Implementing Continuously Programmable Digital Filters with the TMS320C30/40 DSP

APPLICATION REPORT: SPRA190A

Authors: Aaron Robinson - MS Team Leader
Richard Hardie
Harry Heinisch
Advisor: Dr. Fred O. Simons, Jr.; PE: Associate Director of the HCS Lab, Director of FEEDS, and EE Professor
Department of Electrical Engineering
Florida A&M University and Florida State University

Digital Signal Processing Solutions
June 1997

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

Contents

Figures

Table

Implementing Continuously Programmable Digital Filters with the TMS320C30/40 DSP

Abstract

Systems engineers must apply real-time digital hardware solutions to signal processing problems to achieve the goals of reliability, repeatability, and flexibility. DSP offers the only such solution for many applications.

Digital filter components are an integral part of many DSP systems. In particular, continuously programmable digital filters (CPDFs) offer a broad range of high-tech applications, such as optimal filter implementations, Kalman filter design, adaptive system operation, and even the simulation or implementation of linear, time-varying, and nonlinear systems.

This application report describes the implementation of a general purpose CPDF on a Texas Instruments (TI™) TMS320C30/40 development board(s) using optimized coefficient updating algorithms. The performance of a dual-processor design is evaluated for coefficient updating and processing rates as a function of CPDF complexity. As a result, analysts can determine the design limitations for any application. The CPDF implementations presented in this design include IIR filters that are guaranteed to be stable, a major advancement in CPDF design.

1. Introduction

All digital filter implementations are based on algorithms designed to implement difference equations in various cascade, parallel, ladder, or other structures. The equations take the following form:

$$\sum_{n=0}^{N} d_n y(k-n) = \sum_{m=0}^{M} c_m x(k-m)$$

The difference equation algorithms effectively generate the output *y(k)* for k = 0,1,2,3... by iteratively evaluating as follows:

$$y(k-n) = \frac{1}{d_0}\left\{\sum_{m=0}^{M} c_m x(k-m) - \sum_{n=1}^{N} d_n y(k-n)\right\}$$

In most cases the filters are time-invariant, which implies that the d_n,c_m coefficients are constants. Digital filters can be implemented by incorporating algorithms to vary the coefficients with time (to implement time-varying systems). Nonlinear DSP component models require digital filters to be implemented with coefficients that are functions of *x(k) and y(k)*, the inputs and outputs, respectively. In either time-varying or nonlinear implementations, coefficients must be updated continuously.

CPDFs might also be called *continuously programmable digital dynamic hardware,* a term that better illustrates the wide-range of benefits to be derived from these nonlinear and time-varying devices. For example, problem classifications concerned with these devices include adaptive filters, digital system compensators or controllers, noise filters, etc.[1] Specific applications include:

- Kalman filter controllers for optimizing control systems
- Matched filters to pre-condition signals from sensors, to minimize measurement noise, or to extract transmitted signals of a priori characteristics from noisy environments
- Adaptive system components to alleviate excessive changes in system environmental effects due to wide-ranging changes in pressure, temperature, etc
- Programmable window filters to minimize spectrum distortion due to edge effects of acquired multiple data streams
- Network driven learning filters that improve with use based on a formulated criterion

CPDFs can be incorporated into a wide range of high-tech applications that will continue to expand as algorithm and implementation problems are resolved.

2. Optimum Algorithms for Implementing CPDFs

Generating an *H(z)* from an *H(s)* with the bilinear transform relationship can be shown to be equivalent to deriving a Simpson's Rule approximation difference equation model from a differential equation model, which infers a prototype *H(s)* model.

Thus, although *H(s)* is defined only for linear time-invariant models, time-varying and even nonlinear models can be approximated by iteratively applying the bilinear transform relationship to update the a_n, b_m coefficients to their new values since the transformation is equivalent to a valid time-domain operation. Thus, we seek the algorithms for generating the d_n *and* c_m coefficients, respectively, for the denominator *D(z)* and the numerator *N(z)* of *H(z),* which is defined as:

$$H(z) \underline{\Delta} \left. \frac{\sum_{m=0}^{M} b_m s^m}{\sum_{n=0}^{N} a_n s^n} \right|_{s = \frac{z-1}{z+1}}$$

and can be expressed as:

$$H(z) = \frac{\sum_0^m b_m (z-1)^m (z+1)^{N-m}}{\sum_0^N a_n (z-1)^n (z+1)^{N-n}} \underline{\Delta} \frac{N(z)}{D(z)}$$

where:

2/T is assumed to be 1 with no loss of generality because the scale factor *2/T* in the bilinear transformation can be handled by $b_m \leftarrow (2/T)^m b_m$ and $a_n \leftarrow (2/T)^n a_n$; that is, replace the a_n, b_m coefficients with the product of a_n, b_m multiplied by the corresponding powers of *(2/T)*.

