Digital Signal Processing (DSP) is centered around the idea that you can convert an analog signal to a digital signal that is discrete in both time and amplitude, process the resulting digital signal with a computer, and convert the digital result back to an analog signal (usually). A typical DSP system consists of the blocks shown below.

![Fundamental elements of a DSP System](image)

Typically the computer evaluates a difference equation of the form

\[ y_k = b_0 u_k + b_1 u_{k-1} + \cdots + b_m u_{k-M} - a_1 y_{k-1} - a_2 y_{k-2} - \cdots - a_N y_{k-N} \]

In this equation \( k \) is the time counting variable so that samples are taken at 0, T, 2T, 3T ... \( kT \) ... The variable \( y_k \) is the output variable at the time \( kT \). The variables \( y_{k-1}, y_{k-2}, \ldots y_{k-N} \) are the previous output variables. The variables \( u_k, u_{k-1}, \ldots u_{k-M} \) are the input variables both past and present. The coefficients \( a_x \) and \( b_x \) are constants.

The computer program which evaluates this difference equation might look something like this in pseudocode.

```
Initialize Variables
DO Forever
    Call AtoD(Vi) ;Get a sample from the A to D
    Vo = {difference equation} ;Output Vo to D to A
    CallDtoA(Vo)
    Vol = Vo ;Reset the value of the old variable.
    Wait for T seconds to pass
Loop
End
```
The loop runs forever and the loop timing is typically governed by a polled or interrupt driven timer. The processing time for the difference equation evaluation is the most time consuming part of the operation and it is this processing time which determines the fastest sampling period since samples cannot be brought in faster than the machine can process them.

Looking at the difference equation we see that it consists of a sequence of multiplies and adds. Such operations are a mainstay of DSP systems and nearly all specialized DSP chips have a single hardware unit which does a multiply and add operation which is called a MAC for multiply and accumulate.

A few general purpose processors have become fast enough to do DSP algorithms and some have added MAC units as part of the CPU make up. The ARM7 processor is one such CPU that has a MAC unit (in addition to its barrel shifter). The MAC unit on the ARM7 can multiply two signed 32-bit integers and add the product to a third register. The ARM7 also has long integer version of this instruction.

In assembly language the multiply and accumulate instruction looks like this

\[
\text{mla \ rd, rm, rs, rn}
\]

This instruction multiplies does the following \( \text{rd} \leftarrow \text{rm} \times \text{rs} + \text{rn} \)

Unfortunately, the Keil C compiler does not make use of the mla instruction when it evaluates a difference equations but creates code which does the multiplies and as separate assembly operations. So to take advantage of the mac unit on the ARM7 you have to write assembler code.

Creating a digital filter using MatLab

While this is not a course on DSP (for that you need to take EE 311), we can use Matlab as a tool to create useful difference equations which represent digital filters. We will look at two types: Butterworth filters and Chebyshev filters.

Butterworth filters in MatLab

A Butterworth low pass filter is maximally flat in the passband, has a relatively long transition band, and is monotonic in the stop band. In Matlab the relevant function is

\[
\text{[num \ den] = butter(N, wn)}
\]

In this equation num and den are the numerator and denominator polynomials in z for the Butterworth filter, N is the filter order, and wn is the normalized cutoff frequency. (The cutoff frequency is normalized so that \( f_s/2 = 1 \).)

For example,

\[
\begin{align*}
N &= 3 \quad \text{%filter order} \\
fs &= 11050; \quad \text{%sample frequency} \\
wn &= 3000/(fs/2); \quad \text{%normalized cutoff frequency} \\
[num \ den] &= \text{butter}(N, \ wn); \quad \text{%calculate filter} \\
disp(num); \quad \text{%display numerator and denominator} \\
disp(den);
\end{align*}
\]
gives
>> butter1
  0.2025  0.6075  0.6075  0.2025
  1.0000  0.2477  0.3496  0.0228

which translates into the following transfer function in z

\[
H(z) = \frac{y(z)}{u(z)} = \frac{0.2025z^3 + 0.6075z^2 + 0.6075z + 0.2025}{z^3 + 0.2477z^2 + 0.3496z + 0.0228}
\]

To translate this into a difference equation we first multiply numerator and denominator by \(z^{-3}\). This gives

\[
H(z) = \frac{y(z)}{u(z)} = \frac{0.2025 + 0.6075z^{-1} + 0.6075z^{-2} + 0.2025z^{-3}}{1 + 0.2477z^{-1} + 0.3496z^{-2} + 0.0228z^{-3}}
\]

