# Real-time DSP implementation set-up for an acoustic echo canceller

Vervoort, C.P.A.

*Award date:*
1994

# REAL-TIME DSP IMPLEMENTATION SET-UP FOR AN ACOUSTIC ECHO CANCELLER

Master thesis of C.P.A. Vervoort

February 1994


Supervisor : Prof.Dr.Ir. W.M.G. van Bokhoven

Coaches : Ir. G.P.M. Egelmeers
Dr.Ir. P.C.W. Sommen


Period : May 1993 - February 1994


At : Eindhoven University of Technology
Department of Electrical Engineering
Electronic Circuit Design Group (EEB)
Digital Signal Processing

## ABSTRACT

Research of the Digital Signal Processing part of the Electronic Design Group has been concentrated on adaptive filters.

This report describes the set-up to realize a real-time implementation of the Decoupled Partitioned Block Frequency Domain Adaptive Filter (DPBFDAF) on a single digital signal processor (TMS320C30), in order to investigate its real-time performance. The implementation of DPBFDAF will be applied in an acoustic echo canceller configuration. An user interface has been made to change the parameters of the configuration easily.

In the set-up phase of this project efficient routines have been realized. Typical routines, such as input and output communication, Fast Fourier Transform, vector products and convolution in frequency domain have been programmed to minimize the processing time in assembly. This library of basic routines will be used to implement the adaptive filter. These routines can also be called within a high-level programming language. Users of the TMS320C30 digital signal processor can take benefit of this.

CONTENTS

# 1 INTRODUCTION

Research of the Digital Signal Processing part of the Electronic Design Group has been concentrated on adaptive filters. New developments of block frequency domain adaptive filters have been published and presented at conferences. An implementation of such an adaptive filter on a digital signal processor is necessary to investigate its real-time performance.

In this introduction the reason to use adaptive filters and a typical application will be explained. Also, a general description of a digital signal processor and some characteristics of the digital signal processor, used in this project, will be given. At the end the goal of research is presented.

## 1.1 Adaptive Filters

Filters are used to extract some specific information from a signal. This section includes the reason to use an adaptive filter in stead of a fixed filter, a general signal estimation model and a typical application of adaptive filters.

### 1.1.1 Characteristics

An adaptive filter consists basically of two parts:

(1) a filter part, and
(2) an update part.

The filter part produces an output in response to a sequence of input data. The update part provides a mechanism for the adaptive control of an set of adjustable coefficients, used in the filtering part. Both parts work interactively with each other. The features of an adaptive filter, namely the ability to operate satisfactorily in an unknown environment and also track time variations of input signal statistics, make the adaptive filter a powerful device for signal processing and control applications. In comparison to fixed filters, adaptive filters use extra complexity to update the coefficients according to some specific algorithm. However, fixed filters lack the mentioned features of adaptive filters. Both the algorithm for the update and the structure of the adaptive filter use, as much as possible, information to reduce the complexity. In general it is impossible to adapt without any a priori information.

### 1.1.2 Signal Estimation Model

The problem of estimation one signal from another is very important in signal processing. In many applications, the observable signal is a distorted version of the original signal. The signal estimation problem is to recover the original signal from its degraded replica in the best way. With the following notations

| | | | |
|---|---|---|---|
| x[k] | input signal at time k, | e[k] | unknown signal, |
| s[k] | desired signal, | $\hat{e}$[k] | estimated signal, |
| r[k] | residual output signal, | $\tilde{e}$[k] | measurable signal. |

This signal estimation problem can be modelled as follows,

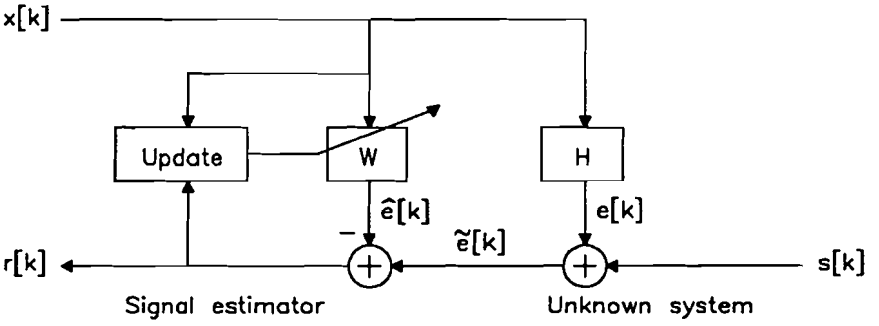


Figure 1-1. Signal estimation model

An unknown system H has to be imitated by an adaptive transversal filter W with N coefficients. The input signal x[k] passed through the unknown system with impulse response h yields a signal e[k]. The desired signal s[k], that is not correlated with x[k], is corrupted by e[k] resulting in a measurable signal $\tilde{e}$[k]. The main task of the adaptive filter is to make an estimate $\hat{e}$[k] of the unknown signal e[k]. The optimal situation is achieved, when the residual output signal r[k] equals the desired signal s[k]. In this case the unknown system is cancelled by the adaptive filter.

A typical application of signal estimation is echo cancellation, which will be discussed next.

### 1.1.3 Acoustic Echo Canceller

This section gives a brief overview of articles [6-9] about the acoustic echo cancelling problem.

Acoustic echo cancellers are indispensable in hands-free telephone systems. Without echo canceller unacceptable roundsinging or ringing effect can occur. The echo is caused by the acoustic coupling between the loudspeaker and the microphone.

The echo reduction, that can be achieved using an echo canceller, depends basically on the length of the acoustic echo path and the characteristics of the room impulse response. In appendix A some room impulse responses can be found of the test environment, including the measurement description. Only small changes in the room results in major changes of the room impulse response. Therefore the echo canceller must be able to adapt rapidly. The tracking capability of adaptive filters may be useful to follow these changes.

The acoustic echo cancellation problem for teleconferencing is a signal estimation problem, as shown in figure 1-2.

