 cycles depending on the storage location of the code. With a total of five multiplications in the IIR algorithm, the critical path is over 1000 cycles. The multiplication routine seems to be the best potential candidate for optimization. The flow chart for the general-purpose multiplication routine generated by the CodevisionAVR compiler follows:



The commented assembly listing for the 8x8 signed multiplication routine generated by the CodevisionAVR compiler is listed in the appendix section 7.2

6.2.1 Hardware based multiplication

The ATMEGA163 compiler generated signed multiplication routine includes some overhead as well. By default the compiler generates its own sign checking code, stores the products sign, and performs an unsigned multiplication assembly instruction. The flow chart for the compiler generated hardware multiplication routines follows:



The critical path of hardware multiplier based algorithm is 69 cycles including the data manipulation (10 cycles), branch, and return overhead of 8-10 cycles depending on the storage location of the code. The commented assembly listing for the hardware multiplier based 8x8 signed multiplication routine generated by the CodevisionAVR compiler is listed in the appendix section 7.2

6.3 Optimizations

6.3.1 Constant Substitution

The next optimization that was performed was constant substitution since the coefficients for the IIR filter are known quantities. The b1 coefficient is the first obvious substitution since b1 = 0 and essentially eliminates 1 multiply accumulate operation. The constant substitution for the general form IIR filter essentially modifies the C source code line:

```
//Filter Multiply and accumulate to Q13 Number
y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;
```

to the constant substitute form:

```
//Filter Multiply and accumulate to Q13 Number
y0temp = (int)0x9E*y1 + (int)0x31*y2 + (int)0x07*x0 + (int)0xF8*x2;
```

Constant substitution for the general form algorithm yielded an improvement of 20.7% (236 cycles) in critical path, and a decrease of 24.8% (33 instructions) in code size on the AT90S8535. Constant substitution for the general form algorithm yielded an improvement of 20.8% (85 cycles) in critical path, and a decrease of 23.0% (29 instructions) in code size on the ATMEGA163.

The constant substitution for the broadcast form IIR filter essentially modifies the C source code lines:

```
// Calculate the actual value for output based on current input and partial sum
y0temp  = (int)b0*x0 + psum_n_m_1;
// Calculate 1st partial sum
psum_n_m_1 = (int)b1*x0 - (int)a1*y0temp + (int)psum_n_m_2;
// Calculate 2nd partial sum
psum_n_m_2 = (int)b2*x0 - a2*y0temp;
```

to the constant substitute form:

```
y0temp  = (int)0x07*x0 + psum_n_m_1;
psum_n_m_1 = psum_n_m_2 - 0x9E*y0temp;
psum_n_m_2 = (int)0xF8*x0 - 0x31*y0temp;
```

Constant substitution for the broadcast form algorithm yielded an improvement of 21.7% (250 cycles) in critical path, and a decrease of 24.2% (32 instructions) in code size on the AT90S8535. Constant substitution for the broadcast form algorithm yielded an improvement of 23.80% (100 cycles) in critical path, and a decrease of 24.8% (32 instructions) in code size on the ATMEGA163.

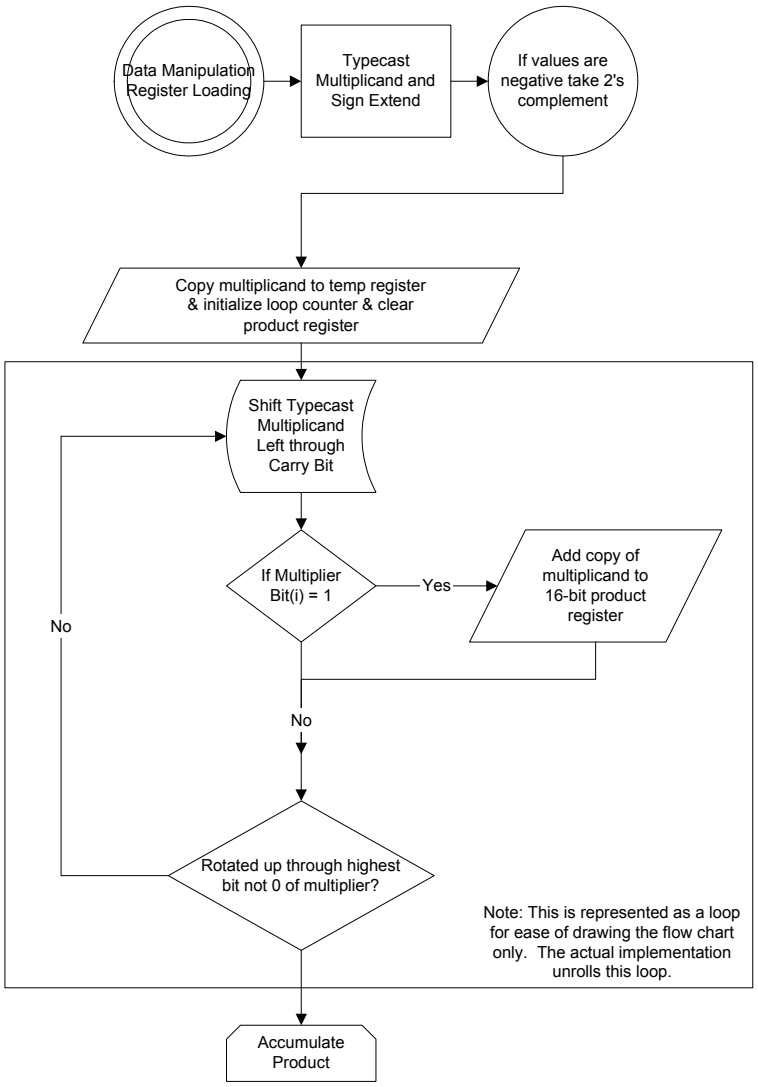
Full C source codes for the constant substitution optimizations are included in the appendix section 7.1.

6.3.2 For Arbitrary Coefficients - Multiply Unrolling

The general purpose IIR filter performance for arbitrary filter coefficients (i.e. the coefficients are variables) can be improved by customization of various parts of the multiply and accumulate functions. The CodevisionAVR compiler generated a subroutine for multiplication for both processors under both types of compiler optimization and for both filter structures. Since both forms of this IIR filter have five multiply and accumulate operations, one optimization is to utilize the Look-Ahead Transformation and "inline" or unroll the iterative multiplication loops for every subsequent multiply and accumulate operation. This would save 3-4 cycles on the RJMP jump to instruction and 5-6 cycles the RET return from sub routine required for each jump to the multiplication subroutine. This results in a savings of 40 - 50 clock cycles each loop of the filter for both structures. This optimization could yield an improvement in critical path of 4.4% to 1088 cycles for the AT90S8535, and an improvement of 12.3% to 358 cycles for the ATMEGA163. The cost of this optimization is five times the code space required for a single multiplication, which is excessive considering the relatively small improvement in critical path.

6.3.3 Optimized Inline Assembly - Multiply/Accumulate Unrolling

The next level of optimization was to improve the critical path for each individual multiplication operation based on the knowledge of the filter coefficients or multipliers in the each multiplication operation. Like the constant substitution, the multiplication operation associated with the b1 filter coefficient (which is zero) is omitted. For the AT90S8535 processor only shift and add assembly instructions are available to perform the multiplication. The multiplication operation was coded by implementing the following flow chart:



It is important to note that for diagramming purposes the shift and add function is represented as a loop. When implemented this loop is unrolled.

One further optimization derived from the number of shifts/adds required for different coefficients. The coefficient corresponding to a1 = 9E (0b10011110) would require a full 16 shifts since it is type cast to an integer and sign extended (2 for each bit since we have a 16 bit value to shift 1 bit) and thirteen 16-bit add (23 8-bit) operations (39 instructions (cycles) total). To reduce this I performed a 2's complement math trick. The operation corresponds to doing -a1*-y1 rather than a1*y1. Four operations were performed to take the 2's complement of the multiplicand (y1) so that the 2's complement of a1 could be hard coded: 2's(0b1111111110011110) = 0b0000000001100001. This new coefficient requires only 14 shifts (2bytes*7 shifts) and three 16-bit additions (6 8-bit adds). Therefore the entire -a1*-y1 operation takes only 20 instructions (cycles) total. For the a1*y1 product, the 2's complement multiplication (-a1*-y1) improves critical path by 19 cycles or 48.7%.

The same type of 2's complement optimization was performed on the b2*x2 product as well. The balance for this optimization is if the number of cycles to take the 2's complement and multiply (shift and add) is less than the number of cycles to just shift and multiply with the original multiplication operands. This is entirely dependent on the bit pattern of the multiplier coefficient. As an example of this technique the b2*x2 multiplication assembly sequence is show below:

```
; b2*x2 multiplication
; b2 is 0xFFF8 - take 2s complement
; use b2 = 0x0008 then take 2s complement of multiplicand to get
; the correct sign on the product
; do 2s complement of x2 and left shift by 3 bits
; *8 is just left shift by 3 bits

        MOV  R20,R8         ; Move multiplicand x2 to R20 R21
        CLR  R21            ; Sign extend multiplicand if positive

        SBRC R20,7   ; Check sign of Multiplicand
        SER  R21            ; Sign extend multiplicand if negative

        ;Take 2s complement of x2 to account for negative of b2
        COM  R20
        COM  R21
        ADD R20,R13         ; R13 has value of 1
        ADC R21,R14; R14 has value of 0 (just add in carry bit from prev add


        ; Left shift word 3 bits to get *8
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        ; the correct result is in R20 R21
; Add the second product to the accumulator
        ADD   R16,R20       ; Add multiplicand low byte to product low byte
        ADC   R17,R21       ; Add multiplicand high byte and product high byte + Carry in
```

The unfolded and optimized assembly code for the general form IIR filter algorithm on the AT90S8535 is included in the appendix section 7.1.