Cross multiply this equation to get

\[
y(z) + 0.2477z^{-1}y(z) + 0.3496z^{-2}y(z) + 0.0228z^{-3}y(z) = 0.2025u(z) + 0.6075z^{-1}u(z) + 0.6075z^{-2}u(z) + 0.2025z^{-3}u(z)
\]

Noting that the inverse z transform of \(y(z) \rightarrow y_k\) and the inverse transform of \(z^3y(z) \rightarrow y_x\) we can transform this equation to the time domain and solve for \(y_k\). This gives:

\[
y_k = 0.2025u_k + 0.6075u_{k-1} + 0.6075u_{k-2} + 0.2025u_{k-3}
- 0.2477y_{k-1} - 0.3496y_{k-2} - 0.0228y_{k-3}
\]

This equation can be implemented on the ARM7 board in floating point mode or, if it is rescaled, in integer mode. Integer mode is faster but is sometimes difficult to scale without producing overflow. Floating point mode is slower but overflow is seldom a problem.

We can also use MatLab to look at the expected frequency response of this filter by plotting the filter gain vs. frequency. The following Matlab code does this.

```matlab
%Butter1.m
N = 3;                       %filter order
fs = 11050;                  %sample frequency
wn = 3000/(fs/2);            %normalized cutoff frequency
[num den] = butter(N, wn);   %calculate filter
disp(num);                   %display numerator and denominator
disp(den);
%freqz finds the frequency response for 512 points between 0 and fs/2
[H f] = freqz(num, den, 512, fs);
figure(1);clf;
plot(f, abs(H));
xlabel('frequency in Hz');
ylabel('filter gain');
title('3rd Order Butterworth filter');
```
Note that the cutoff frequency is at the point where the gain is .707 (the half power point).

**Chebyshev filters in MatLab**

A Chebyshev low pass filter is has some ripple in the passband, has a faster transition band than does the Butterworth filter, and is monotonic in the stop band. In Matlab the relevant function is

\[
\text{[num den]} = \text{cheby1}(N, \text{Rdb}, \text{wn});
\]

In this equation num and den are the numerator and denominator polynomials in z for the type 1 Chebyshev filter, N is the filter order, Rdb is the passband ripple in decibels, and wn is the normalized passband edge frequency. (The passband edge frequency is normalized so that \( f_s/2 = 1 \)). Here is a Matlab program to calculate and plot a 3\(^{rd}\) order Chebyshev low pass filter.

```matlab
%Chebyshev1.m
N = 3;                       %filter order
fs = 11050;                  %sample frequency
wn = 3000/(fs/2);            %normalized passband edge frequency
ripple = 0.05;               %pass band peak to peak ripple
rDB = -20*log10(1-ripple);   %convert ripple to decibels
[num den] = cheby1(N, rDB, wn); %calculate filter
disp(num);                   %display numerator and denominator
disp(den);
%freqz finds the frequency response for 512 points between 0 and fs/2
[H f] = freqz(num, den, 512, fs);
figure(1);clf;
plot(f, abs(H));
xlabel('frequency in Hz');
ylabel('filter gain');
title('3rd Order Chebyshev filter');
```

Note that the cutoff frequency is at the point where the gain is .707 (the half power point).
The numerator and denominator polynomials are
\[ 0.1997 \quad 0.5992 \quad 0.5992 \quad 0.1997 \]
\[ 1.0000 \quad 0.1798 \quad 0.4985 \quad -0.0804 \]

The resulting frequency plot for the filter is shown below.

**Class exercise 10-1:** The program ChebF3.c implements the 3rd order Chebyshev filter above as a digital filter running on the ARM7 processor but it has no timer to make the sample time correct. Modify this program to use a timer0 either in the polled mode or in the interrupt mode to get a sample frequency of 11025Hz.
This program implements a filter in floating point number arithmetic. This is for ARM7 processor board. The filter is designed to have a 3rd order Chebyshev response with

\[
H(Z) = \frac{0.199727Z^3 + 0.5991816Z^2 + 0.5991816Z + 0.199727}{Z^3 + 0.1797521Z^2 + 0.4985085Z^2 - 0.0804431}
\]