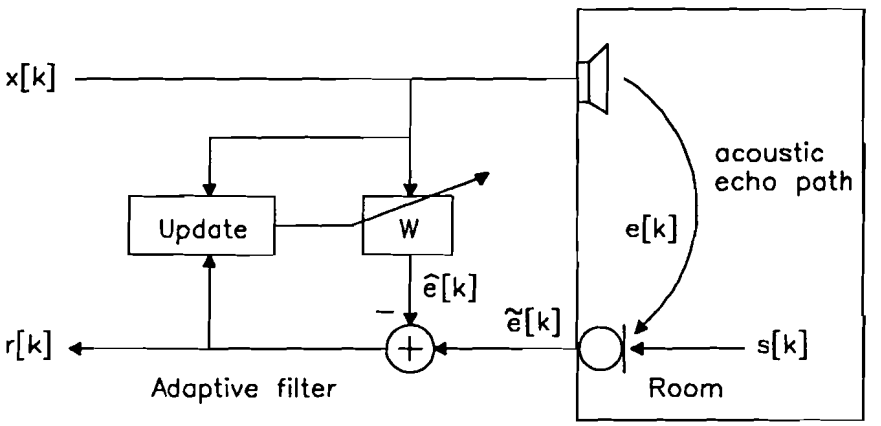


Figure 1-2. Acoustic echo canceller

The signal x[k] coming from the distant teleconference room is convolved with the acoustic response of the local room. This results in an echo signal e[k]. The speech signal s[k] is added to the echo at the microphone. Ideally, the signal r[k], which is sent to the distant room, should be free of echo. Therefore the room acoustic impulse response has to be identified and convolved with x[k] to produce an estimate ê[k] of the echo signal, which is subtracted from the microphone output ẽ[k] to yield the useful signal r[k] to be sent. If this estimation is perfect the exact speech signal can be extracted out of the microphone signal. If not, a residue of echo signal remains.

The performance of echo cancellers is commonly measured by the Echo Return Loss Enhancement (ERLE) [8,9]. Depending on the characteristics of the environment the acoustic echo may be sufficiently strong, such that loss must be introduced into the echo path. The amount of loss introduced, ERLE is defined as:

$$ERLE_{dB} = 10\log_{10}\left(\frac{E\{\tilde{e}^2[k]\}}{E\{r^2[k]\}}\right)$$

where $E\{\tilde{e}^2[k]\}$ and $E\{r^2[k]\}$ are the mathematical expectation of the received echo signal power and the uncancelled, or residual, echo signal power, respectively. During periods with double-talk the adaptation is usually inhibited. Otherwise the adapted echo path would diverge from its previously obtained value.

## 1.2 Digital Signal Processor

Digital signal processing involves the representation, transmission, and manipulation of signals, using numerical techniques and digital processors.

### 1.2.1 General

Digital signal processors (DSPs) are essentially high-speed microprocessors, designed specifically to perform computation intensive digital signal processing algorithms. By taking advantage of the advanced architecture, parallel processing, and dedicated digital signal processing instruction sets.

Programmable DSPs [11,12] began to appear early 1980s. Quite sophisticated signal processing can be done with today's DSPs in the voice band range ($f_s$=8 kHz), but up till now real-time video ($f_s$=5 MHz) is out of the range of a single DSP. Its applications spread a wide range, from traditional radar signal processing till digital audio processing in for example consumer laser disc players. In the last decade digital signal processing has made tremendous progress, in both the theoretical and practical aspects. Digital signal processing is becoming a key tool for multimedia applications, such as combining voice and video information. DSPs are implemented as coprocessors for multimedia processing.

DSPs are also included in disk drives, cellular telephones, modems, radios, medical instruments, automobiles and a number of other products. More and more applications are discovered, where DSPs can provide a better solution than their analog counterparts for reasons of reliability, reproducibility, compactness, efficiency, and flexibility. Different tasks can be performed with the same DSP by changing its program.

### 1.2.2 TMS320C30

In 1988 Texas Instruments introduced the TMS320C30. The third member of the TMS320 family. It is one of the most widely used DSP family for digital signal processing applications.

Some of the key features of the TMS320C30 [13,32]:

- advanced 32-bit microprocessor
- floating-point capability
- 60 nanoseconds cycle execution time
- 33 Million Floating-Point Operations Per Second
- parallel instruction set
- zero overhead loops and single cycle branches
- single cycle type conversion of integer and float
- addressing modes typical to digital signal processing
- modified Harvard architecture
- 64 x 32 bits program cache
- on chip two 1K word RAM blocks, one 4K word ROM block
- on-chip concurrent Direct Memory Access (DMA) controller
- eight extended-precision and auxiliary registers, two index registers

Floating-point DSPs eliminate scaling, one of the major problems in fixed-point DSP and numeric application code. Without floating-point capabilities, programmers must continually scale intermediate results for processing. Floating-point DSPs automatically scale results to hold maximum significance. The floating-point capability permits the handling of numbers of very high dynamic range, without worrying about overflows.

The TMS320C30 utilizes a modified Harvard architecture for speed and flexibility. In a strict Harvard architecture the program and data memories are in two separate spaces, permitting a full overlap of the instruction fetch and execution. The modification of the Harvard architecture allows transfer between program and data spaces. This architecture provides a highly paralleled structure with multiple busses for carrying instructions and data. It allows execution of multiple arithmetic operations in parallel. Program counter, data address generation, arithmetic, logical and multiplication are executed all in parallel to provide arithmetic operation fast enough to operate on some real-time applications. The C30 can not only execute single cycle multiply/accumulates, but can also run loads and stores of data in parallel with arithmetic operations.

The DMA controller is able to perform reads from and writes to any location in the memory map, without interfering with the operation of the CPU. As a consequence, it is possible to interface the TMS320C30 to slow external memories and peripherals (A/Ds, serial ports, timers, etc.) without affecting the computational throughput of the CPU. This controller can be set to bring in or out data to be

processed by the CPU, without ever stopping the CPU.

The instruction cache stores often repeated sections of code. The code may then be fetched from the cache, thus reducing the necessary number of off-chip accesses. This allows for code to be stored off-chip in slower, lower cost memories. Off-chip accesses can only occur once per instruction cycle in stead of twice, and only two busses are available externally instead of four. Also the external busses are freed, thus allowing for their use by the DMA or other devices in the system.