The critical path associated with the multiply and accumulate function on the ATMEGA163 was also optimized. This optimization was done by unrolling the multiply / accumulate operations and hand coding those operations in assembly. This type of operation is significantly easier on the ATMEGA163 since this device has an integer multiplier on chip that is capable of doing signed, unsigned and fractional multiplications (byte*byte = word) in two cycles. Since the hardware multiplier was available and could

do signed multiplication, shift /add operations, and 2' complement operations were not needed. The general approach to this optimization was to load immediate values into registers for the coefficients (b0, b2, a1, a2), do a signed multiply with the multiplicands (x0, x2, y1, y2), and accumulate the results. This optimization utilized an unrolled multiply / accumulate by "inlining" all the multiply / accumulate operations. For comparison the application of this technique on same b2*x2 computation and shown below:

```
; b2*x2 multiplication
; b2 is 0xF8

        MOV  R20,R8         ; Move multiplicand x2 to R20
        LDI  R21,0xF8       ; Load b2 Multiplier into R21
        MULS R21,R20        ; Do signed multiplication  - Result in R0-R1d
; Add the second product to the accumulator
        ADD  R16,R0         ; Add multiplicand low byte to product low byte
        ADC  R17,R1         ; Add multiplicand high byte and product high byte + Carry in
```

The unfolded and optimized assembly code for the general form IIR filter algorithm on the ATMEGA163 is included in the appendix section 7.1.

6.4 General Tradeoffs and Conclusions

The primary tradeoff for this IIR filter implementation is related to the use of fixed-point coefficients. This tradeoff allows for reasonable computation times at the expense of accuracy of the filter. As a result of fixed-point arithmetic the filter center frequency and bandwidth are slightly off of the designed parameters. The positive aspect of the fixed-point mathematics is the ease of generating hand-optimized assembler relative to floating point routines in assembly.

When hand coding inline assembly (assembly mixed with C/C++) it is imperative that the user have a solid understanding of the register utilization of the compiler. This will provide insight in to how to pass variables into and out of the assembly routine as well as provided the user as to which registers are "safe to use". It also provides information on which registers must be saved onto the stack before using them, and restored after use. For example the CodevisionAVR compiler allocates local variables from register R16 to R21, global variables to SRAM, and bit variables from register R2 to R15 (depending on the number of bit variables).

In addition certain assembly instructions will only work for certain register combinations. One such instance where this was an issue, occurred when trying to load an immediate value into a register in the hand-optimized code for the ATMEGA163. On the AVR line of processors the load immediate (LDI) instruction can only load values into the upper half of the register file. Another critical restriction when optimizing the assembly code for the ATMEGA163 was that all of the multiply instructions put the 16-bit result into the R0:R1 pair of registers so general variables could not be allocated to that register pair. The programmer needs to be aware of these types of restrictions when allocating variables to specific registers.

Another constraint is the number and size of variables that are used in the filter algorithm. If the variables used in the algorithm will not all fit into the 32 registers in the register file, then they must be stored in SRAM. If a variable must be stored in SRAM additional load and store instructions must be used to store

register contents, fetch the new data from SRAM, store the results to SRAM, and restore the previous register contents. Each access to SRAM takes two cycles rather than a single cycle. The overhead required to swap out one 8-bit register would be:

2 cycles to back up register in SRAM
2 cycles to copy variable from SRAM to the register
2 cycles to store the modified variable back into SRAM
2 cycles to restore the original register contents from SRAM.

This SRAM Access overhead adds up to 8 cycles for one byte of data and doubles to 16 cycles for word sized data. Fortunately for this IIR filter application, all of the variables were able to be stored in registers.

Since the bulk of the critical path in the IIR algorithm is in the multiply operation, the implementation on the ATMEGA163 is significantly faster than the AT90S8535 regardless of structure with general compiler optimizations. By unfolding the multiplication and generating custom per coefficient multiplication code fragments on the AT90S8535 I was able to compute a general form IIR filter iteration 67.5% faster than the best compiler optimized code running on the ATMEGA163 regardless of optimization type or structure. By generating hand-coded assembly for the AT90S8535 I was able to reduce its critical path to 92 cycles. By generating hand-coded assembly for the ATMEGA163 I was able to reduce its critical path to 30 cycles. Both cases are around an order of magnitude performance increase relative to the general compiler optimizations for the corresponding processor.

The cost of this performance increase is a lack of flexibility if the filter coefficients if they need to be modified or changed. The shift and add optimized multiplication routines for the AT90S8535 would require significant rework to change the filter coefficients and could not be changed during run time without significant branch overhead. The coefficients are much more easily changed on the ATMEGA163 since it only requires loading a register with either an immediate or some previously calculated filter coefficient. The ATMEGA163 optimization is far more suited to runtime adaptations/modifications to filter coefficients than the AT90S8535 optimizations. A table summarizing the code size / critical path for each processor, for both filter structures with all the different optimizations follows:

Configuration Number	Filter Structure	Compiler Optimization		AT90S8535 (Codesize)	AT90S8535 (Critical Path)	ATMega163 (Codesize)	ATMega163 (Critical Path)
1	General	Speed		133	1138	126	408
2		Size		123	1173	126	408
3	Broadcast	Speed		132	1153	129	420
4		Size		115	1202	113	441
5	General	Speed w/Consts.		100	902	97	323
6	Broadcast	Speed w/Consts.		100	903	97	320
7	General	Unrolled Multiply on Speed w. Consts.		400	862	388	283
8	General	Speed, Hand ASM optimized multiply for each filter coefficient		92	92	26	30
				AT90S8535(Hand ASM) V.S. AT90S8535(1-7)		ATMEGA163(Hand ASM) V.S. ATMEGA163(1-7)	
		% improvement of 8 over configuration#	1	30.83	91.92	79.37	92.65
			2	25.20	92.16	79.37	92.65
			3	30.30	92.02	79.84	92.86
			4	20.00	92.35	76.99	93.20
			5	8.00	89.80	73.20	90.71
			6	8.00	89.81	73.20	90.63
			7	77.00	89.33	93.30	89.40
				AT90S8535(Hand ASM) V.S. ATMEGA163(1-7)		ATMEGA163(Hand ASM) V.S. AT90S8535(1-7)	
			1	26.98	77.45	80.45	97.36
			2	26.98	77.45	78.86	97.44
			3	28.68	78.10	80.30	97.40
			4	18.58	79.14	77.39	97.50
			5	5.15	71.52	74.00	96.67
			6	5.15	71.25	74.00	96.68
			7	76.29	67.49	93.50	96.52

In conclusion, it is possible to optimize a particular algorithm to run faster on a part without a hardware multiplier than compiler optimized code will run on a part with a hardware multiplier. It is reassuring to note that as each level of optimization was performed the relative improvement on each processor increased a similar percentage despite the hardware multiplier difference. For example, the percentage improvement between the general structure compiler speed optimization and the general structure compiler speed optimization with constant substitution was 89.8% for the AT90S8535 and 90.71% on the ATMEGA163.

The initial goal was simply to investigate the base algorithm implementation on these two microcontrollers, and to optimize the algorithm on the AT90S8535 to run at least as fast as the compiler optimized algorithm on the ATMEGA163. This goal was met and surpassed by a significant margin. Further optimization efforts improved performance on the ATMEGA163 based algorithm enough that additional computations could potentially be performed on audio band signals and still satisfy the Nyquist criterion. One such application could be a small filter bank implementing an audio band equalizer. Overall the results obtained were better than expected. This is primarily due to the efficiency of the AVR instruction set and the large number of registers available in the register file. The results of this optimization effort were a success!

