

Overview

The PD1 uses Analog Devices ADSP-2115 programmable microprocessor for DSP operations. The drivers for the PD1 implement a convolution algorithm for performing FIR filtering. However, the DSP can be programmed to perform other DSP operations such as function approximation and IIR filtering in real time.

Implementing a unique algorithm on the ADSP-2115 is not a simple task, and you should be familiar with DSP coding before beginning. TDT does not provide technical support for developing DSP code.

Resources

To implement your own algorithm on the PD1 you need Analog Devices Assembler and Linker. These are available through Analog Devices web site (www.analog.com/support/frames/dsp_frameset.html). The software is found under the Development Tools section. Product documentation for the ADSP-2100 series is also available on the website and includes DSP coding examples. The most useful manuals are the ADSP-2100 User's Manual and the ADSP2100 Cross-Software Manual.

Example: Bi-Quad IIR filter

DSP code: The DSP code for this filter is listed below and can be found in the file *dnc.dsp*. This code implements a Bi-Quad IIR filter with coefficients on the right channel of a stereo DSP.

```
{*****
      GLOBAL SYSTEM CONSTANTS
*****}
{
  i6 => coes handler
  i7 => data handler routine
}
.module/abs=0x0040/boot=0  custcode;

.const  VERCODE = 0x5001;  { This is device code }
                          { 5 is vaidity mark }
                          { 1 is DSP2101 ver 1.0 }
                          { 00 No info }

.const  DBUF  = 0x0000;  { Data I/O Bufffer }
.const  CBUF  = 0x01B0;  { Command buffer }

.const  DSPID = 0x01CF;  { This is device ID/Version }

.const  LOUT  = 0x01C0;  { Output buffer for Left/Mono}
.const  ROUT  = 0x01C4;  { Output buffer for Right }
.const  O3    = 0x01C5;  { Output buffer for Right }
.const  O4    = 0x01C6;  { Output buffer for Right }
.const  RDAT  = 0x01C8;  { Right Data In }

.const  CMD_  = 0x01E0;  { CMD int reg }
.const  DAT_  = 0x01E8;  { DAT int reg Left/Mono}
.const  COES_ = 0x01F0;  { COES int reg }

.const  MASKCON = 0x024;

.const  YEL_LED = 0x3ff6;
{.const  RED_LED = tx0; }

{*****
      LOCAL VARIABLES
*****}
.const nBIQ    = 16;          {15 for min phase, 1 for all pass}
.const nCoefs  = nBIQ*3;     {3 per stage, double precision}
.const nData   = nBIQ*2+1;   {2 per stage, double precision, 1 input}

.var/pm/abs=0x0000 BootLoader[0x40]; {2101 DSP boot loader software}
.var/pm                               CoefsHi[nCoefs];
.var/pm                               CoefsLow[nCoefs];
.var/dm                               DataLow[nCoefs];
.var/dm                               DataHi[nCoefs];
.var/dm                               ntaps;

{.init CoefsHi:  4194304,3062528,-543488,4194304,3081216,931328;}
{.init CoefsLow: 0,5133312,6407680,0,2942208,6332928;}
{.init CoefsHi:}
{.init CoefsLow;}
.init CoefsHi:  <coefhi.dat>;
.init CoefsLow: <coeflo.dat>;
.init ntaps:    nBIQ;
{*****
      INTERRUPT TABLE
*****}
.global vectMAIN,vectCMD,vectTIME;
vectMAIN:      jump MAIN;
```

```

vectCMD:      jump handCMD;
vectTIME:    jump handTIME;

main:
  call initall;
  call MainInit;
  call Biquad2init;
  i7 = ^MainHand;
  imask = MASKCON;

  ax0 = 0;

here:
  jump here;

{*****
  Command int handler
  *****)
handCMD:
  reset flag_out;
  ena sec_reg;
  ar = dm(CMD_);      { this load command code }
  ax1 = dm(CBUF);    { this loads arg1 }
  ay1 = dm(CBUF+1);  { this loads arg2 }
  set flag_out;

  af = pass ar;

  {Is it reboot}
  ay0 = 6;
  af = ar - ay0;
  if ne jump noboot;
  reset flag_out;      { These three lines must }
  ar = 0x0210;        { be included in any code }
  dm(0x3fff) = ar;    { that should reboot }

noboot:

endcmd:
  dis sec_reg;
  rti;

{*****
  Timer int handler
  *****)
handTIME:
  rti;

{*****
  Inert int handler
  *****)
inert:
  rti;

{*****
  Basic init call
  *****)
initall:
  px = 0xff;
  m0 = 0;
  m1 = 1;
  m2 = 2;
  m3 = -1;
  m4 = 0;
  m5 = 1;
  m6 = 2;
  m7 = -1;

  l0 = 0;