A powerful capability of the TMS320C30 is the block-repeat instruction. A designated repeat counter register RC holds the count of the repetitions, so that loops can be implemented without any overhead. The net effect is, that the repeated code behaves as if it was straight-line code (no penalty for looping), with program size equal to the one in looped-code.

In the C30, there are eight extended-precision registers, R0-R7, that can be used as accumulators, and eight auxiliary registers, AR0-AR7, for addressing and integer arithmetic. For many applications, these registers are sufficient for temporary storage of values and there is no need to use memory locations. Two index registers IR0 and IR1 are used for indexing the contents of the auxiliary registers AR0-AR7.

Additional addressing modes, added to standard addressing modes of microprocessors, are incorporated to increase performance in digital signal processing. For example, shifting of data in a delay line is expensive. An efficient way of shifting data is realized in modulo-mode addressing. The delay line is then implemented as a circular buffer. This can be realized without instructions, except those required to initialize some specific registers. A bit reversed addressing mode is incorporated to reduce the computational load of fast Fourier transforms. These addressing modes do not need extra processing time, they only restrict the data location in memory. The circular buffer must begin on a power of two boundary. Bit reversal can only take place correctly, if the data is located at an address with a number of least significant bits zero (the base two logarithm of the buffer size).

The architecture and the instruction set of the TMS320C30 permit flexible and compact coding of the algorithms in assembly language, while preserving close correspondence to a high-level language implementation. Each instruction requires four cycles from fetch to execution, but an efficient pipeline can execute them in a single cycle. The DSP automatically takes care of any potential conflicts by inserting some delay (one or more wait states). The pipeline should be observed closely, only if code optimization for speed is required.

Texas Instruments offers support, for instance by free DSP code on bulletin boards and in application manuals. Since the TI DSPs are very popular, the amount of public domain software still grows.

### 1.2.3 Programming

The decision of programming language is important for reasons of efficiency, transferability and time of development. The applications of two important program languages, assembly and high-level, are summarized to make a decision easier.

Programming languages available to program the TMS320C30 DSP, are

- assembly language of DSP,
- high-level language C,
- SPOX, an extension to C for signal processing and mathematical intensive applications.

Applications of assembly languages

- small programs
- restrictions to memory capacity
- real-time
- small data processing
- applications including more in- and output, and process control than calculations
- applications of high volume

Applications of high-level language

- big programs
- low volume applications
- applications in which amount of memory is already great
- applications based upon calculations in stead of in- and output or process control
- programs which will be changed a lot

Since the output of a compiler generates an assembly language source file, this source file may be modified by the user. An interlist utility [33], that interlists the original C source statements into the assembly language output of the compiler, enables inspection of assembly code generated for each C statement. Compilers generate no full optimized code, but introduce some overhead. The reason for this is the automatic process, based upon compromises made to allow all kind of problems. Removal of overhead is possible by programming in both languages. However, the generated code contains lot of branches to (standard) subroutines, that are hard to follow. The combination can result in an increase of debug, test and documentation time, compared to a separate implementation.

In assembly it is possible to minimize the processing in time and/or memory requirements. If a routine is time critical, it should be realized in assembly to obtain a better performance. However, programming in assembly demands a detailed knowledge of the processor (instruction set, addressing modes, interrupts, I/O communication, etc.) [32]. One of the major disadvantages is the hardware dependency, it is not transferable to other processors. A high-level program can be executed at another architecture by using a different compiler. Another disadvantage is the increase in development time, compared to an implementation in a high-level language.

Most real-time applications demand a minimal processing delay. A total control of hardware and efficient programming are necessary to realize this. The assembly language program follows very closely the flow of a high-level language program, because of the architecture and the instruction set of the TMS320C30. For these reasons it has been decided to program in assembly. The slower development process has to be accepted. This decision is also influenced by the restricted fast internal memory space. The programming tips and pipeline operations in [34] should be studied to realize (near) optimal assembly code.

## 1.3 Goal of Research

Research of the Digital Signal Processing part of the Electronic Design Group has been concentrated on adaptive filters. New developments of block frequency domain adaptive filters have been published and presented at conferences. An implementation of such an adaptive filter on a DSP is necessary to investigate its real-time performance. The Decoupled Partitioned Block Frequency Domain Adaptive Filter (DPBFDAF) [4] will be implemented to verify experimentally its characteristics in an acoustic echo canceller configuration. This will be realized on a single TMS320C30 digital signal processor of Texas Instruments.

Simulation results of the DPBFDAF were already available. Therefore a simulation program had been written in SPOX, which presents a high-level software interface to the underlying DSP hardware. SPOX improves the productivity of application developers, but consumes more processing time than necessary and in the case of an acoustic echo canceller acceptable.

The implementation should meet the following constraints

- sample frequency of at least 8 kHz,
- minimal processing delay,
- flexible implementation.

A sample frequency of 8 kHz is often used in telephony. However, in high-quality telephony systems it is preferable to sample at 16 kHz. If possible, the processing delay should be smaller than 2 msec. A delay of this order will be acceptable in an

acoustic echo canceller. A flexible implementation can be realized by breaking up the adaptive filter in modules. These modules should be accessible by parameters and have an interface to call the routines in other programming languages.

An user interface will be necessary to change the parameters of the adaptive filter easily. As will be explained, the adaptive filter can be divided into three parts. A so called filter, update and communication part. At first fast routines have to be programmed. These routines will be used in all parts of the adaptive filter. After this the filter part of this adaptive filter can be set up.

## 2 ADAPTIVE FILTER TYPES

The Decoupled Partitioned Block Frequency Domain Adaptive Filter (DPBFDAF) is a sophisticated adaptive filter. Before the DPBFDAF will be discussed with its many degrees of freedom, the development to this type filter will be described. A comprehensive description of these filters is available in [1-4].

A variety of update algorithms has been developed for adaptive filters. Among others, the choice of one algorithm over another is determined by factors of

- Convergence properties,
- Tracking capabilities,
- Misadjustment,
- Processing delay, and
- Complexity.

An important class of adaptive filters update filter coefficients according to the Least Mean Square algorithm. This update algorithm, based on a gradient search technique, minimizes the difference between the residual signal and desired signal (r[k]-s[k]).

