View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Missouri University of Science and Technology (Missouri S&T): Scholars' Mine

Missouri University of Science and Technology
Scholars' Mine

Electrical and Computer Engineering Faculty Research & Creative Works

Electrical and Computer Engineering

01 Jan 1988

Implementation of DSP Applications Using the AT&T DSP32 C Compiler and Application Library

J. Tow

J. Hartung

Steven L. Grant
Missouri University of Science and Technology, sgrant@mst.edu

Follow this and additional works at: https://scholarsmine.mst.edu/ele_comeng_facwork

Part of the Electrical and Computer Engineering Commons

Recommended Citation

J. Tow et al., "Implementation of DSP Applications Using the AT&T DSP32 C Compiler and Application Library," *Proceedings of the IEEE International Symposium on Circuits and Systems, 1988*, Institute of Electrical and Electronics Engineers (IEEE), Jan 1988.
The definitive version is available at https://doi.org/10.1109/ISCAS.1988.15108

This Article - Conference proceedings is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Electrical and Computer Engineering Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

IMPLEMENTATION OF DSP APPLICATIONS USING THE AT&T DSP32 C COMPILER AND APPLICATION LIBRARY

J. Tow, S. L. Gay, J. Hartung

AT&T Bell Laboratories
Holmdel, N. J. 07733
U.S.A.

ABSTRACT

Digital Signal Processors (DSPs) have traditionally been used in real-time applications with very high data throughput. For this reason, system designers have been reluctant to accept the degradation in performance inherent to machine code compiled from high level languages such as "C". The problem is compounded by the fact that DSPs often use pipeline architectures to achieve their high data throughput resulting in hazards and latencies between instructions. Typically, instruction cycles that occur during these latencies are utilized by the programmer for tasks not involving the data in the pipeline. Simple compiler implementation cannot take advantage of latent instructions which results in a conservative and inefficient executable program. This problem has been addressed in the C compiler package for the AT&T DSP32 family by the addition of a post-optimizer and an extensive application library.

In Section 1 we briefly introduce the C Compiler and the Application Library. Section 2 contains several examples where the intent is to show the ease with which filtering applications can be generated. Finally, in Section 3, a comparison is made between compiled and assembly language coded examples.

1. DSP32 C COMPILER AND APPLICATION LIBRARY

The DSP32 C Compiler is an implementation of the UNIX portable C compiler, which guarantees portability of C programs from UNIX System V machines to the DSP32. It also facilitates upgrades to new C standards such as ANSI C, and language extensions, such as C++. Integer data types, as well as single precision floating point data types, are supported in the compiler.

A post-optimizer implements a number of strategies to improve the speed and reduce the size of compiled code. User control is also provided to improve the use of the pipeline for cases where the optimizer cannot resolve data dependencies. A more detailed description of the C Compiler and Optimizer can be found in [1].

Although a high degree of performance and software productivity is attained by the DSP32 C Compiler and Optimizer, an additional gain can be obtained by using assembly language coded functions. For this reason, a library of assembly language functions that can be called from a C program is provided. The application library currently contains over 60 C-callable functions for generic arithmetic functions (trigonometric, logarithmic, and matrix) and signal processing algorithms (fixed and adaptive filters, FFTs, and graphics/image processing). Some of the standard UNIX C functions such as "printf", and string manipulation are also included in the library. Functions are provided which optimize both code size and speed, and can be chosen by the user as appropriate. Detailed descriptions for a subset of the equivalent assembly language library functions that can be called from an assembly language program can be found in [2].

2. EXAMPLES

This section gives several examples on the use of the DSP32 C Compiler and the application library functions for some commonly encountered filtering applications.