```

```

l1 = 0;
l2 = 0;
l3 = 0;
l4 = 0;
l5 = 0;
l6 = 0;
l7 = 0;

i6 = ^inert;
i7 = ^inert;

ar = 0;
{ar = 0x1249; }          { 0001 0010 0100 1001 }
dm(0x3ffe) = ar;

ar = 0x4f0f;
dm(0x3ff6) = ar;

ar = 1;
dm(0x3ff5) = ar;

dis timer;
{
  ax0 = 0xffff;
  dm(0x3ffd) = ax0;
  ax0 = 0;
  dm(0x3ffc) = ax0;
  ax0 = 5;
  dm(0x3ffb) = ax0;
  ena timer;
}

reset flag_out;
ar = VERCODE;
dm(DSPID) = ar;
set flag_out;

rts;

{*****
  Inits all pointers etc.
  *****/}
MainInit:
  i0 = 0x3800;
  ar = 0;
  cntr = 511;
  do zhists until ce;
zhists:  dm(i0,m1) = ar;

  rts;

{*****
  This is main data receive interupt
  *****/}
MainHand:
  reset flag_out;
  ena sec_reg;

  m2 = -2;
  m6 = -2;

  ar = dm(DAT_);          {sample in}
  dm(LOUT) = ar;

  call biquad2;
  dm(ROUT) = my1;        {sample out}

MainHandEnd:
  m2 = 2;

```

```

        m6 = 2;

        set flag_out;
        dis sec_reg;
        rti;

{*****
biquad2 -- Double precision biquad filter routine.
Assumes:
    data input from DAT
    data output to LOUI.
    dm(ntaps) = number of biquads.

Setup:    1          2 ...
    set LOW  = [b0 a2 a1], [b0 a2 a1], ...
    set HI   = [B0 A2 A1], [B0 A2 A1], ...
            set Data = [ y1 y2], [ y1 y2], ...
    zero the buffers.
    set ntaps to number of biquads.
    set i5 = ^biquad2.

*****}
biquad2init:
    i0 = ^DataLow;
        i1 = ^DataHi;
    ar = 0;
    cntr = nData;
    do zdata until ce;
        dm(i0,m1) = ar;
        dm(i1,m1) = ar;
zdata:    nop;
        i0 = ^ntaps;
        dm(i0,m0)=nBIQ;

        rts;

{*****
biquad2 -- Double precision biquad filter routine.
Assumes:
    data input      :ar
    data output     :my0 .
    dm(ntaps) = number of biquads.

Setup:
        1          2 ...
    set CoefsLow  = [b0 a2 a1], [b0 a2 a1], ...
    set CoefsHi   = [B0 A2 A1], [B0 A2 A1], ...
        set DataLow      = [x0 y2 y1], [ y2 y1], ...
        set DataHigh     = [X0 Y2 Y1], [ Y2 Y1], ...
    zero the buffers.
    set ntaps to number of biquads.
    set i5 = ^biquad2.

Recall  m0 = 0,  m1 = 1, m2 = -1, m3 = -1;
        m4 = 0, m5 = 1, m6 = -2, m7 = -1;

*****}
biquad2:
    ena ar_sat;

    i2 = ^ntaps;
    i0 = ^DataLow;
    i1 = ^DataHi;
    i4 = ^CoefsLow;
        i5 = ^CoefsHi;

        sr = ashift ar by -7 (hi);
        { u*2^-k}

        dm(i0,m0) = sr0;
        { Input data low=0 }
        dm(i1,m0) = sr1;
        { Input data hi}