### 2.1 Time Domain Adaptive Filters

Adaptive filters in time domain apply filter operations directly onto signals, without any extra transformation.

#### 2.1.1 (Normalized) Least Mean Square

The update algorithms adapt the filter coefficients $w_i[k]$, i=0,1,..,N-1, to remove the correlation between the input signal x[k] and the residual signal $r[k]=\tilde{e}[k]-\hat{e}[k]$. The LMS algorithm is based on the steepest descent algorithm, which uses the gradient

$$\underline{\nabla}[k] = \frac{\partial\, E\{(\tilde{e}[k]-\hat{e}[k])^2\}}{\partial E\{\underline{w}[k]\}} = -2E\{\underline{x}[k]r[k]\}$$

to reduce the correlation $E\{\underline{x}[k]r[k]\}$. In this notation k is the time index and the inter sample distance is assumed to be T seconds. An underlined character denotes a vector. The adaptive weight vector $\underline{w}[k]=(w_{N-1}[k],..,w_1[k],w_0[k])$ and the input signal $\underline{x}[k]=(x[k-N+1],..,x[k-1],x[k])$.

The update algorithm can be written as

$$\underline{w}[k+1] = \underline{w}[k] - \alpha\underline{\nabla}[k]$$

where $\alpha$ is called the adaptation constant.

The LMS update algorithm uses an estimate of the gradient $\underline{\nabla}[k]=-2E\{\underline{x}[k]r[k]\}$ formed by the momentary value $\underline{\nabla}_{LMS}[k]=-2\underline{x}[k]r[k]$. Resulting in the following LMS update algorithm

$$\underline{w}[k+1] = \underline{w}[k] + 2\alpha\underline{x}[k]r[k]$$

This gradient estimate introduces an error, but reduces the complexity compared to the steepest descent algorithm.

The convergence properties of LMS depend on the variance of the input signal [1]. This dependency can be removed by normalizing the input signal x[k] with its own variance $\sigma_x^2=E\{\underline{x}^2[k]\}$. In this way the normalized LMS (NLMS) algorithm is constructed,

$$\underline{w}[k+1] = \underline{w}[k] + \frac{2\alpha}{\sigma_x^2}\underline{x}[k]r[k]$$

This update scheme is depicted in figure 2-1.

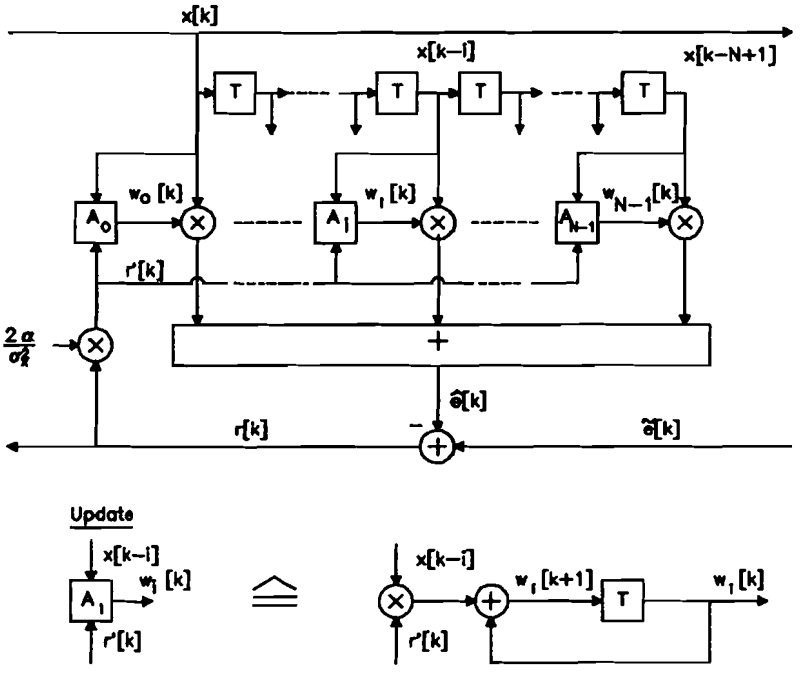


Figure 2-1. Adaptive filter, using NLMS update algorithm.

Convergence properties of the (N)LMS algorithm depend on the statistics of the input signal. Decorrelation of the input signal reduces this dependency. However,

in general decorrelation techniques add extra complexity to the adaptive filter. In time domain an input signal of an adaptive filter can be decorrelated by using an NxN autocorrelation matrix R=E{$\underline{x}[k]\underline{x}^t[k]$}. The superscript t denotes the transpose vector operation.

### 2.1.2 Block NLMS

The Block NLMS (BNLMS) algorithm is performed on block basis. Only one update of all adaptive weights is made every B·T seconds or every block of B samples (B≥1). The BNLMS algorithm

$$\underline{w}[(k+1)B] = \underline{w}[kB] + \frac{2\alpha_B}{B\sigma_x^2} X[kB]\, \underline{r}_B[kB]$$

with residual signal vector $\underline{r}_B[kB]=r[kB-B+1],..,r[kB-1],r[kB])^t$,
the NxB input signal matrix $X[kB] = (\underline{x}_N[kB-B+1],...,\underline{x}_N[kB-1],\underline{x}_N[kB])$, and
for i=0,1,..,N-1 $\underline{x}_N[kB-i]=(x[kB-i-N+1],..,x[kB-i])^t$.

The block processing approach introduces a delay of approximately B·T seconds. An adaptive filter, based on the Block NLMS update algorithm, is depicted in figure 2-2. The box (B-1)(T) collects the B latest samples. The 'down sampler' passes the B latest samples at once, removing the shifting process to create a new vector. The addition box of an input vector to a scalar adds all vector elements.