7. Appendix

7.1 C Source Code

General & Broadcast forms for AT90S8535

```
/*****************************************
This program was produced by the
CodeWizardAVR V1.0.1.7b Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@mail.com

Project : Filter
Version : 1.0
Date    : 4/27/2001
Author  : Erick L. Oberstar M.S.E.E. candidate

Company : University of Wisconsin - Madison
                1513 Univ. Ave. Madison WI 53706
                Copyright 2001
Comments: This application uses an ATmega163 to
                implement a 2nd order digital filter

Chip type           : AT90S8535
Clock frequency     : 8.000000 MHz
Memory model        : Small
Internal SRAM size  : 1024
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

//#include <mega163.h>
#include <90s8535.h>
#include <delay.h>
#include "float2Qpoint.h"
#include "memmap.h"
#include "dac.h"
#define GeneralForm

#define PWM1DCReg OCR1AL

#ifdef GeneralForm
```

```
char x1, x2, y1, y2;     // Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|1/16|1/32|1/64|1/128
#endif

char y0;          // Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|

char     x0;      // x0 is a filter state variable that is used to copy
                  // the 16bit a/d value it is right shifted 8-bits
                  // and type cast to char to allow multlipication in filter algorithm


char b0,b1,b2,a1,a2;              // Filter Coefficients
// Filter Coefficients are of Q6 form
// Q6 can represent number from -2 to 2 - 1/64
// Decimal place is between bits 5 and 6
// | s| x|  x|  x|  x|   x|   x|   x
//       .
// |-2| 1|1/2|1/4|1/8|1/16|1/32|1/64


int y0temp;                  // Temporary 16 bit filter accumulation
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .

#ifndef GeneralForm
// Partial Product Storage 16 bit filter accumulation
int psum_n_m_2,psum_n_m_1;
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .
```

```
#endif


#define ADC_VREF_TYPE 0x00
// Read the ADC conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
ADCSR|=0x40;
while ((ADCSR&0x10)==0);
ADCSR|=0x10;
return ADCW;
}


// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
// Place your code here
// Read Analog Ch 0 & subtract half scale to remove "offset"

//placed in main for debugging
/*      x0 = read_adc(0);
        x0 = x0 - 0x80;         // Subtract off DC offset
        y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;  // Filter Multiply and
accumulate to Q13 Number
        y0 = (char)((y0temp) >> 6);

        // shift states
        y2 = y1; y1 = y0; x2 = x1; x1 = x0;
        PWM1DCReg = y0 + 0x80;   // Add back DC offset and Update PWM register
        PORTB = ~PORTB;         // Toggle output port to allow measurement of ISR / Sampling Rate

*/


}


// Declare your global variables here

void main(void)
{
// Declare your local variables here
// Input/Output Ports initialization
// Port A
```

```
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF;

// Port C
PORTC=0x00;
DDRC=0x00;

// Port D
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Output Compare
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Output Compare
// OC2 output: Disconnected
TCCR2=0x00;
```

```
ASSR=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;

// ADC initialization
// ADC Clock frequency: 4000.000 kHz
ADCSR=0x81;

#ifdef GeneralForm
x0 = x1 = x2 = y0 =  y1 = y2 = 0;      // Initialize Filter State Variables
#endif
// Filter Coefficients

//Real 1kHz Filter
b0 = FloatToQ6(0.11603539655267);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.11603539655267);
a1 = FloatToQ6(-1.51959288821994);
a2 = FloatToQ6(0.76989749559855);



//2KHz filter

/*b0 = FloatToQ6(0.04956754656797);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.04956754656797);
a1 = FloatToQ6(0.95473001076082);
a2 = FloatToQ6(-0.90091007200971);
 */

// Global enable interrupts
```

```
#asm("sei")

// Place your code here
// Add initialization to force critical multiplication path
b0 = 0x07;  //0.11603539655267;
b1 = 0x00;
b2 = 0xF8;  // -0.11603539655267;
a1 = 0x9E;  // -1.51959288821994;
a2 = 0x31;  // 0.76989749559855;

#ifdef GeneralForm
x0 = 0x7F; x1 = 0x7F; x2 = 0x7F;
y0 = 0x7F; y1 = 0x7F; y2 = 0x7F;
y0temp = 0x0000;

#else

 psum_n_m_2 = 0; // Broadcast structure current sum 3
 psum_n_m_1 = 0; // Broadcast structure current sum 2

#endif

while (1)
        {
        // Place your code here
        //x0 = read_adc(0);

        //x0 = x0 - 0x80;         // Subtract off DC offset

#ifdef GeneralForm

        y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;  // Filter Multiply and
accumulate to Q13 Number
        y0 = (char)((y0temp) >> 8);

        // shift states
        y2 = y1; y1 = y0; x2 = x1; x1 = x0;
#else

        y0temp  = (int)b0*x0 + psum_n_m_1;
        psum_n_m_1 = (int)b1*x0 - a1*y0temp + psum_n_m_2;
        psum_n_m_2 = (int)b2*x0 - a2*y0temp;

        y0 = (char)((y0temp) >> 8);    // not really a needed step

#endif
```

```
//        PWM1DCReg = y0 + 0x80;   // Add back DC offset and Update PWM register
          WRSingleDAC(y0 + 0x80, 0);          // Add back DC offset and put on D/A Ch 0


          PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

          };// End of Main While Loop
}         // End of void Main()
```

General and Broadcast form for the ATMEGA163

```
/*********************************************
This program was produced by the
CodeWizardAVR V1.0.1.7b Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@mail.com

Project : Filter
Version : 1.0
Date    : 4/27/2001
Author  : Erick L. Oberstar M.S.E.E. candidate

Company : University of Wisconsin - Madison
__1513 Univ. Ave. Madison WI 53706
__Copywright 2001
Comments: This application uses and ATmega163 to
__implement a 2nd order digital filter

Chip type           : ATmega163L
Clock frequency     : 8.000000 MHz
Memory model        : Small
Internal SRAM size  : 1024
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega163.h>
//#include <90s8535.h>
#include <delay.h>
#include "float2Qpoint.h"
#include "memmap.h"
#include "dac.h"
```

```
//#define GeneralForm


#define PWM1DCReg OCR1AL

#ifdef GeneralForm
char x1, x2, y1, y2;_// Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|1/16|1/32|1/64|1/128
#endif

char y0;_// Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|

char_x0;_// x0 is a filter state variable that is used to copy
__// the 16bit a/d value it is right shifted 8-bits
__// and type cast to char to allow multlipication in filter algorithm


char b0,b1,b2,a1,a2;__// Filter Coefficients
// Filter Coefficients are of Q6 form
// Q6 can represent number from -2 to 2 - 1/64
// Decimal place is between bits 5 and 6
// | s| x|  x|  x|  x|   x|   x|   x
//       .
// |-2| 1|1/2|1/4|1/8|1/16|1/32|1/64


int y0temp;__// Temporary 16 bit filter accumulation
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .

#ifndef GeneralForm
// Partial Product Storage 16 bit filter accumulation
```

```
int psum_n_m_2,psum_n_m_1;__
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .
#endif


#define ADC_VREF_TYPE 0x20
// Read the 8 most semnificative bits
// of the ADC conversion result
unsigned char read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
ADCSR.6=1;
while (ADCSR.4==0);
ADCSR.4=1;
//return ADCW;
return ADCH;
}


// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
// Place your code here
_// Read Analog Ch 0 & subtract half scale to remove "offset"

//placed in main for debugging
/*_x0 = read_adc(0);
x0 = x0 - 0x80;_// Subtract off DC offset
y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;_// Filter Multiply and
accumulate to Q13 Number
y0 = (char)((y0temp) >> 6);

 // shift states
y2 = y1; y1 = y0; x2 = x1; x1 = x0;
PWM1DCReg = y0 + 0x80;_// Add back DC offset and Update PWM register
PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

*/


}
```

```
// Declare your global variables here

void main(void)
{
// Declare your local variables here

// Input/Output Ports initialization
// Port A
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF;_// output for debugging

// Port C
PORTC=0x00;
DDRC=0x00;_//switch input for debugging

// Port D
PORTD=0x00;
DDRD=0x20;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 8000.000 kHz
// Mode: 8 bit Pulse Width Modulation
// OC1A output: Non-Inv.
// OC1B output: Discon.
// PWM output frequency is doubled
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x91;
TCCR1B=0x09;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
```

```
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Output Compare
// OC2 output: Disconnected
// Timer/Counter 2 is cleared on compare match
TCCR2=0x0A;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x53;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
//TIMSK=0x80;
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;
SFIOR=0x00;         // Not in 8535

// ADC initialization
// ADC Clock frequency: 1000.000 kHz
// ADC Voltage Reference: AREF pin
// Only the 8 most semnificative bits of
// the ADC conversion result are used
ADMUX=ADC_VREF_TYPE;
ADCSR=0x83;_// For 1MHz A/D ClockFreq
//ADCSR=0x81;_// For 4Mhz A/D ClockFreq

#ifdef GeneralForm
x0 = x1 = x2 = y0 =  y1 = y2 = 0;_// Initialize Filter State Variables
#endif
// Filter Coefficients

//Real 1kHz Filter
```

```
b0 = FloatToQ6(0.11603539655267);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.11603539655267);
a1 = FloatToQ6(-1.51959288821994);
a2 = FloatToQ6(0.76989749559855);

*/

//2KHz filter

/*b0 = FloatToQ6(0.04956754656797);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.04956754656797);
a1 = FloatToQ6(0.95473001076082);
a2 = FloatToQ6(-0.90091007200971);
 */

// Global enable interrupts
#asm("sei")

// Place your code here
// Add initialization to force critical multiplication path
b0 = 0x07;  //0.11603539655267;
b1 = 0x00;
b2 = 0xF8;  // -0.11603539655267;
a1 = 0x9E;  // -1.51959288821994;
a2 = 0x31;  // 0.76989749559855;

#ifdef GeneralForm
x0 = 0xFF; x1 = 0xFF; x2 = 0xFF;
y0 = 0xFF; y1 = 0xFF; y2 = 0xFF;
y0temp = 0x00FF;

#else

 psum_n_m_2 = 0; // Broadcast structure current sum 3
 psum_n_m_1 = 0; // Broadcast structure current sum 2

#endif
while (1)
_{
      // Place your code here
_x0 = read_adc(0);

_x0 = x0 - 0x80;_// Subtract off DC offset
```

```
#ifdef GeneralForm

y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;_// Filter Multiply and
accumulate to Q13 Number
y0 = (char)((y0temp) >> 8);

// shift states
y2 = y1; y1 = y0; x2 = x1; x1 = x0;

#else

y0temp  = (int)b0*x0 + psum_n_m_1;
psum_n_m_1 = (int)b1*x0 - a1*y0temp + psum_n_m_2;
psum_n_m_2 = (int)b2*x0 - a2*y0temp;

y0 = (char)((y0temp) >> 8);_// not really a needed step

#endif
//PWM1DCReg = y0 + 0x80;_// Add back DC offset and Update PWM register
WRSingleDAC(y0 + 0x80, 0);_// Add back DC offset and put on D/A Ch 0


PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

};// End of Main While Loop
}// End of void Main();
```