```

```

mx0 = dm(i2,m0);           { Get nBIQ }
cntr = mx0;

do biq2loop until ce;

    mx0 = dm(i0,m1),       { Get x0 >y2 }
    my0 = pm(i5,m5);      { Get B0 >A2 }
    mr = mx0 * my0 (us),
    mx0 = dm(i0,m1),       { Get y2 >y1 }
    my0 = pm(i5,m5);      { Get A2 >A1 }
    mr = mr - mx0 * my0 (us),
    mx1 = dm(i0,m3),       { Get y1 >y2 }
    my0 = pm(i5,m6);      { Get A1 >B0 }
    mr = mr - mx1 * my0 (us),
    mx0 = dm(i1,m1),       { Get X0 >Y2 }
    my0 = pm(i4,m5);      { Get b0 >a2 }
    mr = mr + mx0 * my0 (su),
    mx0 = dm(i1,m1),       { Get Y2 >Y1 }
    my0 = pm(i4,m5);      { Get a2 >a1 }
    mr = mr - mx0 * my0 (su),
    mx0 = dm(i1,m2),       { Get Y1 >X0 }
    my0 = pm(i4,m5);      { Get a1 >b0 (next) }
    mr = mr - mx0 * my0 (su),
    mx0 = dm(i1,m1),       { Get X0 >Y2 }
    my0 = pm(i5,m5);      { Get B0 >A2 }

    dm(i0,m1) = mx1;       { Store y1->y2, y2 >y1(next x0) }
    mr0 = mr1;             { Move for LS word }
    mr1 = mr2;             { Move for LS word }

    mr = mr + mx0 * my0 (ss),
    mx0 = dm(i1,m1),       { Get Y2 >Y1 }
    my0 = pm(i5,m5);      { Get A2 >A1 }

    mr = mr - mx0 * my0 (ss),
    mx0 = dm(i1,m3),       { Get Y1 >Y2 }
    my0 = pm(i5,m5);      { Get A1 >B0 (next) }
    mr = mr - mx0 * my0 (ss),
    dm(i1,m1) = mx0;       { Store Y1->Y2, Y2 >Y1 (next X0) }

    sr = ashift mr1 by 1 (hi); { [Y0][ 0]*2 }
    sr = sr or lshift mr0 by 1(lo); { ([ 0][y0][Y0][ 0])*2 }

    dm(i0,m0) = sr0;       { y0'->y1 }
    dm(i1,m0) = sr1;       { Y0'->Y1 }

biq2loop:

    sr = ashift mr1 by 8 (hi); { [Y0][ 0]*2^k }
    sr = sr or lshift mr0 by 8(lo); { ([ 0][y0][Y0][ 0])*2^k }
    my1 = sr1;

    rts;

.endmod;

```

Assembling and Linking:

Use the Analog Devices assembler *asm21* to assemble the code and the linker *ld21* to link the code. The PROM splitter *spl21* and hex to binary converter *hexobj* are run to get the code ready for loading to the PD1 DSPs. See the Analog Devices Cross-Software Manual for information on switches for the assembler, linker, and splitter.

Below is an example batch file for how this code was linked:

```
ld21 %1 -a std2101 -e %1 -p -x  
spl21 %1 %1 -bm -i -bs 1024 -bb 1024  
hexobj %1.bnm %1.bin I
```

To change the coefficients you do not have to reassemble the code. Just link and split with the new coefficient files.

Loading the filter to the DSP

Below we develop the C code to load and run the PD1 with our new DSP code. The coefficients are loaded to the right channel of a stereo DSP. A signal is sampled on ADC channel 0. The unfiltered signal is played out of DAC channel 0 and the filtered signal is played out of DAC channel 1. The raw source file is *dncsim.cpp* and the compiled executable is *dncsim.exe*.

The only difference between this code and standard code you would use to implement an FIR filter is a call to load the DSP with the IIR filter. The loading of the DSP code is accomplished after PD1clear() by calling:

```
PD1bootDSP(1, 0x1, "dnc.bin");
```

This loads the dnc.bin file produced by the linker to PD1 DSP 0. Different DSP code could be loaded to other DSPs with PD1bootDSP().

```
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include <string.h>

extern "C"
{
    #include "xbdrv.h"
    #include "apos.h"
    #include "pd1_sup.h"
}

void main()
{
    int i;
    char c;

    if(!apinit(APb))
    {
        printf("\n\nAP Init Error\n\n");
        exit(0);
    }

    do
    {
        if(!XBlinit(APb))
        {
            printf("\n\nXBUS Init Error\n\n");
            exit(0);
        }
        PD1clear(1);
        PD1fixbug(1);
        printf("\n\n Loading...");
        PD1bootDSP(1, 0x1, "dnc.bin");

        PD1clrsched(1);
        PD1laddsimp(1, DSPoutL[0], DAC[0]);
        PD1laddsimp(1, DSPoutR[0], DAC[1]);
        PD1laddsimp(1, ADC[0], DSPinR[0]);
        PD1laddsimp(1, ADC[0], DSPinL[0]);

        PD1npts(1, -1);
        PD1nstrms(1, 0, 0);
    }
}
```

```

        PD1speriod(1,100);
        PD1arm(1);
        PD1go(1);

        printf("\n\n Running...      (Press any key to quit [r] to cycle");
        c=getch();
        PD1stop(1);
    }while(c=='r');
}