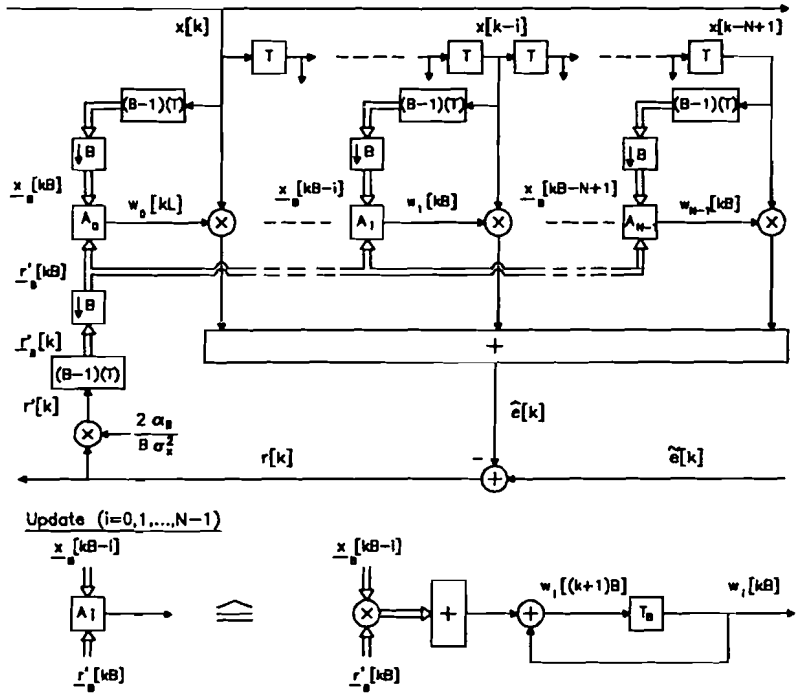


Figure 2-2. Adaptive filter, using BNLMS update algorithm.

The BNLMS algorithm makes a more accurate estimate of the gradient in comparison to the NLMS algorithm, while complexity is in the same order. On the other hand, since updating of the adaptive coefficients is performed less frequently, the BNLMS has a slower rate of convergence in comparison to the NLMS algorithm. Both algorithms exchange misadjustment and rate of convergence.

The two main operations in the BNLMS algorithm are:
(1) a linear convolution to filter the input signal,
(2) a linear correlation to update the adaptive weights.

The linear convolution performs the filtering of the input signal with the adaptive weights. The linear correlation calculates an estimate of the gradient, needed for the update of the adaptive weights. For large filter length N, as needed in echo cancellers, these operations can be carried out efficiently in frequency domain.

## 2.2 Frequency Domain Adaptive Filters

Adaptive filters in frequency domain transform signals to reduce the computational complexity. The coefficients are adjusted independently by an orthogonal transform, known as the Discrete Fourier Transform (DFT). In frequency domain decorrelation of the input signal can be realized with a smaller complexity, compared to the time domain decorrelation methods.

### 2.2.1 FDAF

Convergence properties of an adaptive filter can be made independent of the input signal statistics by decorrelation of the input signal. This decorrelation can be accomplished relatively easy with frequency domain techniques by normalizing the spectrum of each separate frequency component. With this method a reasonable approximation is made of the required time domain decorrelation.

In frequency domain the autocorrelation function $\rho_x[\tau]=E\{x[k]x[k-\tau]\}$ of the input signal x[k] is represented, as the periodic power density spectrum (pds)

$$P_x(e^{j\theta}) = \sum_{\tau=-\infty}^{\infty} \rho_x[\tau]\ e^{-j\theta\tau}$$

If x[k] contains no correlation, the pds is flat. The more correlation the input contains, the less smooth the pds becomes. The input signal x[k] can be decorrelated by normalizing the pds. Splitting the pds into several subbands and dividing every pds subband by its own power is a way to normalize the pds. The whole pds can be normalized, if the number of subbands is large enough to ensure every subband is flat.

Calculation of the adaptive filter output signal ê[k] in frequency domain

$$\hat{e}[k] = \sum_{i=0}^{N-1} x[k-i]w_i[k] = \underline{x}^t[k]\underline{w}[k] = \underline{x}^t[k]F_N F_N^{-1}\underline{w}[k]$$

$$= \frac{1}{N}(F_N\underline{x}[k])^t(F_N^*\underline{w}[k]) = \frac{1}{N}\underline{X}^t[k]\underline{W}[k] = \frac{1}{N}\sum_{i=0}^{N-1} X_i[k]W_i^*[k]$$

with $\underline{X}[k]=F_N\underline{x}[k]=(X_0[k],...,X_{N-1}[k])^t$, $\underline{W}^*[k]=F^*_N\underline{w}[k]=(W^*_0[k],...,W^*_{N-1}[k])^t$,
* is the complex conjugate operation, $F_N$ is the N×N Fourier matrix,
with the $(u,v)^{th}$ element defined by $(F_N)_{u,v}=e^{-j2\pi uv/N}$.

The inverse Fourier transform takes place within the convolution.

The FDAF adaptation algorithm equals

$$\underline{W}^*[k+1] = \underline{W}^*[k] + 2\alpha P^{-1}\underline{X}^*[k]r[k]$$

The power normalization is performed with the matrix $P^{-1}=\text{diag}(1/P_0,..,1/P_{N-1})^t$. The power diagonal matrix $P=\text{diag}\{P_0,..,P_{N-1}\}=\text{diag}\{E\{|X_0[k]^2|\}/N,,E\{|X_{N-1}[k]^2|\}/N\}$, where $E\{|X_i[k]^2|\}=E\{\underline{X}^*_i\underline{X}^t_i\}$.

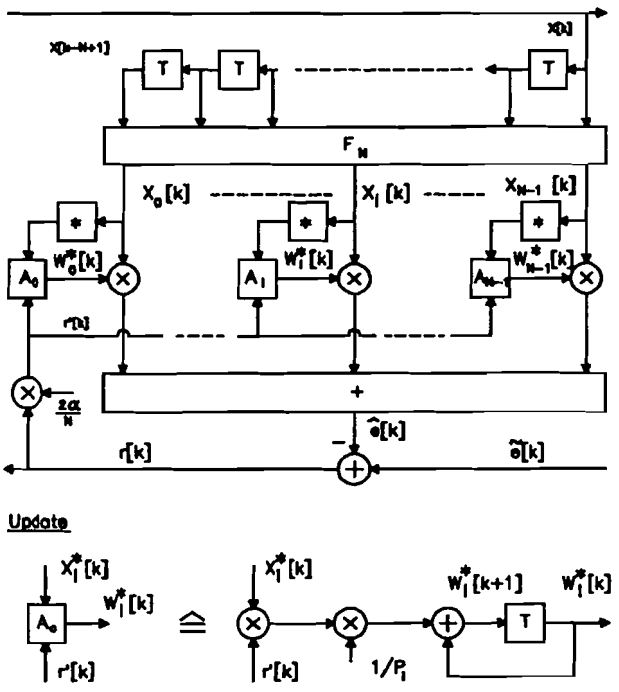


Figure 2-3. Adaptive filter, using FDAF update algorithm.