2.1 Example 1 (LMS Algorithm)

The following example can be found on page 22 of Reference 3. A single-input adaptive FIR filter with two weights is shown in Figure 1. The input, x_k, and the desired signal, d_k, are sinusoids at the same frequency, with N samples per cycle. The LMS (least mean squares) algorithm updates the coefficients of the adaptive filter as follows,

$$w_i(k+1) = w_i(k) + 2\mu \cdot e(k) \cdot x(k-i)$$

The DSP32 application library function, **lms**, computes the following. It first calculates the convolution of the FIR filter input sequence x(k) with the estimated coefficients $w_i(k)$, resulting in the output y_k which is an estimate for d_k. The error signal, e_k, is next formed as the difference between the desired signal, d_k, and the filter output, y_k. The weights (or filter coefficients) are then updated according to the above equation for use in the next call of the function. In the example, N=16, and the coefficients are initialized (by default) to zero. The input and desired output samples are obtained from the array, **in**, which was initialized to the first 20 values of the input signal. Note that the output samples, or the cosine values, are obtained from the array as four samples ahead of the input values. The lms function returns the error term which is stored in the array, **out**, at 16 sample intervals. Some of the manual pages for the application library functions used in this paper can be found in the Appendix.

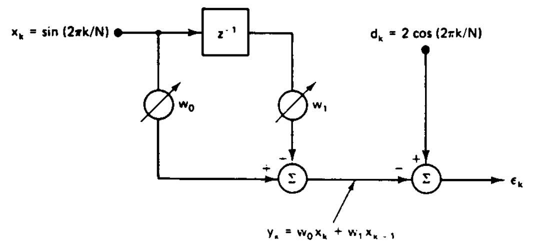

Fig. 1 An Adaptive FIR Filter With Two Weights

The C program listing follows:

```
/* Example 1 -- LMS algorithm with a 2-tap adaptive FIR filter */

#include <libap.h>
#define L       2         /* Length of LMS adaptive FIR filter */
#define lmscon 0.4        /* adaptation constant (2μ) */

float   in[20] = {
  0.3826834, 0.70710678, 0.9238795, 1., 0.9238795, 0.70710678,
  0.3826834, 0., -.3826834, -.70710678, -.9238795, -1., -.9238795,
  -.70710678, -.3826834, 0., 0.3826834, 0.70710678, 0.9238795, 1.
};
float   out[64];
float   afircoe[L];

main()
{
        static float   afirsv[L], afirout;
        static float   afirsv[L], afirout;
        register int    i, j;
        register float *p = in;
        register float *q = out;
        register float *coef = afircoe;
        register float *sv = afirsv;

        for (i=63; i-- >=0;) {
                for (j=14; j-- >=0;) {
                        lms(L, *p, 2.0* *(p+4), afirout, lmscon,
                            coef, sv);
                        p++;
                }
                *q++ = lms(L, *p, 2.0* *(p+4), afirout, lmscon,
                                coef, sv);
                p = in;
        }
}
```

The above program can be compiled with the DSP32 C Compiler and run using the DSP32 simulator or the DSP32-DS Development System [4]. At the end of 1024 passes of the LMS algorithm the value of the filter coefficients, i.e., [afircoe[1], afircoe[0]], are [4.828427, -5.226253] which closely agrees with the theoretical value of [2cot(2 π /16), -2csc(2 π /16)]. The corresponding final error value, out[63], is -7.729977e-8.

2.2 Example 2 (Example 1 with added random signal)

This example can be found on page 103 of Reference 3 where a random signal is added to the input of Example 1. The application library function, **gran**, which generates gaussian random numbers with zero mean and unity variance is used to simulate the random signal. A sigma value of 0.1, which corresponds to a random signal with average power of 0.01, is used in the example. A **printf** statement (which can only be used with the software simulator) is inserted into the C program to display, on the terminal, the trajectory of the filter coefficients and the error value after every input sample. The value of the coefficients after 256 passes of the LMS algorithm are [3.441290, -4.114841] which agrees well with the theoretical optimum result, [3.784, -4.178].

```
/* Example 2 -- Example 1 with a random signal added at
                the input */

#include <stdio.h>
#include <libap.h>
#define L       2          /* Length of the adaptive FIR filter */
#define lmscon 0.2         /* adaptation constant (2μ) */
#define sigma  0.1         /* sigma of random signal power */

float   in[20] = {
  0.3826834, 0.70710678, 0.9238795, 1., 0.9238795, 0.70710678,
  0.3826834, 0., -.3826834, -.70710678, -.9238795, -1., -.9238795,
  -.70710678, -.3826834, 0., 0.3826834, 0.70710678, 0.9238795, 1.
};
float   afircoe[L];
float   iseed = 1.0;      /* random number generator seed */

main()
{
        static float   afirsv[L], afirout;
        register int    i, j;
        register float *p = in;
        register float *coef = afircoe;
        register float *sv = afirsv;
        float e;

        for (i=15; i-- >=0;) {
           for (j=15; j-- >=0;) {
              e = lms(L, *p+sigma*gran(&iseed), 2.0* *(p+4),
                      afirout, lmscon, coef, sv);
              p++;
              printf("%f,%f, error = %f \n",afircoe[1],afircoe[0],e);
           }
           p = in;
        }
}
```