Constant Substitution General and Broadcast Form AT90S8535

```
/*****************************************
This program was produced by the
CodeWizardAVR V1.0.1.7b Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@mail.com

Project : Filter
Version : 1.0
Date    : 4/27/2001
Author  : Erick L. Oberstar M.S.E.E. candidate

Company : University of Wisconsin - Madison
__1513 Univ. Ave. Madison WI 53706
```

__Copywright 2001
Comments: This application uses and ATmega163 to
__implement a 2nd order digital filter

Chip type   : ATmega163L
Clock frequency   : 8.000000 MHz
Memory model   : Small
Internal SRAM size : 1024
External SRAM size : 0
Data Stack size   : 256
***/

//#include <mega163.h>
#include <90s8535.h>
#include <delay.h>
#include "float2Qpoint.h"
#include "memmap.h"
#include "dac.h"
#define GeneralForm

#define PWM1DCReg OCR1AL

#ifdef GeneralForm
char x1, x2, y1, y2;_// Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s| x| x| x| x| x| x| x
//   .
// |-1|1/2|1/4|1/8|1/16|1/32|1/64|1/128
#endif

char y0;_// Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s| x| x| x| x| x| x| x
//   .
// |-1|1/2|1/4|1/8|

char_x0;_// x0 is a filter state variable that is used to copy
__// the 16bit a/d value it is right shifted 8-bits
__// and type cast to char to allow multlipication in filter algorithm

//char b0,b1,b2,a1,a2;__// Filter Coefficients

```
// Filter Coefficients are of Q6 form
// Q6 can represent number from -2 to 2 - 1/64
// Decimal place is between bits 5 and 6
// | s| x|  x|  x|  x|   x|   x|   x
//       .
// |-2| 1|1/2|1/4|1/8|1/16|1/32|1/64


int y0temp;__// Temporary 16 bit filter accumulation
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .

#ifndef GeneralForm
// Partial Product Storage 16 bit filter accumulation
int psum_n_m_2,psum_n_m_1;__
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .
#endif


#define ADC_VREF_TYPE 0x00
// Read the ADC conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
ADCSR|=0x40;
while ((ADCSR&0x10)==0);
ADCSR|=0x10;
return ADCW;
}


// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
// Place your code here
// Read Analog Ch 0 & subtract half scale to remove "offset"

//placed in main for debugging
/*_x0 = read_adc(0);
```

```
x0 = x0 - 0x80;_// Subtract off DC offset
y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;_// Filter Multiply and
accumulate to Q13 Number
y0 = (char)((y0temp) >> 6);

// shift states
y2 = y1; y1 = y0; x2 = x1; x1 = x0;
PWM1DCReg = y0 + 0x80;_// Add back DC offset and Update PWM register
PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

*/

}

// Declare your global variables here

void main(void)
{
// Declare your local variables here
// Input/Output Ports initialization
// Port A
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF;

// Port C
PORTC=0x00;
DDRC=0x00;

// Port D
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
```

```
// Clock value: Timer 1 Stopped
// Mode: Output Compare
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Output Compare
// OC2 output: Disconnected
TCCR2=0x00;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;

// ADC initialization
// ADC Clock frequency: 4000.000 kHz
ADCSR=0x81;

#ifdef GeneralForm
x0 = x1 = x2 = y0 =  y1 = y2 = 0;_// Initialize Filter State Variables
#endif
```

```
// Filter Coefficients

//Real 1kHz Filter
b0 = FloatToQ6(0.11603539655267);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.11603539655267);
a1 = FloatToQ6(-1.51959288821994);
a2 = FloatToQ6(0.76989749559855);

//2KHz filter

/*b0 = FloatToQ6(0.04956754656797);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.04956754656797);
a1 = FloatToQ6(0.95473001076082);
a2 = FloatToQ6(-0.90091007200971);
 */

// Global enable interrupts
#asm("sei")

// Place your code here
// Add initialization to force critical multiplication path
//b0 = 0x07;  //0.11603539655267;
//b1 = 0x00;
//b2 = 0xF8;  // -0.11603539655267;
//a1 = 0x9E;  // -1.51959288821994;
//a2 = 0x31;  // 0.76989749559855;

#ifdef GeneralForm
x0 = 0xFF; x1 = 0xFF; x2 = 0xFF;
y0 = 0xFF; y1 = 0xFF; y2 = 0xFF;
y0temp = 0x00FF;

#else

 psum_n_m_2 = 0; // Broadcast structure current sum 3
 psum_n_m_1 = 0; // Broadcast structure current sum 2

#endif

while (1)
{
 // Place your code here
//x0 = read_adc(0);
```

```
x0 = 0x7F; // Force Critical Path
x0 = x0 - 0x80;_// Subtract off DC offset

#ifdef GeneralForm

y0temp = (int)0x9E*y1 + (int)0x31*y2 + (int)0x07*x0 + (int)0xF8*x2;_// Filter Multiply and accumulate
to Q13 Number
y0 = (char)((y0temp) >> 8);

// shift states
y2 = y1; y1 = y0; x2 = x1; x1 = x0;

#else

y0temp  = (int)0x07*x0 + psum_n_m_1;
psum_n_m_1 = psum_n_m_2 - 0x9E*y0temp;
psum_n_m_2 = (int)0xF8*x0 - 0x31*y0temp;

y0 = (char)((y0temp) >> 8);_// not really a needed step

#endif
//_PWM1DCReg = y0 + 0x80;_// Add back DC offset and Update PWM register
WRSingleDAC(y0 + 0x80, 0);_// Add back DC offset and put on D/A Ch 0


PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

};// End of Main While Loop
}// End of void Main
```