To verify the Simons-Harden[2] algorithms for evaluating or obtaining *N(z) and D(z)*, consider the following definitions for *D(z).* If *P(s)* is the denominator of *H(s)*, then:

$$D(z) = (z+1)^N P\left(\frac{z-1}{z+1}\right) \tag{1}$$

Let

$$E(z) = \underline{\underline{\Delta}} P(z-1) \tag{2}$$

$$F(z) \underline{\underline{\Delta}} z^N E\left(\frac{1}{z}\right) \tag{3}$$

$$G(z) \underline{\underline{\Delta}} F\left(z+\frac{1}{2}\right) \tag{4}$$

$$J(z) \underline{\underline{\Delta}} z^N G\left(\frac{1}{z}\right) \tag{5}$$

$$D(z) \underline{\underline{\Delta}} J(2z) \tag{6}$$

By successively applying the transformations defined by (2) through (6), the algorithm is verified by showing:

$$D(z) = (z+1)^N P\left(\frac{z-1}{z+1}\right)$$

Thus, starting with:

$$D(z) = J(2z)$$

$$D(z) = (2z)^N G\frac{1}{2z}$$

$$D(z) = (2z)^N F\left(\frac{1}{2z}+\frac{1}{2}\right)$$

$$D(z) = (2z)^N \left(\frac{z+1}{2z}\right)^N E\left(\frac{2z}{z+1}\right)$$

$$D(z) = (z+1)^N P\left(\frac{2z}{z+1}-1\right)$$

$$D(z) = (z+1)^N P\left(\frac{z-1}{z+1}\right)$$

It is concluded that the defined transformations yield the desired result.

To summarize the transformations required to implement the algorithm, first observe from (1) that the coefficients of the original *H(s)* denominator polynomial are the coefficients of polynomial *P(s)*. Therefore, implementation of the algorithm consists of the following steps:

Step 1: Translate the original polynomial *P(s)* one unit to the right (this can be accomplished by an algorithm consisting of a series of syntactic divisions).[3]

Step 2: Reverse the order of the coefficients.

Step 3: Translate the coefficients 1/2 unit to the left.

Step 4: Reverse the order of the coefficients again.

Step 5: Multiply the nth coefficient by 2^n.

With the algorithms presented, minimum memory and processing are required to obtain a dynamic difference equation simulation model from a linear shift-invariant continuous model. For example, Medina used a MATLAB simulation to show that the Simons-Harden algorithm(s) required 1200 FLOPS, whereas the direct MATLAB bilinear transform algorithms required 12000 FLOPS to execute a 10th-order *H(s)* to *H(z)* conversion[4]. The transformation calculations were duplicated on the TI TMS320C30 DSP.

Section 3 describes the 90 percent savings in processing required to implement these algorithms in terms of the C source code shown in the Appendix. This code was compiled and linked for execution on the TMS320C30 DSP.

3. Implementing Optimum CPDF Algorithms in C Code

Implementing the CPDF algorithms requires three distinct tasks:

1) Translating a polynomial

2) Reversing the order of coefficients

3) Multiplying each coefficient by a power of two

As described in Section 2, you can translate a polynomial using a series of syntactic divisions. This process can be implemented with the following standard difference equation:

$$a_N \leftarrow a_N \qquad a_{N-1} \leftarrow a_{N-1} + \alpha \; a_N$$
$$a_n \leftarrow a_n + \alpha \; a_{n+1} \qquad \text{for } n=N-2, \ldots, 0$$

where:

N = order of the polynomial
a = vector containing the coefficients of the polynomial

This difference equation must be evaluated *N* times. The first time the index *n* ranges from 0 to *N-1,* the second time *n* ranges from 0 to *N-2,* and so on until the process is performed *N* times. The C code implementation of the Step 1 translation algorithm is shown in Example 1 (standard matrix notation is used here only to enhance readability; pointers are used in the actual implementation).