With these tools, one can easily explore other alternatives of the LMS algorithm, e.g., starting with different initial values for the coefficients, changing the filter length, or observing the rate of convergence for different adaptation constant values. In particular, this example converges reasonably well only when the value of the adaptation constant, 2μ, is less than 1.4. There is, however, a trade-off in the rate of convergence and the sample to sample variance of the converged coefficients. Therefore, for a particular implementation the user often experiments with different values for the adaptation constant. The C compiler and the application library provides a flexible tool for this type of experimentation.

2.3 Example 3 (IIR filter)

From stability considerations, IIR filters are generally implemented in a cascade of second-order direct form I or II structures. In addition to the filter response, frequently, it is of interest to know the intermediate outputs at the filter internal sections. A conceptual block diagram as shown in Figure 2 can be used for this purpose.

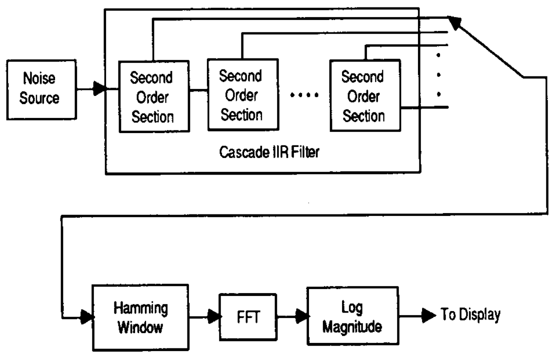

Figure 2. Example 3 Block Diagram

A white noise generator is used to excite the filter. The windowing block collects a block of output samples, e.g., 256, from the output of the filter or from one of the filter internal nodes, and then weights them with a window function. A discrete Fourier transform of the windowed data is computed by the FFT block. The log-magnitude block forms an integer array of the log magnitude of the FFT outputs and passes the data to the display.

A DSP32 C program which implements Figure 2 is given below in Example 3. The program implements a 3-section IIR direct form II filter with the use of the application library function, **iird**. The function, **ran**, which generates uniformly distributed random numbers over the range [0.0, 1.0) is used as the input signal to the filter.

```
/* Example 3 -- Direct Form II IIR filter with white noise input */

#include <libap.h>

#define L       4       /* No. of second-order sections */
#define N       256     /* No. of points in window or the FFT */
#define M       8       /* log2N */

float   iirdcoe[] = {                           /* filter coefficients */
          0.856676,  0.000000,  0.000000,  0.00000, 1.,
          0.928466, -0.0716619,  0.182630,  0.000000, 0.0716619,
          0.928466,  0.1299680  -0.182630,  0.131665, 0.1299680,
          0.00000,  0.2884230,  0.00000, -0.292188, 0.2884230
        };

float  iseed = 1.0;
float  A[2*N];     /* array for input samples and FFT output */
int    B[N/2];     /* array for log magnitude of the FFT output */

main()
{
        static float  iirdsv[2*L];
        register int    i;
        register float ar, ai;
        register float *p = A;
        register int   *q = B;
        register float *coef = iirdcoe;
        register float *sv = iirdsv;

/* Collects 256 filter outputs and stored in array A */

        for(i=255; i-- >=0; ){
                *p++ = iird(L, ran(&iseed), coef, sv);
                *p++ = 0.0;
        }
        p = A;

/* Do Hamming window */

        chamm0(N, M, p);

/* Call FFT function */

        fft(N, M, p);

/* Compute log magnitude */

        for(i = N/2 - 1; i-- >=0; ){
                ar = *p++;
                ai = *p++;
                *q++ = 10.0 * log10(ar * ar + ai * ai);
        }
}
```