Constant Substitution General and Broadcast Form ATMEGA163

```
/*****************************************
This program was produced by the
CodeWizardAVR V1.0.1.7b Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@mail.com

Project : Filter
Version : 1.0
Date    : 4/27/2001
Author  : Erick L. Oberstar M.S.E.E. candidate
```

```
Company : University of Wisconsin - Madison
                  1513 Univ. Ave. Madison WI 53706
                  Copywright 2001
Comments: This application uses and ATmega163 to
                  implement a 2nd order digital filter

Chip type           : ATmega163L
Clock frequency     : 8.000000 MHz
Memory model        : Small
Internal SRAM size  : 1024
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega163.h>
//#include <90s8535.h>
#include <delay.h>
#include "float2Qpoint.h"
#include "memmap.h"
#include "dac.h"
#define GeneralForm


#define PWM1DCReg OCR1AL

#ifdef GeneralForm
char x1, x2, y1, y2;     // Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|1/16|1/32|1/64|1/128
#endif

char y0;          // Filter State Variables
// Filter State Variables are of Q7 form
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|

char     x0;      // x0 is a filter state variable that is used to copy
                  // the 16bit a/d value it is right shifted 8-bits
```

```
                // and type cast to char to allow multlipication in filter algorithm


//char b0,b1,b2,a1,a2;            // Filter Coefficients
// Filter Coefficients are of Q6 form
// Q6 can represent number from -2 to 2 - 1/64
// Decimal place is between bits 5 and 6
// | s| x|  x|  x|  x|   x|   x|   x
//       .
// |-2| 1|1/2|1/4|1/8|1/16|1/32|1/64


int y0temp;                 // Temporary 16 bit filter accumulation
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .

#ifndef GeneralForm
// Partial Product Storage 16 bit filter accumulation
int psum_n_m_2,psum_n_m_1;
// Filter multiply / accumulate result Variable is Q13 form
// Q13 can represent number from -4 to 4 - 1/8192
// Decimal place is between bits 13 and 14
// |s|x|x|x| x|x|x|x| x|x|x|x| x|x|x|x
//       .
#endif


#define ADC_VREF_TYPE 0x20
// Read the 8 most semnificative bits
// of the ADC conversion result
unsigned char read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
ADCSR.6=1;
while (ADCSR.4==0);
ADCSR.4=1;
//return ADCW;
return ADCH;
}


// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
```

```
{
// Place your code here
// Read Analog Ch 0 & subtract half scale to remove "offset"

//placed in main for debugging
/*      x0 = read_adc(0);
        x0 = x0 - 0x80;          // Subtract off DC offset
        y0temp = (int)a1*y1 + (int)a2*y2 + (int)b0*x0 + (int)b1*x1 + (int)b2*x2;  // Filter Multiply and
accumulate to Q13 Number
        y0 = (char)((y0temp) >> 6);

        // shift states
        y2 = y1; y1 = y0; x2 = x1; x1 = x0;
        PWM1DCReg = y0 + 0x80;   // Add back DC offset and Update PWM register
        PORTB = ~PORTB;          // Toggle output port to allow measurement of ISR / Sampling Rate

*/
}

// Declare your global variables here

void main(void)
{
// Declare your local variables here

// Input/Output Ports initialization
// Port A
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF; // output for debugging

// Port C
PORTC=0x00;
DDRC=0x00; //switch input for debugging

// Port D
PORTD=0x00;
DDRD=0x20;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
```

```
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 8000.000 kHz
// Mode: 8 bit Pulse Width Modulation
// OC1A output: Non-Inv.
// OC1B output: Discon.
// PWM output frequency is doubled
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x91;
TCCR1B=0x09;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Output Compare
// OC2 output: Disconnected
// Timer/Counter 2 is cleared on compare match
TCCR2=0x0A;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x53;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
//TIMSK=0x80;
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
```

```
ACSR=0x80;
SFIOR=0x00;        // Not in 8535

// ADC initialization
// ADC Clock frequency: 1000.000 kHz
// ADC Voltage Reference: AREF pin
// Only the 8 most semnificative bits of
// the ADC conversion result are used
ADMUX=ADC_VREF_TYPE;
ADCSR=0x83;          // For 1MHz A/D ClockFreq
//ADCSR=0x81;        // For 4Mhz A/D ClockFreq

#ifdef GeneralForm
x0 = x1 = x2 = y0 =  y1 = y2 = 0;      // Initialize Filter State Variables
#endif
// Filter Coefficients

//Real 1kHz Filter
b0 = FloatToQ6(0.11603539655267);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.11603539655267);
a1 = FloatToQ6(-1.51959288821994);
a2 = FloatToQ6(0.76989749559855);

*/

//2KHz filter

/*b0 = FloatToQ6(0.04956754656797);
b1 = FloatToQ6(0.0);
b2 = FloatToQ6(-0.04956754656797);
a1 = FloatToQ6(0.95473001076082);
a2 = FloatToQ6(-0.90091007200971);
 */

// Global enable interrupts
#asm("sei")

// Place your code here
// Add initialization to force critical multiplication path
//b0 = 0x07;  //0.11603539655267;
//b1 = 0x00;
//b2 = 0xF8;  // -0.11603539655267;
//a1 = 0x9E;  // -1.51959288821994;
//a2 = 0x31;  // 0.76989749559855;
```

```
#ifdef GeneralForm
x0 = 0xFF; x1 = 0xFF; x2 = 0xFF;
y0 = 0xFF; y1 = 0xFF; y2 = 0xFF;
y0temp = 0x00FF;

#else

 psum_n_m_2 = 0; // Broadcast structure current sum 3
 psum_n_m_1 = 0; // Broadcast structure current sum 2

#endif
while (1)
        {
    // Place your code here
        //x0 = read_adc(0);

        x0 = 0x7F; // Force Critical Path
        x0 = x0 - 0x80;          // Subtract off DC offset

#ifdef GeneralForm

        y0temp = (int)0x9E*y1 + (int)0x31*y2 + (int)0x07*x0 + (int)0xF8*x2;     // Filter Multiply and
accumulate to Q13 Number
        y0 = (char)((y0temp) >> 8);

        // shift states
        y2 = y1; y1 = y0; x2 = x1; x1 = x0;

#else

        y0temp  = (int)0x07*x0 + psum_n_m_1;
        psum_n_m_1 = psum_n_m_2 - 0x9E*y0temp;
        psum_n_m_2 = (int)0xF8*x0 - 0x31*y0temp;

        y0 = (char)((y0temp) >> 8);   // not really a needed step

#endif
//      PWM1DCReg = y0 + 0x80;   // Add back DC offset and Update PWM register
        WRSingleDAC(y0 + 0x80, 0);          // Add back DC offset and put on D/A Ch 0


        PORTB = ~PORTB;         // Toggle output port to allow measurement of ISR / Sampling Rate

        };// End of Main While Loop
}       // End of void Main();
```