```

Sending Coefficients in a Data Packet

You can avoid having to relink your code every time you want to load new coefficients by programming the DSP to use filter coefficients that are sent as PD1 data packets. To see how this is done, we will look at the DSP code for the FIR filtering for the PD1. This file is *fir.dsp*.

See Chapter 3 of the PD1 manual for instructions on how to send filter coefficients as a data packet. We will focus here on how to deal with the filter coefficients on the DSP.

When filter coefficients are sent as a packet they are loaded into the DSP program memory.

```

{
The following VARs are used globally --

    m0 = 0;
    m1 = 1;
    m3 = -1;

    m4 = 0;
    m5 = 1;
    m6 = 2;

    i0 => X1 (delay line Left)
    i0 = num taps

    i4 => B* (coes)      >> saved by int handler
    i4 = num taps + 1

    i5 => load-to coes
    i5 = num taps + 1

    i6 => coes handler
    i7 => data handler routine

    i3 => used to count taps on loading

secondary !!!
    ax0 => next coe set
    ay0 = ntaps - 1

primary !!!
    ax0, af => led blink rate
        0 = NOP
        1 = Quik Move Coes
        2 = Interp Coes

    sr0 => coes act flag
    sr1 => data act flag

    ay0 = newcoes flag (for interp)
}

```

```

.module/abs=0x0040/boot=3 fir;

.var/pm/abs=0x0200 B1[255];
.var/pm/abs=0x0300 B2[255];
.var/dm/abs=0x3800/CIRC X1[127];
.var/dm/abs=0x3880/CIRC X2[127];
.var/dm/abs=0x3900/CIRC BB[255];

.const dred = 0x3ff0;
.const dyel = 0x3ff1;

.include <COM_H.dsp>;

.const VERCODE = 0x5001; { This is device code }
                                { 5 is vaidity mark }
                                { 1 is DSP2101 ver 1.0 }
                                { 00 No info }

.const DBUF = 0x0000; { Data I/O Buffer }
.const CBUF = 0x01B0; { Command buffer }

.const DSPID = 0x01CF; { This is device ID/Version }

.const LOUT = 0x01C0; { Output buffer for Left/Mono}
.const ROUT = 0x01C4; { Output buffer for Right }
.const RDAT = 0x01C8; { Right Data In }

.const CMD_ = 0x01E0; { CMD int reg }
.const DAT_ = 0x01E8; { DAT int reg Left/Mono}
.const COES_ = 0x01F0; { COES int reg }

.const MASKCON = 0x27;

.global vectMAIN,vectCMD,vectTIME;
vectMAIN: jump main;
vectCMD: jump cmdhand; { handCMD }
vectTIME: jump timehand; { handTIME }

main:
    call initall;

    ay0 = 0;

foreg:
    ar = pass ay0;
    if eq jump foreg;

runinterp:
    ay0 = 0;

    { Move in new coes }
    i4 = ^B1;
    i2 = ^BB;
    ar = 14;
    cntr = ar;
    do mvum until ce;
    ar = pm(i4,m5);
mvum: dm(i2,m1) = ar;

    ay1 = m2;
    my1 = ay1;
    ar = 32767;
    ar = ar - ay1;
    ay1 = 1;
    ar = ar + ay1;
    my0 = ar;

bigloop:
    i4 = ^B2;
    i2 = ^BB;
    ar = 14;