The FDAF decorrelates the input signal in frequency domain by normalizing the input pds. A disadvantage is the equivalence of filter length and DFT length. The smoother the input pds, the less spectral resolution (= subbands) are needed. Otherwise the convergence behaviour degrades. If the input pds is smooth enough

to be normalized with a shorter DFT length, the filter can be partitioned. Resulting in a reduction of the complexity. The filter partitioning will be discussed, after including block processing to the adaptive filter.

### 2.2.2 Block FDAF

Implementing the BNLMS update equation in frequency domain and performing the spectral power normalization, as in FDAF, leads to the BFDAF. The goal of the BFDAF approach is to tackle two problems simultaneously

(1) Convergence properties are made almost independent of input signal statistics by using power normalization of each separate frequency component.
(2) Complexity is reduced by implementing the convolution and correlation in frequency domain.

For large filters the BNLMS algorithm can be implemented very efficiently in frequency domain, by using Fast Fourier Transforms (FFTs) for the transformation between time- and frequency domain. The length of these transforms is given by the number $M \geq N+B-1$, with N the number of adaptive weights and B the samples processed at once. This block processing approach results in a processing delay of B-1 samples.

The filter part of the BFDAF has to perform a convolution of the signal x[kB] and the coefficients w[kB] of the adaptive filter. Using block processing with block-length $B \geq 1$ this convolution can be described in time domain, as

$$\underline{\hat{e}}[kB] = X^t[kB]\underline{w}[kB]$$

with $\underline{\hat{e}}[kB]=(\hat{e}[kB-B+1], .., \hat{e}[kB])^t$, $\underline{w}[kB]=(w_{N-1}[kB], .., w_0[kB])^t$,
$X^t[kB]=(\underline{x}_B[kB-N+1], .., \underline{x}_B[kB])$ and $\underline{x}_B[kB-i]=(x[kB-i-B+1], .., x[kB-i])^t$.

The update algorithm of BFDAF is the transformed BNLMS algorithm in frequency domain [3]

$$\underline{W}[(k+1)B] = \underline{W}[kB] + \frac{2\alpha}{B} G P^{-1} \underline{X}^*[kB] \otimes \underline{R}'[kB]$$

with $\underline{W}[kB]=F_M(I_N\ 0)^t J_N \underline{w}[kB]$, $J_N$ NxN mirror matrix, $J_N\underline{w}[kB]$ is mirror of $\underline{w}[kB]$
window matrix $G=F_M(I_N\ 0)^t(I_N\ 0)F_M^{-1}$ $\underline{X}^*[kB]=F_M^*\underline{x}_M[kB]$, $P=E\{\underline{X}^*[kB]\underline{X}[kB]\}/M$,
the symbol $\otimes$ represents an elementwise vector product operation,
$\underline{R}'[kB]=F_M(0\ I_B)^t\underline{r}_B[kB]$ transformed and windowed residual vector.

Using circular techniques, the block processing needs often all M input signal samples for the calculation of B output signal samples. The result of such block processing techniques is a vector from which only a part is needed. The window G is used to throw away superfluous elements.

Under ideal circumstances, the convergence properties of BFDAF with a coloured input signal x[k] are almost equal to those of the BNLMS algorithm, if a white noise signal is applied.

### 2.3 Partitioning and Decoupling of BFDAF

In many practical situations, such as an acoustic echo canceller, a large filter length N is needed, while only a small processing delay can be accepted. For large filters the BFDAF approach results in a very large processing delay, if the implementation complexity should remain relative low. Using a small block length B in BFDAF makes it inefficient by increased complexity. This section describes a solution to this problem.

#### 2.3.1 Partitioned BFDAF

Partitioning of the BFDAF into smaller parts leads to the Partitioned BFDAF (PBFDAF) [2,3]. In comparison with the BFDAF, the PBFDAF structure can be realized with smaller FFTs, resulting in a reduced processing delay. If some "a priori" information is available of the input signal spectrum, this information can be used to reduce complexity, since decorrelation is performed with less divisions in the PBFDAF approach. The partition factor should be chosen in relation with the input pds, to preserve good convergence behaviour. In this way the spectral resolution Q and the filter length N are decoupled. These techniques are carried out by partitioning the impulse response, and by that the update algorithms, in separate parts.

The impulse response $\underline{w}[kB]$ of the original adaptive filter is splitted in N/Q separate parts of length Q. The input signal matrix X[kB] can also be partitioned into N/Q parts. With this, the original length B output signal vector $\underline{\hat{e}}_B[kB]$ of the adaptive filter can be rewritten as

$$\underline{\hat{e}}_B[kB] = X^t[kB]\underline{w}[kB] = \sum_{i=0}^{N/Q-1} X_i^t[kB]\underline{w}_i[kB]$$

with $\underline{w}[kB]=(\underline{w}^t_{N/Q-1}[kB], .., \underline{w}^t_0[kB])^t$, $\underline{w}_i[kB]=(w_{(i+1)Q-1}[kB], .., w_{iQ}[kB])^t$,
$X^t[kB] = (\underline{x}^t_{N/Q-1}[kB], .., \underline{x}^t_0[kB])$, and $X^t_i=(\underline{x}_B[kB-iQ-Q+1], .., \underline{x}_B[kB-iQ])$.

The filter estimate, result of convolution between input signal x[kB] and coefficients w[kB], calculated in frequency domain

$$\underline{\hat{e}}[kB] = (0\ I_B)F_M^{-1}\sum_{i=0}^{N/Q-1}(\underline{X}_i[kB]\otimes \underline{W}_i[kB])$$

with $I_B$ the BxB identity matrix, $(0\ I_B)$ the MxB window matrix, discarding the first M-B elements ($M\geq Q+B-1$), $\underline{W}_i[kB]=F_M(I_Q\ 0)^t J_Q \underline{w}_i[kB]$, $\underline{X}_i[kB]=F_M\underline{x}_M[kB-iQ]$.