Customized Multiplication with constant substitution on AT90S8535

```
/*****************************************
This program was produced by the
CodeWizardAVR V1.0.2.1 Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@xnet.ro

Project : Multiply Testing
Version :
Date    : 3/25/2002
Author  : Erick L. Oberstar
Company : University of Wisconsin - Madison
Comments:
Test 8bit*8bit multiply with 16-bit result


Chip type           : AT90S8535
Clock frequency     : 8.000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*****************************************/

#include <90s8535.h>

//#define b0 0x07  //0.11603539655267;
//#define b1 0x00
//#define b2 0xF8  // -0.11603539655267;
//#define a1 0x9E  // -1.51959288821994;
//#define a2 0x31  // 0.76989749559855;

#define b0 0x0007  //0.11603539655267;
#define b1 0x0000
#define b2 0xFFF8  // -0.11603539655267;
#define a1 0xFF9E  // -1.51959288821994;
#define a2 0x0031  // 0.76989749559855;

// Declare your global variables here

// Bit variables allocate Register R2 - R15 (128-bit variables)
// Allocates registers for custom multiply routine
```

```
bit customMultiplyAllocate;  // Block out all 128 bits R2-R15
//Mulitplication Registers
//        Product register = R2:3  ; y0temp = R2:3, y0 = R3 (output(n))
//        Partial product register =  R4:5
//        Multiplicand (sign extended) = R6:7
//Filter State variables:
//        x2 register = R8        ; input(n-2)
//        x1 register = R9        ; input(n-1)
//        x0 register = R10       ; input(n)
//        y2 register = R11       ; output(n-2)
//        y1 register = R12       ; output(n-1)

void main(void)
{
// Declare your local variables here

signed int accumulator;         // Accumulator for sum of products
                                        // Resides @ R16:17
signed int product;             // Product register for multiplication results
                                        // Resides @ R18:19
signed int multiplicand;        // Signed Multiplicand place holder R20:21
                                        // Resides @ R20:21

// Input/Output Ports initialization
// Port A
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF;

// Port C
PORTC=0x00;
DDRC=0x00;

// Port D
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
```

```
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Output Compare
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Output Compare
// OC2 output: Disconnected
TCCR2=0x00;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;
```

```
// initialize custom multiply routine registers to zero
#asm
        CLR R2;
        CLR R3;
        CLR R4;
        CLR R5;
        CLR R6;
        CLR R7;
        CLR R8;
        CLR R9;
        CLR R10;
        CLR R11;
        CLR R12;
        CLR R13;
        CLR R14;
        CLR R15;
        CLR R16;
        CLR R17;
        CLR R18;
        CLR R19;
        CLR R20;
        CLR R21;

        ; preload x0 with a value to multiply with
        LDI R20, 0xFF
        MOV R10,R20          ; Load x0 with value
        ; dont need to load x1 as b1 is zero
        MOV R8,R20 ; Load x2 with value
        MOV R11,R20          ; Load y2 with value
        MOV R12,R20          ; Load y1 with value
        ; Load 1s register - used for 2s complement
        LDI R20, 0x01
        MOV R13,R20
#endasm


while (1)
        {
        //y0temp = (int)0x9E*y1 + (int)0x31*y2 + (int)0x07*x0 + (int)0xF8*x2;    // Filter Multiply and
accumulate to Q13 Number


        #asm

        ; b0*x0 multiplication
        ; b0 is 0x07
```

```
        CLR R18             ; Clear Product Register before multiplication
        CLR R19

        MOV  R20,R10        ; Move multiplicand x0 to R20 R21
        CLR  R21            ; Sign extend multiplicand if positive

        SBRC R20,7   ; Check sign of Multiplicand
        SER  R21            ; Sign extend multiplicand if negative

        ; multiplier bit 0 = 1 do addition
        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

        ; multiplier bit 1 = 1 do addition
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C

        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

        ; multiplier bit 2 = 1 do addition
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C

        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

        ; multiplier bit 3 - bit 15 = 0 dont do anything

; Add the first product to the accumulator
        ADD   R16,R18       ; Add multiplicand low byte to product low byte
        ADC   R17,R19       ; Add multiplicand high byte and product high byte + Carry in

; b2*x2 multiplication
; b2 is 0xFFF8 - take 2s complement
; use b2 = 0x0008 then take 2s complement of multiplicand to get
; the correct sign on the product
; do 2s complement of x2 and left shift by 3 bits
; *8 is just left shift by 3 bits

        MOV  R20,R8         ; Move multiplicand x2 to R20 R21
        CLR  R21            ; Sign extend multiplicand if positive

        SBRC R20,7   ; Check sign of Multiplicand
        SER  R21            ; Sign extend multiplicand if negative
```

```
        ;Take 2s complement of x2 to account for negative of b2
        COM  R20
        COM  R21
        ADD R20,R13         ; R13 has value of 1
        ADC R21,R14; R14 has value of 0 (just add in carry bit from prev add


        ; Left shift word 3 bits to get *8
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C
        ; the correct result is in R20 R21
; Add the second product to the accumulator
        ADD   R16,R20       ; Add multiplicand low byte to product low byte
        ADC   R17,R21       ; Add multiplicand high byte and product high byte + Carry in

; a1*y1 multiplication
; a1 is 0xFF9E - take 2s complement
; use a1 = 0x0062 then take 2s complement of multiplicand to get
; the correct sign on the product
; a1 = 0000 0000 0110 0010 so
; shift left 1 & add, shift left 4 more and add, shift left 1 and add

        CLR R18             ; Clear Product Register before multiplication
        CLR R19

        MOV  R20,R12        ; Move multiplicand y1 to R20 R21
        CLR  R21            ; Sign extend multiplicand if positive

        SBRC R20,7   ; Check sign of Multiplicand
        SER  R21            ; Sign extend multiplicand if negative

        ;Take 2s complement of y1 to account for negative of a1
        COM  R20
        COM  R21
        ADD R20,R13         ; R13 has value of 1
        ADC R21,R14; R14 has value of 0 (just add in carry bit from prev add

        ; multiplier bit 1 = 1 do addition       (bit 1 is fist bit with value = 1)
        LSL    R20          ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL    R21          ; Shift multiplicand high byte over by 1 [ ]<-C

        ADD   R18,R20       ; Add multiplicand low byte to product low byte
```

```
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

        ; multiplier bit 5 = 1
        ; Left shift word 4 bits to get to next add (bit 5 of multiplier is next 1)
        LSL   R20           ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL   R21           ; Shift multiplicand high byte over by 1 [ ]<-C

        LSL   R20           ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL   R21           ; Shift multiplicand high byte over by 1 [ ]<-C

        LSL   R20           ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL   R21           ; Shift multiplicand high byte over by 1 [ ]<-C

        LSL   R20           ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL   R21           ; Shift multiplicand high byte over by 1 [ ]<-C

        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

        ; multiplier bit 6 = 1 do addition
        ; Left shift word 1 bit to get to next add (bit 6 of multiplier is next 1)
        LSL   R20           ; Shift multiplicand low byte over by 1  C <- [ ]
        ROL   R21           ; Shift multiplicand high byte over by 1 [ ]<-C

        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in

; Add the third product to the accumulator
        ADD   R16,R18       ; Add multiplicand low byte to product low byte
        ADC   R17,R19       ; Add multiplicand high byte and product high byte + Carry in

; a2*y2 multiplication
; a2 is 0x31
    CLR R18             ; Clear Product Register before multiplication
    CLR R19

        MOV  R20,R11        ; Move multiplicand y2 to R20 R21
CLR  R21                ; Sign extend multiplicand if positive

        SBRC R20,7   ; Check sign of Multiplicand
        SER  R21            ; Sign extend multiplicand if negative

        ; multiplier bit 0 = 1 do addition
        ADD   R18,R20       ; Add multiplicand low byte to product low byte
        ADC   R19,R21       ; Add multiplicand high byte and product high byte + Carry in
```

```
            ; multiplier bit 1 to 3 = 0 just shift
            LSL    R20             ; Shift multiplicand low byte over by 1  C <- [ ]
            ROL    R21             ; Shift multiplicand high byte over by 1 [ ]<-C

            LSL    R20             ; Shift multiplicand low byte over by 1  C <- [ ]
            ROL    R21             ; Shift multiplicand high byte over by 1 [ ]<-C

            LSL    R20             ; Shift multiplicand low byte over by 1  C <- [ ]
            ROL    R21             ; Shift multiplicand high byte over by 1 [ ]<-C

            ; multiplier bit 4 = 1 do addition
            LSL    R20             ; Shift multiplicand low byte over by 1  C <- [ ]
            ROL    R21             ; Shift multiplicand high byte over by 1 [ ]<-C

            ADD    R18,R20         ; Add multiplicand low byte to product low byte
            ADC    R19,R21         ; Add multiplicand high byte and product high byte + Carry in

            ; multiplier bit 5 = 1 do addition
            LSL    R20             ; Shift multiplicand low byte over by 1  C <- [ ]
            ROL    R21             ; Shift multiplicand high byte over by 1 [ ]<-C

            ADD    R18,R20         ; Add multiplicand low byte to product low byte
            ADC    R19,R21         ; Add multiplicand high byte and product high byte + Carry in

            ; multiplier bit 6 - bit 15 = 0 dont do anything

        ; Add the fourth & final product to the accumulator
            ADD    R16,R18         ; Add multiplicand low byte to product low byte
            ADC    R17,R19         ; Add multiplicand high byte and product high byte + Carry in

        ; Update states
            ;          y2 = y1; y1 = y0; x2 = x1; x1 = x0;
            MOV R11,R12            ; y2 = y1
            MOV R12,R17            ; y1 = accumlator(high) = y0 ; (accumulator = y0temp)
            MOV R8,R9              ; x2 = x1
            MOV R9,R10 ; x1 = x0
        #endasm

        PORTB = ~PORTB;
    };
}
```

Customized Multiplication with constant substitution on ATMEGA163

```
/*****************************************
This program was produced by the
```

CodeWizardAVR V1.0.2.1 Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro , hpinfotech@xnet.ro

Project : Multiply Testing
Version :
Date : 3/25/2002
Author : Erick L. Oberstar
Company : University of Wisconsin - Madison
Comments:
Test 8bit*8bit multiply with 16-bit result

Chip type : ATMEGA163
Clock frequency : 8.000000 MHz
Memory model : Small
Internal SRAM size : 512
External SRAM size : 0
Data Stack size : 128
**********/

```
#include <mega163.h>

//#define b0 0x07  //0.11603539655267;
//#define b1 0x00
//#define b2 0xF8  // -0.11603539655267;
//#define a1 0x9E  // -1.51959288821994;
//#define a2 0x31  // 0.76989749559855;

#define b0 0x0007  //0.11603539655267;
#define b1 0x0000
#define b2 0xFFF8  // -0.11603539655267;
#define a1 0xFF9E  // -1.51959288821994;
#define a2 0x0031  // 0.76989749559855;

// Declare your global variables here

// Bit variables allocate Register R2 - R15 (128-bit variables)
// Allocates registers for custom multiply routine

bit customMultiplyAllocate;  // Block out all 128 bits R2-R15
//Mulitplication Registers
//        Product register = R2:3  ; y0temp = R2:3, y0 = R3 (output(n))
```

```
//      Partial product register =  R4:5
//      Multiplicand (sign extended) = R6:7
//Filter State variables:
//      x2 register = R8        ; input(n-2)
//      x1 register = R9        ; input(n-1)
//      x0 register = R10       ; input(n)
//      y2 register = R11       ; output(n-2)
//      y1 register = R12       ; output(n-1)

void main(void)
{
// Declare your local variables here

signed int accumulator;         // Accumulator for sum of products
                                        // Resides @ R16:17
signed int product;             // Product register for multiplication results
                                        // Resides @ R18:19
signed int multiplicand;        // Signed Multiplicand place holder R20:21
                                        // Resides @ R20:21

// Input/Output Ports initialization
// Port A
PORTA=0x00;
DDRA=0x00;

// Port B
PORTB=0x00;
DDRB=0xFF;

// Port C
PORTC=0x00;
DDRC=0x00;

// Port D
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
```

```
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Output Compare
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Output Compare
// OC2 output: Disconnected
TCCR2=0x00;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;



// initialize custom multiply routine registers to zero
#asm
        CLR R2;
        CLR R3;
```

```
        CLR R4;
        CLR R5;
        CLR R6;
        CLR R7;
        CLR R8;
        CLR R9;
        CLR R10;
        CLR R11;
        CLR R12;
        CLR R13;
        CLR R14;
        CLR R15;
        CLR R16;
        CLR R17;
        CLR R18;
        CLR R19;
        CLR R20;
        CLR R21;

        ; preload x0 with a value to multiply with
        LDI R20, 0xFF
        MOV R10,R20          ; Load x0 with value
        ; dont need to load x1 as b1 is zero
        MOV R8,R20 ; Load x2 with value
        MOV R9,R20 ; Load x1 with value
        MOV R11,R20          ; Load y2 with value
        MOV R12,R20          ; Load y1 with value
        ; Load 1s register - used for 2s complement
        LDI R20, 0x01
        MOV R13,R20
#endasm


while (1)
        {
        //y0temp = (int)0x9E*y1 + (int)0x31*y2 + (int)0x07*x0 + (int)0xF8*x2;    // Filter Multiply and
accumulate to Q13 Number


        #asm
            CLR R16             ; Clear Accumulator before multiplication
            CLR R17

        ; b0*x0 multiplication
        ; b0 is 0x07
```

```
        MOV  R20,R10        ; Move multiplicand x0 to R20
        LDI  R21,0x07       ; Load b0 Multiplier into R21
        MULS R21,R20        ; Do signed multiplication  - Result in R0-R1d
; Add the first product to the accumulator
        ADD  R16,R0         ; Add multiplicand low byte to product low byte
        ADC  R17,R1         ; Add multiplicand high byte and product high byte + Carry in

; b2*x2 multiplication
; b2 is 0xF8

        MOV  R20,R8         ; Move multiplicand x2 to R20
        LDI  R21,0xF8       ; Load b2 Multiplier into R21
        MULS R21,R20        ; Do signed multiplication  - Result in R0-R1d
; Add the second product to the accumulator
        ADD  R16,R0         ; Add multiplicand low byte to product low byte
        ADC  R17,R1         ; Add multiplicand high byte and product high byte + Carry in

; a1*y1 multiplication
; a1 is 0x9E

        MOV  R20,R12        ; Move multiplicand y1 to R20
        LDI  R21,0x9E       ; Load a1 Multiplier into R21
        MULS R21,R20        ; Do signed multiplication  - Result in R0-R1d
; Add the third product to the accumulator
        ADD  R16,R0         ; Add multiplicand low byte to product low byte
        ADC  R17,R1         ; Add multiplicand high byte and product high byte + Carry in

; a2*y2 multiplication
; a2 is 0x31

        MOV  R20,R11        ; Move multiplicand y2 to R20 R21
        LDI  R21,0x31       ; Load a2 Multiplier into R21
        MULS R21,R20        ; Do signed multiplication  - Result in R0-R1d
; Add the fourth product to the accumulator
        ADD  R16,R0         ; Add multiplicand low byte to product low byte
        ADC  R17,R1         ; Add multiplicand high byte and product high byte + Carry in


; Update states
        ;       y2 = y1; y1 = y0; x2 = x1; x1 = x0;
        MOV R11,R12         ; y2 = y1
        MOV R12,R17         ; y1 = accumlator(high) = y0 ; (accumulator = y0temp)
        MOV R8,R9           ; x2 = x1
        MOV R9,R10 ; x1 = x0
#endasm
```

```
        PORTB = ~PORTB;
    };
}
```