In the cascade implementation of direct form II second-order sections, the outputs at the internal sections that are important for overflow and scaling considerations, are the so-called "branch" nodes [Chapter 11 of Reference 5]. The branch node output at an intermediate section corresponds to the response of the filter due to the poles of that section and all of the poles and zeros preceding that section. Therefore, one way to observe the responses at the filter and each of its branch nodes outputs, is by looking at the output of a "modified" filter as follow. The first section of the modified filter consists of the denominator term of the first section of the filter and an numerator term equals to 1. The i th section of the modified filter consists of the ith denominator term and the (i-1)th numerator term of the filter, for i = 2 to L where L is the number of second-order sections of the filter. The (L+1)th section of the modified filter consists of the Lth numerator term of the filter and a denominator term equals to 1. Hence, the coefficients in the program listing of Example 3, **iirdcoe[]**, correspond to a four-section filter.

The 256 filter output samples are stored successively in the even locations of the complex array, **A**, where the imaginary parts, or the odd locations of the array, are set to zero. The application library function, **chamm0**, is used to do an in-place Hamming window on this array. The FFT function is implemented with a call to the in-place library function, **fft**. Finally, the log magnitude of the output of the fft function is computed with the function, **log10**, and stored in the integer array, **B**.

The 3-section filter used in the above example has the following specification: an 8 kHz sampling rate, a passband between 1900 Hz and 2100 Hz with a ripple less than 0.1 dB, and a stopband below 1400 Hz and above 2600 Hz with loss greater than 50 dB. The filter coefficients were obtained from [6] using the direct form I design.

The program in Example 3 can be successively compiled and run with the value of L in the #define statement set to 4, 3, 2, and 1 for the calculation of the filter output and the branch node output at section 3, 2, and 1, respectively. With the program interfaced to a PC for the display of the output, as described in the example of [7], the spectrum (with multiple traces of 256 samples) for L = 4, 3, 2, and 1, is shown in Figures 3a to 3d, respectively. Notice that the peak output at the branch node of section 2 is approximately 20 dB higher than that of the filter peak output. This is due to the fact that the program implements a direct form II algorithm while the filter coefficients were scaled according to the direct form I design.

3. HOW EFFICIENT ARE THE C PROGRAMS ?

Programs written in a high level language, such as C, incur an overhead as compared to those written in the processor assembly language. For signal processing applications, an important consideration is the program execution time. In this paper, we express efficiency (in %) as the ratio of the execution time of a DSP32 assembly language program to an equivalent program written in the C language. The DSP32 application library contains over 50 assembly language functions [2]. For each function, Reference 2 also contains an assembly language test program which calls the function a multiple number of times. Each of these assembly programs has been recoded in C using the C-callable application library. A comparison of the execution times for these programs is summarized in Table 1.

A few comments are in order here. The C programs are less efficient with functions that operate on scalar quantities, such as the trigonometric and the logarithmic functions, where the function returns a single value for a single input value. C programs are more efficient with functions that operate on vectors or arrays of data, such as the Discrete Fourier Transform. The larger the dimension of the array, the more efficient the C program becomes.

These programs have been written following the few simple rules for writing efficient C programs for the DSP32 C Compiler [1]. These rules include, the use of pointers for array references, the use of post modification of pointers, and the use of register variables. The comparison also reflects the current status of the optimizer. Improvements in the optimizer, such as in-line coding of loops, will further reduce the overhead of the C programs. In addition, the compiler also allows the user to program directly in assembly language within the C source code. This is useful when assembly language coding of critical parts of the code is needed to improve the performance.

It was previously mentioned that the compiled and linked C object code can be run on the DSP32 simulator or run in real time with the DSP32-DS Development System. The DSP32 integrated development environment allows the user to switch easily between them using the same command language [4]. The use of the real time development system is preferred whenever the number of instructions executed by the program is "large". As an example, Example 2 requires the execution of about 60,000 instructions with the printf statement removed. This corresponds to a 10ms run time on the Development System, while it takes as long as three minutes using the software simulator on an MSDOS based PC.