```

```

    cntr = ar;
    do ium until ce;
        mx0 = pm(i4,m4);
        mr = mx0 * my0 (ss);
            mx0 = dm(i2,m1);
        mr = mr + mx0 * my1 (rnd);
        pm(i4,m5) = mr1;
ium:  nop;

    ar = m2;
    ar = pass ar;
    if eq jump foreg;

    ar = pass ay0;
    if eq jump bigloop;

        jump runinterp;

{*****
  Call this when LSYNC or GSYNC comes.
  will check if interp is on and
  handle it from there.
  *****/}
swapcoes:
    ar = m2;
    ar = pass ar;
    if eq jump noint;
        dis sec_reg;
        ay0 = 1;
        ena sec_reg;
        rts;
noint:
        i4 = ax0;
        ax0 = i5;
        i5 = i4;
        rts;

{*****
  Timer handler (just does lights)
  *****/}
.global timehand;
timehand:
    imask = MASKCON;
    dm(0x3ff6) = sr0;
    tx0 = srl;
    af = af - 1;
    if ne jump noflsh;
    af = pass ax0;
    sr0 = dm(dred);
    srl = dm(dyel);
    rti;
noflsh:
    sr0 = 0x498f;
    srl = 0x0;
    rti;

{*****
  Basic init call
  *****/}
initall:
    px = 0xff;
        m0 = 0;
        m1 = 1;
        m3 = -1;
        m4 = 0;
    m5 = 1;
    m6 = 2;

```

```

13 = 0;

        i6 = ^inert;
        i6 = ^testcall;

        ar = 0;
{   ar = 0x1249; }   { 0001 0010 0100 1001 }
        dm(0x3ffe) = ar;

ar = 0x4f0f;
dm(0x3ff6) = ar;

ar = 0x2000;
dm(0x3ff5) = ar;

dis timer;
ax0 = 0xffff;
dm(0x3ffd) = ax0;
ax0 = 0;
dm(0x3ffc) = ax0;
ax0 = 5;
dm(0x3ffb) = ax0;
ena timer;

        imask = MASKCON;

reset flag_out;
ar = VERCODE;
dm(DSPID) = ar;
set flag_out;

        ayl = 0xff;

        cntr = 0x2000;
        do flshum until ce;

        sr0 = 0x4f0f;
        srl = 0xffff;
        cntr = 0x100;
        do www until ce;
www:      nop;
flshum:  nop;

ax0 = 100;
ar = 0x498f;
dm(dred) = ar;
ar = 0;
dm(dyel) = ar;
af = pass ax0;
rts;

{*****
Command handler
*****}
.global cmdhand;
cmdhand:

        reset flag_out;
        ena sec_reg;
        ar = dm(CMD_);           { this load command code }
        ax1 = dm(CBUF);         { this loads arg1 }
        ayl = dm(CBUF+1);       { this loads arg2 }
        set flag_out;

        af = pass ar;

        {Is it reset}
        af = af - 1;
        if ne jump notinit;
        ar = 0x4f0f;
        dm(dred) = ar;
        ar = 0xffff;

```

```

        dm(dyel) = ar;
        call Setup;
        jump endcmd;
notinit:

        {Is it testmode}
        af = af - 1;
        if ne jump nottest;
        i6 = ^inert;
        i7 = ^testcall;
        ar = 0x498f;
        dm(dred) = ar;
        ar = 0x0;
        dm(dyel) = ar;
        jump endcmd;
nottest:

        {Is it idlemode}
        af = af - 1;
        if ne jump notidle;
        i6 = ^inert;
        i7 = ^inert;
        ar = 0xffff;
        dm(dyel) = ar;
        jump endcmd;
notidle:

        {Is it lock coes}
        af = af - 1;
        if ne jump notlock;
        i6 = ^inert;
        ar = 0x0;
        dm(dyel) = ar;
        jump endcmd;
notlock:

        {Is it interp on}
        af = af - 1;
        if ne jump notinterp;
        m2 = ax1;
        jump endcmd;
notinterp:

        {Is it reboot on}
        af = af - 1;
        if ne jump notreboot;
        reset flag_out;      { These three lines must }
        ar = 0x0210;         { be included in any code }
        dm(0x3fff) = ar;     { that should reboot      }
notreboot:

endcmd:

        dis sec_reg;
        rti;

{*****}
Init Stuff
initializes all arrays for FIR filtering.
Sets up loop counters etc. Assumes
AX1 hold ntaps and AY1 holds mode.
*****}
Setup:
        i7 = ^inert;
            i6 = ^start;

            i0 = ^X1;
                i4 = ^B1;

        14 = 0;
        10 = 0;
            ar = 0;
        cntr = 512;