Source Code to convert a floating point number to a 8-bit Q6 or Q7 fixed point The code for float2Qpoint.c is given below:

```
/*
Author  : Erick L. Oberstar M.S.E.E. candidate

Company : University of Wisconsin - Madison
                  1513 Univ. Ave. Madison WI 53706
                  Copywright 2001
*/
// Source code for floating point to 8-bit Qpoint conversion

#include <math.h>
#include "float2Qpoint.h"

//convert float to 8bit Q7 Number
// Q7 can represent number from -1 to 1 - 1/128
// Decimal place is between bits 6 and 7
// | s|  x|  x|  x|   x|   x|   x|    x
//    .
// |-1|1/2|1/4|1/8|1/16|1/32|1/64|1/128

unsigned char FloatToQ7(float fpval)
{
        unsigned char charval = 0;
        if (fpval < 0)
        {
                charval = charval||bitseven;
        }
        return;
}

//convert float to 8bit Q6 Number
// Q6 can represent number from -2 to 2 - 1/64
// Decimal place is between bits 5 and 6
// | s| x|  x|  x|  x|   x|   x|   x
//       .
// |-2| 1|1/2|1/4|1/8|1/16|1/32|1/64

unsigned char FloatToQ6(float fpval)
{
```

```
 char charval = 0;

if (fpval<0.0)
{
   charval = charval|bitseven;
   fpval = 2.0 - fabs(fpval);
}
if (fpval>=1.0)
{
   charval = charval|bitsix;
   fpval = fpval - 1.0;
}
if (fpval>=(1.0/2.0))
{
   charval = charval|bitfive;
   fpval = fpval - (1.0/2.0);
}
if (fpval>=(1.0/4.0))
{
   charval = charval|bitfour;
   fpval = fpval - (1.0/4.0);
}
if (fpval>=(1.0/8.0))
{
   charval = charval|bitthree;
   fpval = fpval - (1.0/8.0);
}
if (fpval>=(1.0/16.0))
{
   charval = charval|bittwo;
   fpval = fpval - (1.0/16.0);
}
if (fpval>=(1.0/32.0))
{
   charval = charval|bitone;
   fpval = fpval - (1.0/32.0);
}
if (fpval>=(1.0/64.0))
{
   charval = charval|bitzero;
   fpval = fpval - (1.0/64.0);
}
return charval;
}// end of unsigned char FloatToQ6(float fpval)
```

7.2 Compiled C-Code ASM/listing files

Compiler generated 8-bitx8-bit signed to 16bit signed Result:

The critical path of this algorithm is 213 cycles including the data manipulation (10 cycles), branch, and return overhead of 8-10 cycles depending on the storage location of the code.

The setup – variable initialization & type casting

```
        ;□ □ □      126 b0 = 0x07;  //0.11603539655267;
00006f e007             LDI  R16,LOW(7)      //R16 is the b0 coefficient
        ;□ □ □      127 b1 = 0x00;
000070 e010             LDI  R17,LOW(0)      //R17 is the  b1 coefficient
        ;□ □ □      128 b2 = 0xF8;  // -0.11603539655267;
000071 ef28             LDI  R18,LOW(248)  //R18 is the b2 coefficient
        ;□ □ □      129 a1 = 0x9E;  // -1.51959288821994;
000072 e93e             LDI  R19,LOW(158)  //R19 is the a1 coefficient
        ;□ □ □      130 a2 = 0x31;  // 0.76989749559855;
000073 e341             LDI  R20,LOW(49)    //R20 is the a2 coefficient
        ;□ □ □      131
        ;□ □ □      132
        ;□ □ □      133 while (1)
         _0x2:
        ;□ □ □      134      {
        ;□ □ □      135      // Place your code here
        ;□ □ □      136              x0 = 0x0A;//10;
000074 e05a             LDI  R21,LOW(10)    // R21 is x0
        ;□ □ □      137              y0temp = 0;    //y0temp is software stack offset +8&9
000075 27ee             CLR  R30
000076 87e8             STD  Y+8,R30
000077 87e9             STD  Y+8+1,R30
        ;□ □ □      138
        ;□ □ □      139              y0temp = (int)b0*x0;
000078 2fe0             MOV  R30,R16         // copies b0 to R30
000079 27ff             CLR  R31             // clears  R31(upper half of typecast)
00007a fde7             SBRC R30,7           // If msb of b0 is 1
00007b efff             SER  R31             // Then sign extend to upper byte
00007c 2fae             MOV  R26,R30    // Copy 16-bit b0 to  pair R26-27(R27 is MSB)
00007d 2fbf             MOV  R27,R31
00007e 2fe5             MOV  R30,R21         // copy x0 to R30
00007f 27ff             CLR  R31             // clears  R31(upper half of typecast)
000080 fde7             SBRC R30,7           // If msb of x0 is 1
000081 efff             SER  R31             // Then sign extend to upper byte
```

```
000082 d03b        RCALL __MULW12 //Call the multiply routine
000083 87e8        STD  Y+8,R30       // Store the Results of the multiply
000084 87f9        STD  Y+8+1,R31
```

The signed multiply routine

```
      __MULW12:
0000be d004        RCALL __CHKSIGNW      //Check the sign of 16-bit x0 & b0
                                       // if they are neg 2's complement them
                                         //Store product result sign in T Flag
0000bf dfee        RCALL __MULW12U       //Do unsigned multiply routine
0000c0 f40e        BRTC __MULW121        // If product result should be positive
0000c1 dfe8        RCALL __ANEGW1        // Take 2's complement of result
      __MULW121:
0000c2 9508        RET                   // else just return
```

Checking the signs

```
      __CHKSIGNW:
0000c3 94e8        CLT                 //Clear the T bit in the SREG register
0000c4 fff7        SBRS R31,7          // If x0 val in R30:R31 is:
0000c5 c002        RJMP __CHKSW1       //Positive call CHKSW1
0000c6 dfe3        RCALL __ANEGW1      // negative do 2's complement of it
0000c7 9468        SET                 // set the T flag in SREG register 1 neg num

// If val in R30:R31 is positive
      __CHKSW1:
0000c8 ffb7        SBRS R27,7          // Check if b0 (R26:R27) is negative
0000c9 c006        RJMP __CHKSW2       // If its positive bail out
// otherwise take 2's complement of it
0000ca 95a0        COM  R26            // take 1's complement of LSB
0000cb 95b0        COM  R27            // take 1's complement of MSB
0000cc 9611        ADIW R26,1          // Add 1 to integer val in R26:27

//Toggle the state of the T Flag (something to do with sign bit)
0000cd f800        BLD  R0,0           // Copy T Flag to R0.Bit0
0000ce 9403        INC  R0             // Bump R0 by 1 & store back
0000cf fa00        BST  R0,0           // Causes T Flag to toggle
                                       // T Flag holds sign of product result
      __CHKSW2:
0000d0 9508        RET
```