TABLE 1 -- Efficiency of C vs. Assembly Program

FUNCTION	EFFICIENCY
Trigonometric & Logarithmic Functions	74 - 87 %
Matrix Multiply -- generic (3x3 matrices)	75 %
Matrix Multiply -- 5x5 matrices	83 %
Matrix Invert -- generic (6x6 matrices)	96 %
FIR Filters -- 19 taps	87 %
FIR Filters -- 99 taps	96 %
IIR Filters -- generic (2 sections)	76 %
IIR Filters -- 4 sections	83 %
FFT -- 16 points	94 %
FFT -- 256 points	99.6 %
Window -- 64 points	90 %
LMS (Example 1) -- 10 taps FIR filter	70 %
LMS (Example 1) -- 64 taps FIR filter	90 %

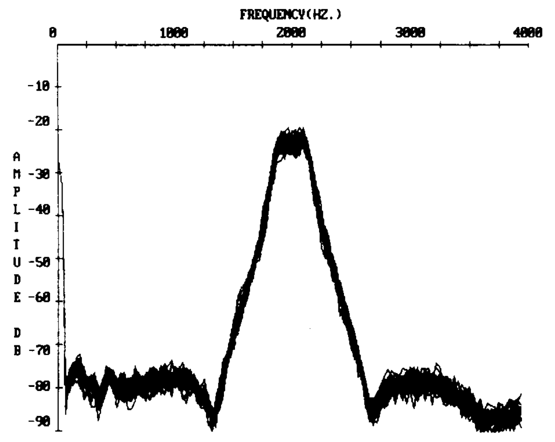

Fig. 3a Filter Output (L = 4)

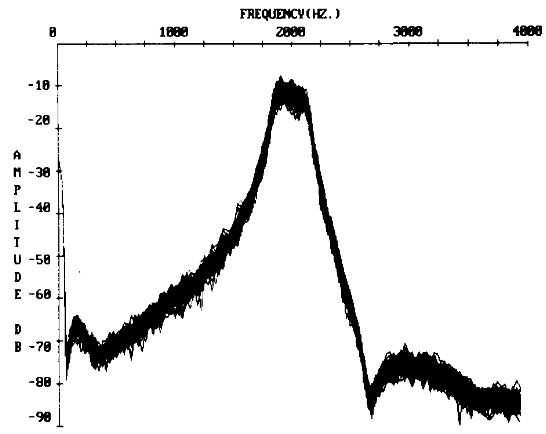

Fig. 3b Third Section Branch Node Output (L = 3)

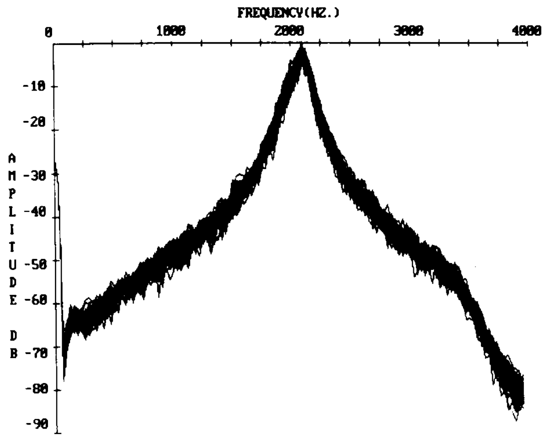

Fig. 3c Second Section Branch Node Output (L = 2)

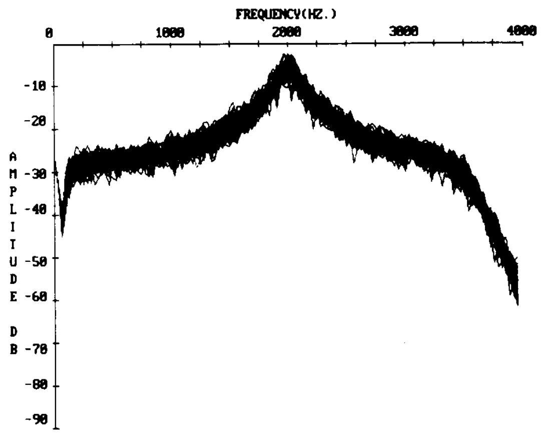

Fig. 3d First Section Branch Node Output (L = 1)

4. CONCLUSION

This paper briefly introduced the C Compiler and the Application Library for the DSP32 Digital Signal Processor. The three examples shown illustrated the ease of writing C programs for some common filtering applications. In most cases, a small real-time penalty is incurred by using C, rather than assembly language coding. Programming in C, however, allows a dramatic improvement in programmer productivity in terms of implementing, testing, and maintaining an application.