Example 1. C Code Implementation of the Step 1 Translation Algorithm

```
Alpha = -1;
for (k=0; k< order; k++) {
        for (n=l; n<=(Order-k); n++) {
        Polynomial[n] = Polynomial[n] + Alpha * Polynomial[n+l];
     }
 }
```

Step 2 in the CPDF algorithm implementation reverses the order of the coefficients of the polynomial. Step 2 is not performed as a separate task; instead, the coefficients are reversed indirectly in Step 3 (translating the coefficients 1/2 unit to the left).

Step 3 is similar to Step 1 with two important modifications:

- The coefficients are accessed backwards to accomplish Step 2.
- Step 4, which consists of reversing the order of the coefficients, is also accomplished.

As a result, Step 3 is a more manipulated version of Step 1 that allows for a more efficient code by skipping Step 2 and Step 4. Example 2 shows the C code implementation for Step 3:

Example 2. C Code Implementation for Step 3

```
Alpha = 0. 5;
for (k=0; k< Order; k++) I
      for (n=0; n<=(Order-k-1); n++)
         Polynomial[Order + n - 1] += Alpha @ Polynomial[Order - n];

   }

}
```

Step 5 (multiplying the nth coefficient by 2^n) multiplies the nth coefficient by 2^n. The following example shows the C code implementation using the *pow* function, which returns 2^n (note that the trivial case of multiplying by 2^0 is skipped and the case of multiplying by 2^1 is performed directly).

Example 3. C Code Implementation Using the pow *Function*

```
Polynomial[l] *= 2;
for (k=2; k < Order; k++)(
            Polynomial[n] *= pow(2,n);
}
```

Recall that the scaling factor was assumed to be unity in the derivation of the CPDF algorithms described in Section 2. The scale factor *2/T*, or *C*, is handled by multiplying the nth coefficient by C^n before the above algorithm is applied.

Example 4. Handling the Scale Factor

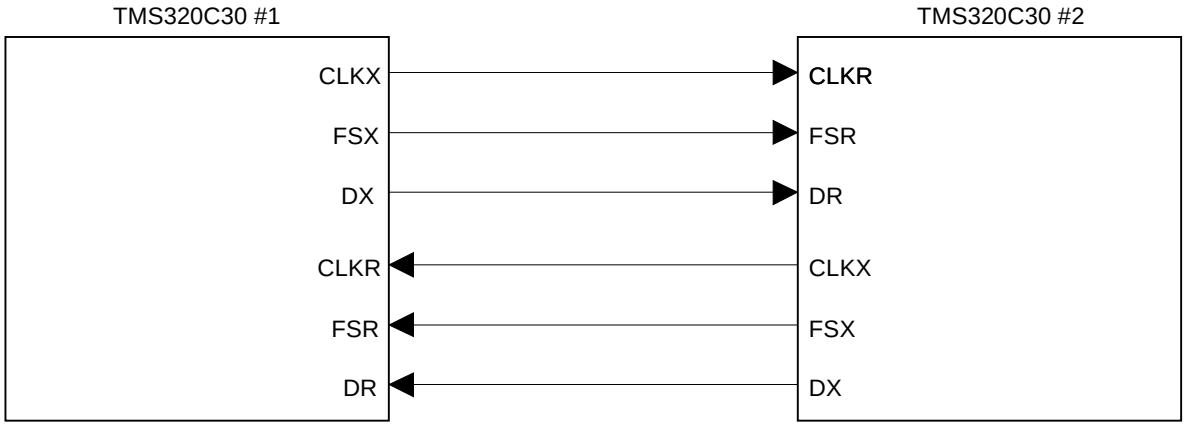
```
Polynomial[l] *= C;
      for (k=2; k <= Order; k++)
            Polynomial[n] *= pow(C, n);
```

Section 3 describes the interfacing issues that occur between multiple DSPs. These issues must be addressed before proceeding with a CPDF application.

4. Interfacing Multiple TMS320C30 DSPs

The prototype design uses a dual processor architecture that directly connects two TMS320C30 DSPs via the serial ports in the handshake mode by tying CLKX to CLKR, FSX to FSR, and DX to D (see Figure 1).

Figure 1. Interface Connections for Multiple TMS320C30 Architectures



The handshake mode of dual processor operation was chosen over other modes (for example, serial and parallel operation) for its simplicity. The handshake mode requires no external hardware setup and is the most rapid means of splitting up tasks for a dual processor architecture.