```

```

        do clrddel until ce;
            pm(i4,m5) = ar;
clrddel: dm(i0,m1) = ar;

        ar = 128;
        call m_setNTAPS;

        i0 = ^X1;
        i1 = ^X2;
        ax0 = ^B2;

        i4 = ^B2;
        i5 = ^B1;

        reset flag_out;
        ar = 0;
        dm(COES_) = ar;
        dm(DAT_) = ar;
        dm(RDAT) = ar;
        dm(LOUT) = ar;
        dm(ROUT) = ar;
        set flag_out;

        mr1 = 0;
        mr0 = 0;
        m2 = mr0;

        dis sec_reg;
        ay0 = 0;
        ena sec_reg;

        rts;

{*****
start... This is base coes handler
Looks for stuff in the following order...
MARK -> FORMID -> ...
Loads coes to I5 and detects
when it rolls back to AX0.
then flips I4 to I5 etc
*****}
start:    { Look for MARK }
            reset flag_out;
            ena sec_reg;
            ar = dm(COES_);
            set flag_out;

            ayl = xSTART;
            af = ar - ayl;
            if ne jump notmark;
            i6 = ^getformat;
            jump ddd;
notmark:
            ayl = LSYNC;
            af = ar - ayl;
            if ne jump notlsync;
            call swapcoes;
            jump ddd;
notlsync:
            ayl = GSYNC;
            af = ar - ayl;
            if ne jump notgsync;
            call swapcoes;
            jump ddd;
notgsync:

ddd:
            dis sec_reg;
            rti;

```

```

{===== Get FORMAT ID =====}
getformat:
    reset flag_out;
        ena sec_reg;
    ar = dm(COES_);
    set flag_out;

    i6 = ^m_getntaps;

    ayl = MONO;
    af = ar - ayl; { Check for MONO }
    if ne jump notmono;
        i6 = ^m_getntaps;
        i7 = ^m_FIR;
        jump g fend;

notmono:

    ayl = STEREO;
    af = ar - ayl; { Check for STEREO }
    if ne jump notstereo;
        i6 = ^s_getntaps;
        i7 = ^s_FIR;
        jump g fend;

notstereo:

    ayl = MONSTER;
    af = ar - ayl; { Check for MONO-STEREO }
    if ne jump notmonster;
        i6 = ^s_getntaps;
        i7 = ^ms_FIR;
        jump g fend;

notmonster:

g fend:
    dis sec_reg;
    rti;

{===== Get MONO ntaps =====}
m_getntaps:
    reset flag_out;
        ena sec_reg;
    ar = dm(COES_);
    set flag_out;

    call m_setNTAPS;
    i6 = ^m_getcoes;
    dis sec_reg;
    rti;

{===== Get STEREO ntaps =====}
s_getntaps:
    reset flag_out;
        ena sec_reg;
    ar = dm(COES_);
    set flag_out;

    call s_setNTAPS;
    i6 = ^s_getcoes;
    dis sec_reg;
    rti;

{===== Get MONO coes =====}
m_getcoes:
    reset flag_out;
        ena sec_reg;
    ar = dm(COES_);
    set flag_out;
    pm(i5,m5) = ar; { Get data and store to TAPs buffer }

```

```

        modify(i3,m3);
        ar = i3;
        af = pass ar;
        if ne jump m_notall;
        i6 = ^start;
m_notall:
        dis sec_reg;
        srl = 0xffff;
        rti;

{===== Get STEREO coes =====}
s_getcoes:
        reset flag_out;
        ena sec_reg;
        ar = dm(COES_);
        set flag_out;
        pm(i5,m5) = ar;  { Get data and store to TAPs buffer }

        modify(i3,m3);
        ar = i3;
        af = pass ar;
        if ne jump s_notall;
        i6 = ^start;
s_notall:
        dis sec_reg;
        srl = 0xffff;
        rti;

{*****
 This routine writes a zero to both outs
*****}
inert:
{      sr0 = 0x4f0f; }
      rti;

{*****
 This is basic PASS routine
*****}
testcall:
        reset flag_out;
        ena sec_reg;
        ar = dm(DAT_);
        dm(LOUT) = ar;
        ar = dm(RDAT);
        dm(ROUT) = ar;

        set flag_out;
        dis sec_reg;
        sr0 = 0x4f0f;
        rti;

{*****
 This routine runs FIR
 Uses Coes pointed to by ax0
*****}
m_FIR:
        reset flag_out;
        ena sec_reg;
        mx0 = dm(DAT_);
        dm(LOUT) = mx0;
        set flag_out;

        ar = i4;

        i4 = ax0;
        mr = 0, my1 = pm(i4,m5);
        mr = mr - mx0 * my1 (ss);

        dm(i0,m0) = mr;

        mr = 0, mx0 = dm(i0,m1), my0 = pm(i4,m5);