REFERENCES

[1] J. Hartung, S. L. Gay, and S. G. Haigh, "A Practical C Language Compiler/Optimizer for Real Time Implementation on a Family of Floating Point DSPs," Proc. IEEE ICASSP, April 11-14, 1988.

[2] *WE*® DSP32-AL Digital Signal Processor, Application Software Library, Reference Manual, April 1987, AT&T Technologies, 555 Union Boulevard, Dept. 50AL203140, Allentown, PA 18103, 1-800-372-2447.

[3] B. Widrow and S. D. Stearns, ADAPTIVE SIGNAL PROCESSING, Prentice-Hall, Inc., (1985).

[4] J. R. Boddie, W. P. Hays, and J. Tow, "The Architecture, Instruction Set and Development Support for the *WE*® DSP32 Digital Signal Processor," Proc. IEEE ICASSP, April 7-11, 1986, pp. 421-424.

[5] Jackson, L. B., DIGITAL FILTERS AND SIGNAL PROCESSING, Kluwer Academic Publishers, (1986).

[6] Filter Design and Analysis Program, Momentum Data Systems, Inc., Version 1.3, 1666 Newport Boulevard, # 115, Costa Mesa, CA 92627 (1987).

[7] J. R. Boddie, et al.,"A Floating Point DSP with Optimizing C Compiler," Proc. IEEE ICASSP, April 11-14, 1988.

APPENDIX -- DESCRIPTION OF FUNCTIONS **fft**, **iird**, and **lms**

FUNCTION NAME: **fft**

The fft function implements an in-place, decimation-in-time, radix 2 fast Fourier transform (FFT) algorithm.

SYNOPSIS:

```
void fft(N, M, fftIO)
int  N, M;
float *fftIO;
```

DESCRIPTION:

The use of each argument is explained below.

N - Length (number of points) of the complex FFT. The maximum length is 1024.

$M = \log_2 N$, $1 \le M \le 10$

fftIO - Pointer to array of floating-point data. The data is in the order: real1, imaginary1, real2, imaginary2, ..., realN, imaginaryN.

FUNCTION NAME: **iird**

Implements an infinite impulse response (IIR) filter corresponding to the direct form II cascade of second order sections with five multiplications per section.

SYNOPSIS:

```
float iird(order, sample, coeff, state)
int order;
float sample, *coeff, *state;
```

DESCRIPTION:

The filter transfer function is the product of second order sections:

$$\prod_{i=1}^{N} \left\{ \frac{n[i,0] + n[i,1] \cdot z^{-1} + n[i,2] \cdot z^{-2}}{1 + d[i,1] \cdot z^{-1} + d[i,2] \cdot z^{-2}} \right\}$$

where i = 1, 2,...N, with N equals the number of sections.

The function returns the floating-point output. The use of each argument is explained below.

order - Number of second-order IIR filter sections.

sample - Floating-point input value.

coeff - Pointer to array of floating-point coefficients. The coefficients are stored in the order: d[1,2], n[1,2], d[1,1], n[1,1], n[1,0], d[2,2], n[2,2], d[2,1], n[2,1], n[2,0], ..., d[n,2], n[n,2], d[n,1], n[n,1], n[n,0].

state - Pointer to array of floating-point state variables. The number of state variables is twice the number of sections.

FUNCTION NAME: **lms**

Implements a real adaptive FIR filter using the least-mean-square (LMS) algorithm without tap leakage.

SYNOPSIS:

```
float lms(length, sample, refer, afirout, lmscon, coeff, sv)
int length;
float sample, refer, *afirout, lmscon, *coeff, *sv;
```

DESCRIPTION:

The function returns the floating-point error value. The use of each argument is explained below.

length - Integer length (order) of the adaptive filter. Length > 1.

sample - Floating-point input value.

refer - Floating-point reference or desired output value.

afirout - Pointer to location storing the floating-point adaptive FIR output.

lmscon - Floating-point adaptive constant 2μ.

coeff - Pointer to array of floating-point adaptive filter coefficients. The filter coefficients are stored in reverse order.

sv - Pointer to array of floating-point static variables. The number of static variables equals the length of the filter.