All N/Q separate adaptive weights vectors are updated using a block update algorithm with segment length M, where the adaptive filter length is Q and processing delay is B·T. An BFDAF structure is used for each separate part.

The BFDAF update equation for each separate adaptive weight vector $\underline{W}_i[kB]$ (i=0,1,..,N/Q-1) is given by

$$\underline{W}_i[(k+1)B] = \underline{W}_i[kB] + \frac{2\alpha}{B}GP_i^{-1}\underline{X}_i^*[kB]\otimes\underline{R}'[kB]$$

with $\underline{X}^*[kB]=F_M^*\underline{x}_M[kB-iQ]$, $G=F_M(I_Q\ 0)^t(I_Q\ 0)F_M^{-1}$ window matrix,
$\underline{R}'[kB]=F_M(0\ I_B)^t\underline{r}_B[kB]$ transformed and windowed residual vector.
$P_i=E\{\underline{X}_i^*[kB]\underline{X}_i[kB]\}/M$, to decorrelate by power normalization.

An adaptive filter can be partitioned in N/Q consecutive separate smaller adaptive filters each having length Q, and each using an BFDAF update algorithm. This partitioning is depicted in figure 2-4. The boxes (B-1)(T) collect the B latest samples in a vector. The down sampler creates an overlap of B latest and M-B older samples, this overlap is necessary in block processing.

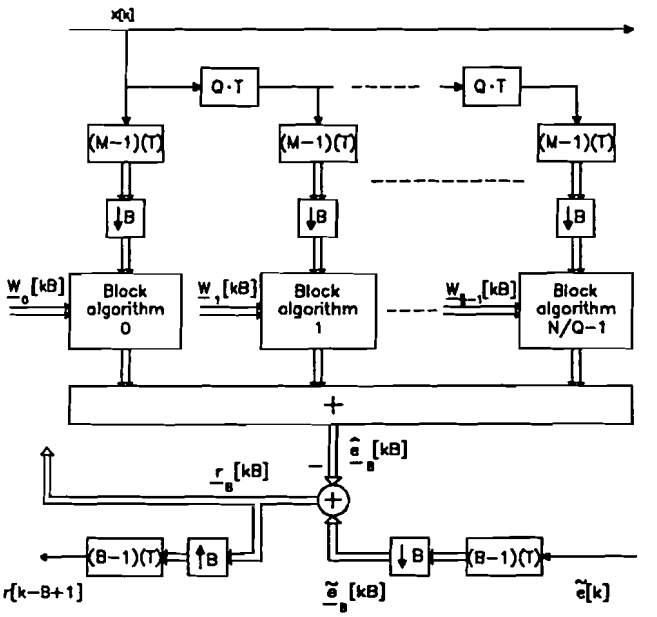


Figure 2-4. Adaptive filter, using Partitioned BFDAF update algorithm.

Dimensions in filter and update part of the mentioned filters are coupled. Convergence properties and processing delay are mutually dependent. Improvement of one will deteriorate the other. This means, a compromise between processing delay and convergence properties must be made.

### 2.3.2 Decoupled PBFDAF

As described in [4], convergence characteristics of update algorithms for adaptive filters are strongly influenced by statistical properties of the input signal. This dependency can be removed by decorrelation of the input signal. In frequency domain this decorrelation is performed by normalizing the power of separate frequency components. A Block Frequency Domain Adaptive Filter (BFDAF) with good convergence properties can be realized efficiently for a large adaptive filter. One of the problems is the large processing delay for a large number N of adaptive weights. To reduce this delay, the adaptive filter can be partitioned, leading to the Partitioned BFDAF (PBFDAF). However, normalization over a reduced number of frequency components can lead to unacceptable convergence behaviour. The Decoupled PBFDAF (DPBFDAF) decouples the decorrelation dimensions and filter partition factor. In this way the processing delay is reduced independent from the decorrelation dimension. The result is an improved convergence behaviour and a smaller computational complexity.

The degrees of freedom of parameters in DPBFDAF, compared to previously mentioned adaptive filters in frequency domain, is shown in the next table, with the following parameter definitions:

Adaptive filter length N

| | filter part | update part |
|---|---|---|
| Block length | B | A |
| Partition length | Q | Z |
| Fourier transform length | $M \geq B+Q-1$ | $L \geq A+Z-1$ |

Table 2-1. Relation of filter parameters compared to DPBFDAF

| Filter type | Block length | Fourier length | Partition length |
|---|---|---|---|
| PBFDAF | A = B | L = M | Z = Q |
| BFDAF | A = B | L = M | Z = Q = N |
| FDAF | A = B = 1 | L = M | Z = Q = N |

A functional description of the Decoupled Partitioned Block Frequency Domain Adaptive Filter [4] will be given to explain its operation.

## 2.4 Functional Description of DPBFDAF

Decoupling of the filter and update part results in a flexible structure. Three very important characteristics of adaptive filters, the convergence properties, processing delay and complexity, can be adjusted independently in this way. The Decoupled PBFDAF can be divided into three parts, a filter, interface and update part. The coupling of filter and update part is established in the interface part.

### 2.4.1 Filter part

The filter part performs the filtering of the input signal with the adaptive weights by a linear convolution in frequency domain. The adaptive filter of length N is partitioned in N/Q consecutive separate smaller adaptive filters. This approach reduces the processing delay.

Input samples, appearing at the sample rate, are collected in a delay line of length M. This delay line of M-1 delays can be represented by a buffer, in which M samples are stored. A total of M data samples is available to filter. However, block processing will take place with a lower number of B samples. Down-sampling with a factor B results in an overlap of M-B older samples. This overlap is necessary to calculate a convolution correctly in frequency domain. A well known technique to convolve an infinite length input sequence with a finite length impulse response is the overlap-save method [15]. The finite length input sequence is represented by x[k], the adaptive weights $w_i[kB]$ is the finite length impulse response. With this method the input signal is split into segments, which are processed separately by applying block processing techniques. In stead of convolving all N coefficients (= length of adaptive filter) at once, the convolution process is performed in N/Q separate parts of Q coefficients by partitioning. The desired signal is a composition of these separate results.