In the handshake mode, data is transmitted from the TMS320C30 #1 out of the serial port with a single leading 1. TMS320C30 #2 receives the data and transmits a single 0 back to the transmitting TMS320C30 #1 to acknowledge it has received the data. This process is handled automatically. The C code used to accomplish this task consisted of setting up the serial port global control register for the handshake mode of operation and statements to transfer the data across the serial link. Example 5 shows a brief section of pseudo C code used to accomplish this task.

Example 5. Section of Pseudo Code Used for Dual Processing

Transmit side (Processor 1)

```
spgr -> controlword = OxOEBC0064 /* handshake mode with 32 bit transfers */
while(1) {
while (!((spgr -> controlword) & OxO2)) /* wait for tx buffer emptv */
coeff-xmtd -> controlword = new-coeff.. /* txmt updated coefficient */
```

Receive side (Processor 2)

```
spgr -> controlword = OxOEBC0064;     /* handshake mode with 32 bit transfers */
while(1){
while (!((spgr -> controlword) & OxOl));   /* wait for rcv data ready
coeff-rcvd = (rcv-reg -> controlword);  /* read new coefficient
}
```

A single processor architecture was first attempted to operate the CPDFs, but a certain amount of processor time was allotted to the calculation of coefficients. A dual architecture was then tried to allow one processor to stand alone and calculate coefficients while the other implemented the CPDF. This approach freed up the filtering processor for filter operations only while the other processor updated coefficients.

5. Execution of a CPDF TMS320C30 Demonstration

A demonstration program was written to measure the accuracy and performance of the CPDF algorithms. The coefficients of two *s*-domain, all-pole transfer functions were defined: one transfer function was an implementation of a fourth-order Butterworth low-pass filter; the other transfer function was an implementation of a fourth-order Chebychev low-pass filter.

Starting with the Butterworth filter, the transfer function was converted to a digital filter using the bilinear transform (CPDF) algorithms. This filter uses a direct form structure to process input data. The current filter (in analog form) was linearly interpolated toward the Chebychev filter by a fixed increment. This new analog filter was then converted to a digital filter and the process continued. When the interpolation process reached the Chebychev filter, the filter was interpolated back toward the Butterworth filter by the same fixed increment. This process continued as long as data came in.

The demonstration program was written completely in C using a standard editor. The TI TMS320C30 C Compiler and Linker (PC release 4.60) generated the output file needed for the TI Evaluation Module. The output file was debugged using the TMS320C30 EVM C Source Debugger (release 5.0) running on a Gateway 2000 P5-100 host computer equipped with a TI TMS320C30 Evaluation Module. The program was checked for accuracy by comparing how specific variables changed to hand-calculated values. The program worked as specified after the errors were corrected.

The performance of the direct-form filtering function and the bilinear transform (CPDF) function were determined with the debuggers *clk* command. A breakpoint was set at the start of the filtering function and at the end of the function. After running to the first breakpoint, a `runb` command was executed to run to the next breakpoint. The *? clk* command was then executed to determine the number of CPU clock cycles consumed by the portion of code between the two breakpointed C statements[5]. This procedure was repeated for the bilinear transform function. To determine the requirements as a function of filter complexity, the order of the filter was changed from 2 to 10 in increments of 1.

The program was compiled for each filter order. The number of CPU clock cycles was determined for each filter order. CPU clock cycles were converted to kHz by using the cycle time for the processor. The TMS320C30 has a 60 ns, single-cycle execution time. Frequency was calculated by taking the inverse of the product of the number of CPU clock cycles required and the cycle time. The resulting data is listed in Table 1 and shown in Figure 2 and Figure 3.

Table 1. CPDF TMS320C30 Demonstration Results

Order	Filter (Clocks)	Filter (kHz)	CPDF (Clocks)	CPDF (kHz)
2	134	124.0	1118	14.9
3	171	97.5	2022	8.24
4	208	80.1	2974	5.60
5	245	68.0	3974	4.19
6	282	59.1	5022	3.32
7	319	52.2	6118	2.72
8	356	46.8	7262	2.30
9	393	42:4	8454	1,97
10	430	38.8	9685	1.72