```

```

        cntr = ay0;
        do sop until ce;
sop:      mr = mr + mx0 * my0 (ss), mx0 = dm(i0,m1), my0 = pm(i4,m5);
        mr = mr + mx0 * my0 (rnd), mx0 = dm(i0,m3);
        if mv sat mr;

        i4 = ar;
        dis sec_reg;
        sr0 = 0x4f0f;
        rti;

{*****
  This routine runs FIR (MONO-STEREO)
  Uses Coes pointed to by ax0
  and alternater LEFT -> RIGHT -> LEFT -> RIGHT
  branches to punch point after load
  *****)}
ms_FIR:
        reset flag_out;
        ena sec_reg;
        mx0 = dm(DAT_);
        mx1 = mx0;
        dm(LOUT) = mr0;
        dm(ROUT) = mr1;
        set flag_out;
        jump punch;

{*****
  This routine runs FIR (STERO)
  Uses Coes pointed to by ax0
  and alternater LEFT -> RIGHT -> LEFT -> RIGHT
  *****)}
s_FIR:
        reset flag_out;
        ena sec_reg;
        mx0 = dm(DAT_);
        mx1 = dm(RDAT);
        dm(LOUT) = mr0;
        dm(ROUT) = mr1;
        set flag_out;
punch:
        ar = i4;

        { This does left }
        i4 = ax0;
        mr = 0, my1 = pm(i4,m6);
        mr = mr - mx0 * my1 (ss);
        dm(i0,m0) = mr1;
        mr = 0, mx0 = dm(i0,m1), my0 = pm(i4,m6);
        cntr = ay0;
        do left until ce;
left:    mr = mr + mx0 * my0 (ss), mx0 = dm(i0,m1), my0 = pm(i4,m6);
        mr = mr + mx0 * my0 (rnd), mx0 = dm(i0,m3);
        if mv sat mr;
        sr0 = mr1;

        { This does right }
        i4 = ax0;
        modify(i4,m5);
        mr = 0, my1 = pm(i4,m6);
        mr = mr - mx1 * my1 (ss);
        dm(i1,m0) = mr1;
        mr = 0, mx0 = dm(i1,m1), my0 = pm(i4,m6);
        cntr = ay0;
        do right until ce;
right:   mr = mr + mx0 * my0 (ss), mx0 = dm(i1,m1), my0 = pm(i4,m6);
        mr = mr + mx0 * my0 (rnd), mx0 = dm(i1,m3);
        if mv sat mr;

        mr0 = sr0;

```

```

i4 = ar;

        dis sec_reg;
sr0 = 0x4f0f;
        rti;

{*****
This routine sets up for NTAPS (in AR)
Make sure sec_regs enabled.
Also this is mono version.
*****}
m_setNTAPS:
        10 = ar;      { delay line is exactly ntaps }
        11 = ar;
        ay0 = ar;
        ar = ay0 - 1;
        ay0 = ar;      { AY0 will hold ntaps - 1 }

        ar = 10;
        af = pass ar;
        ar = af + 1;
        14 = ar;      { COE buffers are ntaps + 1 (for SF) }
        15 = ar;      { COE buffers are ntaps + 1 (for SF) }
        i3 = ar;

        rts;

{*****
This routine sets up for NTAPS (in AR)
Make sure sec_regs enabled.
Also this is stereo version.
*****}
s_setNTAPS:
        10 = ar;      { delay line is exactly ntaps }
        11 = ar;
        ay0 = ar;
        ar = ay0 - 1;
        ay0 = ar;      { AY0 will hold ntaps - 1 }

        ar = 10;
        af = pass ar;
        af = af + 1;
        ar = pass af;
        ar = ar + af;

        14 = ar;      { COE buffers are 2*ntaps + 2 (for SFs) }
        15 = ar;      { COE buffers are 2*ntaps + 2 (for SFs) }
        i3 = ar;

        rts;

        nop;
        nop;

.endmod;

```

Illegal Values

Remember that the PD1 reserves integers from +32761 to +32767 for packet encoding.