This filter was designed in MatLab as a 3rd order Chebyshev filter with

- \(f_{\text{pass}} = 3000\)
- \(R_p = 0.05\)
- \(N = 3\)
- \(w_n = 3000\)

```c
#include <LPC213X.H>
const float b0 = 0.1997272;
const float b1 = 0.5991816;
const float b2 = 0.5991816;
const float b3 = 0.1997272;
const float a1 = 0.1797521;
const float a2 = 0.4985085;
const float a3 = -0.080443;
void main(void)
{
    unsigned int uInt;
    unsigned int dtoaOut;
    float u, y;
    float u1, u2, u3;
    float y1, y2, y3;
    VPBDIV = 0x02;        //Set the Pclk to 30 Mhz
    AD0CR  = 0x00200601;  //Setup A/D:10-bit AIN0 @ 4.28MHz software cntrl
    AD0CR  |= 0x01000000; // Start A/D Conversion
    PINSEL1 = 0x00480000;// P0.25 set to DA Out, P0.27 set to input AD0.0
    dtoaOut = 0;
    while(1)
    {
        uInt = AD0DR;       // Read A/D Data Register
        while ((uInt & 0x80000000) == 0)  //Wait for the conversion to end
            uInt = AD0DR;
        uInt = ((uInt >> 6) & 0x000003FF);//Shift into position for 10-bit
        u = (float)uInt;     //1024.0;
        u = u/1024.0;
        y = b0*(u + u3) + b1*(u1 + u2);
        y = y - a1*y1 - a2*y2 - a3*y3;
        DACR = (int)(y*1024.0)) << 6;
        y3 = y2;             //Shift variables for next
        y2 = y1;             // iteration
        y1 = y;
        u3 = u2;
        u2 = u1;
        u1 = u;
        AD0CR  |= 0x01000000; // Restart A/D Conversion
    }
}
```
## Fourier Transform Summary

<table>
<thead>
<tr>
<th>Frequency Domain</th>
<th>Time Domain</th>
<th>Fourier Series</th>
<th>Discrete Fourier transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous in Time</td>
<td>Periodic</td>
<td>$f(t) = \sum_{k=\infty} C_k e^{j2\pi k T} t$</td>
<td>The DFT is the same as the Fourier transform if both the time and frequency variables are made discrete. For the DFT both the time and frequency domains become periodic. Since all variables are discrete, the DFT is convenient for computer calculations. The fast Fourier Transform (FFT) provides an efficient algorithm for calculating the DFT.</td>
</tr>
<tr>
<td>Discrete in Time</td>
<td>Periodic</td>
<td>$C_k = \frac{1}{T} \int f(t) e^{-j2\pi k T} dt$</td>
<td>Discrete in time Periodic</td>
</tr>
<tr>
<td>Continuous in Frequency</td>
<td>Not periodic</td>
<td>$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt$</td>
<td>Discrete in frequency Periodic</td>
</tr>
<tr>
<td>Discrete in Frequency</td>
<td>Not periodic</td>
<td>$F(k) = T \sum_{n=0}^{N-1} f(nT) e^{-\frac{j2\pi kn}{N}}$</td>
<td></td>
</tr>
</tbody>
</table>

### Fourier Transform

The Fourier transform can be derived from the Fourier series by allowing the period to go to infinity. The Fourier transform shows the frequency makeup of non-periodic signals. The time function is continuous as is the frequency domain transform. If $C_k$ is the value of the Fourier series coefficients for a function with period $T$, then the Fourier transform is the limit of $T C_k$ as $T$ goes to infinity. The Fourier transform can be "generalized" by replacing $e^{j2\pi k T}$ with the complex variable $s$. The result is the Laplace transform.

$$F(s) = \int_{-\infty}^{\infty} f(t) e^{-st} dt$$

### Discrete Time Fourier Transform

The DTFT is the same as the Fourier transform except that the time variable is discrete. The frequency spectrum of the resulting non-periodic discrete-time signal is continuous and periodic. The DTFT can be "generalized" by allowing a new complex variable, $z$, to replace $e^{j2\pi k T}$. This results in the z-transform.

$$F_{\text{DTFT}}(z) = \sum_{n=0}^{\infty} f(nT) z^{-n}$$

$$F_{\text{DTFT}}(z) = \int_{-\infty}^{\infty} F(s) e^{j2\pi s} ds$$
The LaPlace Transform
The LaPlace transform is derived from the Fourier transform by replacing $j\omega$ by $s$ where $s$ is an arbitrary complex number having both a real part, $\sigma$, and an imaginary part $\omega$.

$$F(s) = \int_{-\infty}^{\infty} f(t) e^{-st} dt$$

This is the defining equation for the LaPlace transform. It’s counterpart, the inverse LaPlace transform, is:

$$f(t) = \frac{1}{2\pi j} \int_{\sigma-j\infty}^{\sigma+j\infty} F(s) e^{st} ds$$

The Z transform
The z-transform is derived from the DTFT where we generalize and replace $e^{j\omega T}$ with $z$ where $z$ is an arbitrary complex number.

$$F(z) = \sum_{n=-\infty}^{\infty} f(nT) \cdot z^{-n}$$

The inverse z transform can be derived using the Cauchy integral theorem from complex variables. The inverse z transform is:

$$f(n) = \frac{1}{2\pi j} \oint_{c} F(z) z^{n-1} dz$$