Figure 2. Filter Update Rates

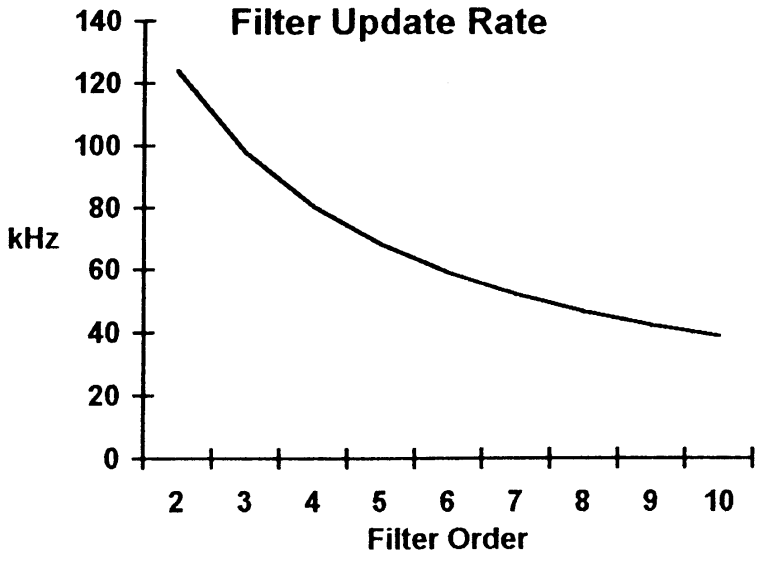
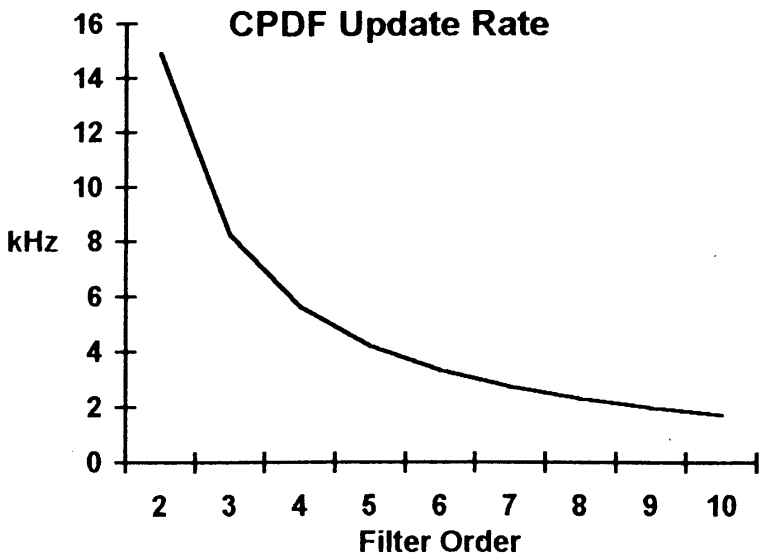


Figure 3. CPDF Update Rates

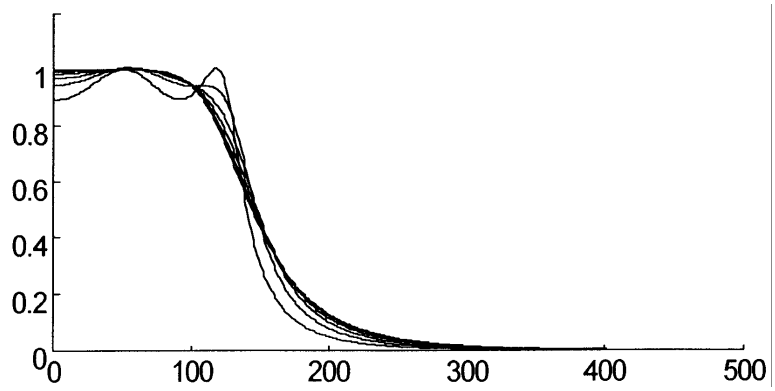


This demonstration was chosen as one familiar to DSP analysts. In particular, the fourth-order IIR Chebychev and Butterworth filter *H(z)* models were derived and implemented with the typical design cycle from standard filter algorithms; C source code file generation (see the Appendix); C file compilation, linking, and debugging; downloading to a TMS320C30 SBE (single-board computer) development system; and execution.

Although the usual development board system monitoring validated program execution, a graphical output of the demonstration program could not be captured using the TMS320C30 Evaluation Module on a host PC. For this reason, the demonstration program was simulated with MATLAB for Windows v4.2cl.

MATLAB was also used to evaluate the frequency response of the CPDF. The MATLAB Notebook v1.0 program was used to import the plot generated from the MATLAB simulation program. Figure 4 shows a fourth-order Butterworth plot linearly interpolated toward a fourth-order Chebychev in five increments.

Figure 4. Fourth-Order Butterworth Plot



This basic demonstration was extended to variable order as described earlier in this section to derive the set of CPDF design curves presented in Figure 2 and Figure 3.