Dealing with a negative number

```
// If val in R30:R31 is positive take its 2's complement
        __ANEGW1:
0000aa 95e0         COM  R30            // take 1's complement of LSB
0000ab 95f0         COM  R31            // take 1's complement of MSB
0000ac 9631         ADIW R30,1          // Add 1 to integer val in R30:31
0000ad 9508         RET
```

The unsigned multiply routine

```
        __MULW12U:
0000ae 2e0a         MOV  R0,R26//R0:R1 is the Local copy of the multiplicand (a)
0000af 2e1b         MOV  R1,R27
0000b0 e181         LDI  R24,17   // Initial loop counter value (17) R27 is loop counter
0000b1 27aa         CLR  R26      // Clear out accumulator R26:R27
0000b2 1bbb         SUB  R27,R27
0000b3 c005         RJMP __MULW12U1
        __MULW12U3:
0000b4 f410         BRCC __MULW12U2   // If bit is 1 we add copy of multiplicand
0000b5 0da0         ADD  R26,R0
0000b6 1db1         ADC  R27,R1
        __MULW12U2:
0000b7 95b6         LSR  R27      // Shift Accumulator Right (effects Carry bit)
0000b8 95a7         ROR  R26      // This has effect of shifting multiplicand left before
                                  // adding it the next time it is needed
        __MULW12U1:
0000b9 95f7         ROR  R31      // Rotate multiplier right through the Carry Bit
0000ba 95e7         ROR  R30      // Use Carry bit from this to decide to add or not
0000bb 958a         DEC  R24      // Decrement Loop counter
0000bc f7b9         BRNE __MULW12U3 // Loop again if not done yet
0000bd 9508         RET
```

Compiler Generated 8x8 Signed Multiply routine for part with hardware multiplier:

The signed multiply routine

```
        __MULW12:
0000af d004         RCALL __CHKSIGNW
0000b0 dff6         RCALL __MULW12U
0000b1 f40e         BRTC __MULW121
0000b2 dff0         RCALL __ANEGW1
        __MULW121:
0000b3 9508         RET
```

Unsigned multiplication

```
        __MULW12U:
```

```
0000a7 9ffa        MUL  R31,R26      // Multiplication done in 2 parts because of
0000a8 2df0        MOV  R31,R0       // 8 bit type cast to 16 bits
0000a9 9feb        MUL  R30,R27
0000aa 0df0        ADD  R31,R0
0000ab 9fea        MUL  R30,R26
0000ac 2de0        MOV  R30,R0
0000ad 0df1        ADD  R31,R1
0000ae 9508        RET
```

Dealing with a negative number

```
        __ANEGW1:
0000a3 95e0        COM  R30     // If negative take 2's complement
0000a4 95f0        COM  R31
0000a5 9631        ADIW R30,1
0000a6 9508        RET

        __CHKSIGNW:              // Check and store sign of this term
0000b4 94e8        CLT
0000b5 fff7        SBRS R31,7
0000b6 c002        RJMP __CHKSW1
0000b7 dfeb        RCALL __ANEGW1
0000b8 9468        SET
        __CHKSW1:
0000b9 ffb7        SBRS R27,7   // Check sign of other term and 2's complement if Neg
0000ba c006        RJMP __CHKSW2
0000bb 95a0        COM  R26
0000bc 95b0        COM  R27
0000bd 9611        ADIW R26,1
0000be f800        BLD  R0,0
0000bf 9403        INC  R0
0000c0 fa00        BST  R0,0
        __CHKSW2:
0000c1 9508        RET
```

7.3 Matlab Souce Code

```
close all
clear all
hold off

% Digital Filter On Micro-controller Project (ATmega163L)
% Atmel 8-Bit AVR
% Written By Erick L. Oberstar
% University of Wisconsin Madison
% oberstar@cae.wisc.edu
% copywright 2002

% y(n) = -a1*y(n-1) - a2*y(n-2) + b0*x(n) + b1*x(n-1) + b2*x(n-2)
% <y>       <y1>        <y2>      <x>      <x1>       <x2>

% Narrow band digital filter

fd = 1000;          % Desired digital filter center frequency (Hz)
fs = 12048;    % Sampling rate (samples/sec)
T = 1/fs;           % Sampling interval (seconds)
Bd = 500;           % Desired digital filter band width (Hz)


% Prewarping
fa = tan(pi*fd*T)/(pi*T);
famin = (tan(pi*(fd-(Bd/2))*T))/(pi*T);
famax = (tan(pi*(fd+(Bd/2))*T))/(pi*T);
Ba = famax - famin;

G = (Ba/(2*fa))*sqrt(pi^2*(Ba^2 + 16*fa^2));

% Digital Filter Coefficients

p = 2*pi*Ba;
q = pi^2*Ba^2 + 4*pi^2*fa^2;

b0 = 2*G/(T*(4/T^2 + 2*p/T + q));
b1 = 0;
b2 = -b0;

a1 = (2*q - 8/T^2)/(4/T^2 + 2*p/T + q);
a2 = (4/T^2 - 2*p/T + q)/(4/T^2 + 2*p/T + q);

y1 = 0;
```

```
y2 = 0;
x1 = 0;
x2 = 0;

N = 1000;       % Number of Samples
f = 1000;       % Frequency of Sine Wave
n = 0:1:N;
t = n/fs;
x = sin(2*pi*f*n/fs);

for k=1:N+1
   y(k) = -a1*y1 - a2*y2 + b0*x(k) + b1*x1 + b2*x2;
   y2 = y1;
   y1 = y(k);
   x2 = x1;
   x1 = x(k);
end

plot(t,x)
hold on
plot(t,y,'r')


% Now do data broadcast structure
% Initialize partial product/sums
 psum_n_m_2 = 0; % Broadcast structure current sum 3
 psum_n_m_1 = 0; % Broadcast structure current sum 2
 %yb = zeros(1,N+1); % Broadcast structure output

for k=1:N+1
   yb(k)  = b0*x(k) + psum_n_m_1;
   psum_n_m_1 = b1*x(k) - a1*yb(k) + psum_n_m_2;
   psum_n_m_2 = b2*x(k) - a2*yb(k);
end
plot(t,yb,'g')
grid on
ylabel('yb & y');
xlabel('Time (t)');
titlestring(1) = {'Data Broadcast & Standard IIR Structure filter outputs'};
title(titlestring);
h = legend('xin','yb','y',1);

hold off

figure(2)
error = max(yb-y);
```

```
plot(t,yb-y)
grid on
ylabel('yb-y');
xlabel('Time (t)');
titlestring(1) = {'Error between Data Broadcast & Standard IIR Structure, Max Error is:'};
titlestring(2) = {num2str(error)};
title(titlestring);
h = legend('yb-y',2);



% Frequency Response

figure(3)
[H,W,S] = freqz([b0 b1 b2],[1 a1 a2],1024,fs);
 S.xunits = 'hz';
 freqzplot(H,W,S);
```

7.4 References

Hu, Dr. Yu Hen
Algorithm Transformation for Recurrent Computing Algorithms
http://www.cae.wisc.edu/~ece734/notes02/algtran.ps
© 1998-2001 Dr. Yu Hen Hu

Haiduc, Pavel
CodeVisionAVR V1.0.2.1 User Manual, Rev. N
http://www.hpinfotech.ro/cvavrman.pdf
© 1998-2001 HP InfoTech S.R.L.

ATMEL ATmega163/ATmega163L Data Sheets
Rev. 1142C-9/01
http://www.atmel.com/atmel/acrobat/doc1142.pdf
© Atmel Corporation 2001

ATMEL AT90S8535 Data Sheet
Rev. 1041H–11/01
http://www.atmel.com/atmel/acrobat/doc1041.pdf
© Atmel Corporation 2001

ATMEL AVR Instruction Set
Rev 0856C-9/01
http://www.atmel.com/atmel/acrobat/doc0856.pdf
© Atmel Corporation 2001

Milenkovic, Dr. Paul
http://courses.engr.wisc.edu/ecow/get/ece/432/professorm/homework/ece432h3.pdf
© Dr. Paul Milenkovic 2001

AVR® STK500 User Guide
http://www.atmel.com/atmel/acrobat/doc1925.pdf
© Atmel Corporation 2000

AVR201: Using the AVR® Hardware Multiplier – Application Note
Rev. 1631A–02/00
http://www.atmel.com/atmel/acrobat/doc1631.pdf
© Atmel Corporation 2000

Kesab K. Parhi
VLSI Digital Signal Processing Systems – Design and Implementation.
John Wiley & Sons Inc. © 1999

AVR200: Multiply and Divide Routines – Application Note
Rev. 0936B–10/98

http://www.atmel.com/atmel/acrobat/doc0936.pdf
© Atmel Corporation 1998

AVR222: 8-Point Moving Average Filter – Application Note
Rev. 0940A-A–8/97
http://www.atmel.com/atmel/acrobat/doc0940.pdf
© Atmel Corporation 1997

Floating Point to Fixed Point Conversion of C Code
Andrea G. M. Cilio and Henk Corporaal
Delft University of Technology: Computer Architecture and Digital Techniques Dept.
Mekelweg 4, 2628CD Delft, The Netherlands
http://citeseer.nj.nec.com/cache/papers/...../floating-point-to-fixed.pdf
© Unknown