6. Evaluating Design Specifications for CPDF Applications

To understand how to use the design curves, consider the requirement to design a fifth-order CPDF. Based on the filter update rate curve shown in Figure 2, the maximum sample rate is 7O kHz. This implies that the filter bandwidth is limited to approximately 7 kHz, assuming a 10:1 sample-to-signal bandwidth rate.

Alternately, the coefficient update rate is limited to about 4 kHz, based on the CPDF update rate curve shown in Figure 3. Thus, coefficients can be updated once for every two sequential outputs. A multitude of design specification combinations can be evaluated. For example, if the update rate spec is @ 4 kHz, a coefficient update for each filter output can be achieved.

7. Summary and Conclusions

The concept of continuously updating the coefficients of a digital filter leads to the identification of a broad range of applications, including high-tech (such as optimal filters, Kalman filters, and adaptive filters). A dual architecture TMS320C30 configuration has been proposed, demonstrated, and evaluated for implementing CPDFs. Based on this presentation, the following conclusions apply:

- The Simons-Harden highly efficient coefficient updating algorithms provide the basis for feasible real-time CPDFs.
- TMS320C30 architectural issues are addressed, which provides a means of mapping CPDF algorithms onto dual DSP configurations.
- CPDF processing requirements were used to formulate design specification curves with which an analyst can evaluate CPDF feasibility from complexity parameters and processing speed requirements.
- A demonstration example was implemented and documented which validated all CPDF steps.

8. Two Advanced CPDF Applications

This section describes the following applications, both of which are based on the CPDF concepts described in this application report:

- CPDF optimal filter design procedure[6]
- Kalman filter example[7]

8.1 CPDF Optimal Filter Design Procedure

The highly efficient updating algorithms present a new and interesting CPDF application opportunity. Specifically, optimal *H(z)* digital filters can be designed from analog *H(s)* prototype models to meet frequency domain criteria in the *z*-domain while applying optimization techniques to the *s*-domain coefficients.

The optimization process used to design an optimal digital filter begins with the selection of an initial *H(s)* prototype approximation to meet a specified error or cost function *H(z)* criteria in the frequency domain (for example, a low-pass filter with the cutoff frequency criteria provided if a low-pass type filter is the targeted design). Then the highly efficient coefficient updating algorithms are used to derive the *H(z)* (for the *H(v)* prototype) and evaluate the *H(z)* frequency response.

In the remote case the cost function evaluation meets the design criteria, the initial *H(s)* implies that the corresponding *H(z)* is the optimal filter. Otherwise, iterative evaluations of the *H(z)* cost functions are evaluated as a function of *H(s)* coefficient perturbations so that parameter optimization techniques can be applied to arrive at an optimal filter.

Robinson applied the method of undetermined coefficients to design optimal digital filters.[6] His optimal filter design process was adapted to the CPDF architecture with the *H(z)* filter implemented on one processor while the other processor performed all transformation routines, error calculations, and error gradients. Thus, the dual DSP CPDF architecture provided an efficient and optimal digital filter design tool.

8.2 A Kalman Filter Example

The Kalman filter falls under the general class of CPDFs. The Kalman filter is an optimal noise filter suited to a wide range of high-tech applications and is commonly used in control systems both as a state estimator and as a noise filter. New high-speed processors have increased the number of possible real-time Kalman filter applications.

As an aid in the application of real-time Kalman filters, Lamb developed a method to determine the computational requirements of a filter based on the order of the filter and the number of filter inputs.[7] This method allows the control system analyst to determine the computational requirements of a proposed real-time Kalman filter prior to actual filter formulation and testing. More importantly, the analyst can use the Kalman filter order and the number of filter inputs as complexity specifications to define a CPDF DSP architecture suitable for implementing the time-varying or simplified time-invariant Kalman filter in real time. Thus, the design curves in Lamb's thesis provide the means to determine quickly and easily the feasibility of a prospective Kalman filter application.

9. CPDF C Source Code Files

The demonstration program shown in Example 6 was written completely in C using a standard editor. The Texas Instruments TMS320C30 C Compiler and Linker (PC release 4.60) was used to generate the output file needed for the Texas Instruments Evaluation Module. The output file was debugged using the TMS320C30 EVM C Source Debugger (release 5.0) running on a Gateway 2000 P5-100 host computer equipped with a Texas Instruments TMS320C30 Evaluation Module. The program was checked for accuracy by comparing how specific variables changed to hand-calculated values.

The requirements used to generate the output file were carefully documented to ensure reproduction (for example, the software titles with their revisions and hardware components, including the host computer specifics).

Example 6. CPDF "C" SOURCE CODE LISTINGS

CPDF C Source File TI.C

```
/* Function Declarations */
void BilinearTx(double *, double *, int, double );
void GetFilter(double *, double *, double);
void Filter(double *, double *, int, float *, int);

/*Constants */
#define ORDER 4
#define BUFFERSIZE 10


main() {

      double C, Input, Output;
      double DigitalNum[ORDER+l] = [0), DigitalDen[ORDER+l] = {0};
      float Buffer[BUFFERSIZE) = {0};
      int BufferSize, n, Continue;
        Continue = 1;
      Input = 1;
            /*Calculate the constant for the Bilinear Transform, C */
            /*Normalized frequency W = 1*/
            /*Let the cutoff frequency w = 2*pi*0.75*/
            /*Let the sample period T = 1/2 */
            /*C = W cot(w T/2) = cot(2*4*atan(l)*O.75/2/2) =
            cot(l.5*atan(l))
*/

      C = 1/tan(l.5*atan(l));

      /*****************************************/
      /* Main Loop of Program
            Get/Update Filter Coefs
            Fill input buffer
            Filter Buffer
            Output Buffer
            Repeat Until input buffer is empty */

BufferSize=BUFFERSIZE;
   do    {

      /*Get Filter Coefficients*/
      GetFilter(DigitalNum, DigitalDen, C);

      /*Read s samples into buffer. */
      for(n = 0; n < BufferSize; n++) i
            *(Buffer+n) = Input;
      }

      /*Filter Buffer
      Filter(DigitalNum, DigitalDen, ORDER, Buffer,
            BufferSize);

      /*Output Buffer*/
      for(n = 0; n < BufferSize; n++) {
            Output = *(Buffer + n);
      }
      while(Continue);
}
/************************End of Main() ***********************************
```

```
/************************GetFilter () ************************************
void GetFilter(double *Num, double 'Den, double C)   {

      /*Analogl is a 4th ORDER ButterWorth normalized LP'/
      /*Analogl is a 4th ORDER Chebyl normalized LPI/
      /* These polynomials are arranged [aN aN-1... aO } (like Matlab) */
      double AnalogNuml[] ={O, 0, 0, 0, 1};
      double AnalogDenl[] = {1, 2.6131259, 3.4142135, 2.6131259, 1};
      double AnalogNum2[] ={0, 0, 0, 0, 0.245916};
      double AnalogDen2[] = {1, 0.953, 1.454, 0.743, 0.276};
      static int first call = 1, direction = 1, i=O;
      const int Steps L-- 3;
      int n;

      /*If this is the first call then load the Analogl Coefs. */
      if(first - call)
            first - call = 0;
            memcpy(Num, AnalogNuml, (ORDER+l)* sizeof (double));
            memcpy(Den, AnalogDenl, (ORDER+l)* sizeof (double));
      }

/* Otherwise, ramp the past coefs towards Analogl or 2 depending on
            direction*/
      else  {

            i += direction;
            if(i==Steps)          {  /* Must be Analog2  */
                  direction = -1;
                  memcpy(Num,  AnalogNum2, (ORDER+l)* sizeof (double));
                  memcpy(Den,  AnalogDen2, (ORDER+l)* sizeof (double));
            }
            else if(!i)        /* if i = 0 must be Analogl
                  direction = 1;
                  memcpy(Num, AnalogNuml, (ORDER+l)* sizeof (double));
                  memcpy(Den, AnalogDenl, (ORDER+l)* sizeof (double));
            }
            else  {         /*Otherwise between Analogl and 2*/
                  for(n = 0; n <= ORDER; n++)  {
                        Num[n) =*(AnalogNuml + n) + i * ( *(AnalogNum2 + n)-
                              *(AnalogNuml + n) ) /Steps;
                        *(Den + n) = *(AnalogDenl + n) + i * ( *(AnalogDen2+
n) -
                              *(AnalogDenl + n) ) /Steps;
                  )
            }
      }
      /*Obtain Digital Filter from the analog coefs*/
      BilinearTx(Num, Den, ORDER, C);



}
/*********************** End of GetFilter () *****************************/
```

```
/************************ Bi1inear Tx () ********************************/
void BilinearTx( double *dNumCoefs, double *dDenCoefs, int norder, double dC)
      {
      /* Note: the size of the array is norder + 1 */

      int n, m;
      float falpha;
      double Temp;

      /*an <- C'n * an */
   *(dDenCoefs + nOrder-1) *= dC;
   for (n=O; n<nOrder-1; n++)
            *(dDenCoefs+n) *= pow(dc, nOrder-n);

      /* E(z) = P(z - 1) */
      falpha = -1;
      for (n=O; n< nOrder; n++) {
            for (m=l; m<=(nOrder-n); m++) {
                  *(dDenCoefs + m)+= falpha * *(dDenCoefs + m-1);
                  *(dNumCoefs + m)+= falpha * *(dNumCoefs + m-1);
            }
      }

      /* F(z) = z^N E(l/z) */
            /*Compensated for*/
      /* G(z) = F(z + 112) */
      falpha = (float)0.5;
      for (n=O; n< norder; n++) {
            for (m=O;.m<=(nOrder-n-1); m++)    {
                  *(dDenCoefs + nOrder-m-l)+= fAlpha * *(dDenCoefs + norder-
                  m);
                  *(dNumCoefs + nOrder-m-l)+= fAlpha * *(dNumCoefs + nOrder-
                  m);
            }
      }
      /* J(z) = z'N*G(l/z) */
                  /*Compensated for*/
      /* D(z) = J(2*z) */
      for (n=O; n < norder; n++){
            *(dDenCoefs + n) *= pow(2,nOrder-n);
            *(dNumCoefs + n) *= pow(2,nOrder-n);
      }

      /*Adjust coefs so a(O) = 1*/
      Temp = *(dDenCoefs);
      if( Temp != 1.)   {
            for (n=O; n<=nOrder; n++)       {
                  *(dDenCoefs + n) /=   Temp;
                  *(dNumCoefs + n) /=   Temp;
            }
      }
}
/************************End of Bilinear Tx () ***************************/
```

```
/************************ Filter () ************************************/
void Filter(double *b, double *a, int Order, float *Buffer,
      int BufferSize) f

     static double History[ORDER+1]; /* Use static so the History is
remebered
                                                                  between

calls */
      float temp;
      int n, k;

      for(n=O; n < BufferSize; n++) {
            /*w(n) = x(n) - sum(k=l to N) of ak * w(n-k) */
            temp = *(Buffer+n);
            for(k=l; k<=Order; k++)
                  temp -= *(a + k) * History[k – 1};

            /*Only need to keep N History's {w(n - k)'s},
                  so shift them all down one
                  to reduce memory requirements, rather than keeping them
                  all.
                  All the w(n)'s that is */
            for(k=Order; k > 0; k--)
                  History[k) = History[k – 1];
            History[O] = temp;

            /*Use the same buffer for input and output*/
            *(Buffer+n) = 0;
            /* y(n) = sum(k=O to L) of bk * w(n - k) */
            for(k = 0; k<=Order; k++)
                  *(Buffer+n) += *(b+k) * History[k);
      }
}
/*************************End of Filter () *******************************/
```

References

[1] Simons, Jr., F.O.; George, A.D.; Medina. J.; and Freatfiv, B.A.; "Design and Simulation of Continuously Programmable Digital Filters", *Proceedings Of the 26th Southeastern Symposium on System Theory;* University of Ohio; Athens, Ohio; March 20-22, 1994.

[2] Simons, Jr., F.O. and Harden, R.C.; "An Optimized Simulation of Dynamic Continuous Models", *Proceedings of the 96th Annual ASEE Conference,* Portland, Oregon, June 19-23, 1988.

[3] D'Azzp, J.J. and Houpis, L.H., Feedback Control Systems Analysis and Synthesis. 2nd edition. McGraw-Hill Book Co.. 1966

[4] Medina. J., unpublished MATLAB work, HCS Research Lab, FAMU-FSU College of Engineering, 1994.

[5] Edited, *TMS320C3x C Source Debuggers Guide,* Texas Instruments, Inc.. 1993.

[6] Robinson. A., "The Design of Optimal Digital Filters Directly From Analog Prototypes", *M.S. Thesis,* FAMU-FSU College of Engineering, Tallahassee, Florida. 1996.

[7] Lamb, J.M., "Implementation of Kalman Filters on Microprocessor-based Digital Signal Processors". *M.A. Thesis*, FAMU-FSU College of Engineering, Tallahassee, Florida, 1995.