A transform from time to frequency domain is applied to reduce complexity. The (cyclic) convolution is carried out in frequency domain by a (elementwise vector) multiplication of the transformed weight vector and the transformed input signal vector. The resulting vector is transformed back to time domain by an inverse DFT. The estimate $\hat{e}$ of the unknown signal e can be extracted out of this result. Only B out of M samples from this cyclic convolution represent a linear convolution result. Thus M-B samples have to be discarded, resulting in the filter estimate vector $\underline{\hat{e}}[kB]$.

The adaptive filter, applied as acoustic echo canceller, estimates the continuously varying echo of the room. Speech together with echoes present at the room enters the microphone and form the measurable signal $\tilde{e}[k]$. The residual output speech signal, including removal of (estimated) echoes, can be obtained by collecting B samples of the signal $\tilde{e}[k]$ and down sampling this vector by a factor B. The

original sample rate is obtained by up sampling this vector with a factor B and desegmenting it into samples r[k-B+1], that is the output signal of a transposed delay line. The residual signal r[k-B+1] is the B-1 samples delayed output signal, that is sent to an other room. Ideally, it is the speech signal without any echo. In general it is desirable to reduce the echoes to a 'reasonable' level. A certain amount of echo will always be present.

The filter part of adaptive filter is depicted in figure 2-5, in which the boxes q·D represent a cascade of q=Q/B delay operations of one sample (D=B·T seconds). The box $\text{delayline}_M$ is the same as the box (M-1)(T), that is already used in figures previously. The box s/p corresponds to a box ↑B, followed by a box (B-1)(T)

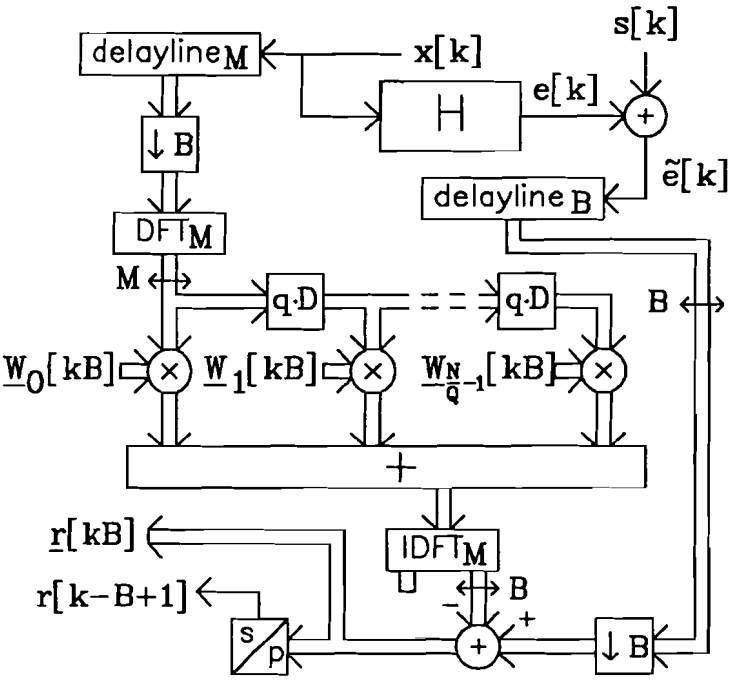


Figure 2-5. Filter part of DPBFDAF

A minimum allowable processing delay constraint can be met by choosing the filter block length B small, this block length represents the number of samples processed at once. Applying this block processing technique results in a processing delay of B-1 samples (= (B-1)·T seconds).

### 2.4.2 Update part

The update part calculates an estimate of the gradient, that is needed for the update of the adaptive weights by a linear correlation in frequency domain. It is also performed on a partitioned filter, except the partition factor N/Z of the update part is smaller than the filter partition factor N/Q.

In general the update part uses parameters with different length at another block rate. As a result the delay line of the filter and update part can not be combined. The block index $\kappa$ is used in stead of k, as the block lengths are in general not equal. All variables equivalent to variables of the filter part have the same name, but with a tilde (~) above it.

In general more samples in the update part will be processed at once, compared to the filter part to obtain a good decorrelation result. Decorrelation of the input signal $\underline{x}[\kappa A]$ in time domain corresponds to normalization of the spectrum in frequency domain. The adjustment of filter coefficients is based on the gradient estimate to reduce the difference between the signal e and its estimate $\hat{e}$. The gradient is estimated by a transformed LMS algorithm. This update algorithm calculates the (cross-) correlation between the input signal and the residual signal.

Correlation can be realized efficiently in frequency domain. An overlap-save method is used, as in the filter part. The correlation process is divided into smaller parts by partitioning. Correlating two signals is an equivalent operation as convolving two signals, except for an extra mirroring of the input signal. This mirroring is carried out in frequency domain by using the conjugate operator.

The adjustment vector is the update value of the filter coefficients. It includes the gradient estimate multiplied with the adaptation constant. This vector is transformed back to time domain, only the first Z samples represent a correct update factor. Because of the ordering of vector elements, it has to be mirrored.

To calculate new adjustment vectors the transformed input signal vector of A samples is mirrored $\underline{X}^*[\kappa A]$ and multiplied with the transformed normalized (decorrelated) residual signal vector $2\alpha\underline{\tilde{P}}^{-1}[\kappa A]\underline{R}[\kappa A]$. A new vector of filter coefficients is derived by adding the adjustment vector to the old filter coefficients vector. Such update of filter coefficients takes place after processing A samples or A·T seconds.

The adaptive filter, configured as echo canceller, should follow variations of the echo path in the room closely. Otherwise the estimate would diverge, these change can cause the (unacceptable) roundsinging effect. The tracking capability of the adaptive filter is basically inside the update part. If the time between updates of filter coefficients is too large, changes can not be followed.

The update part of adaptive filter is depicted in figure 2-6, in which the boxes $z^{-\tilde{D}}$ represent a cascade of z=Z/A delay operations of one sample ($\tilde{D}$=A·T seconds). The label J represent a mirror operation, in which the vector order is reversed.

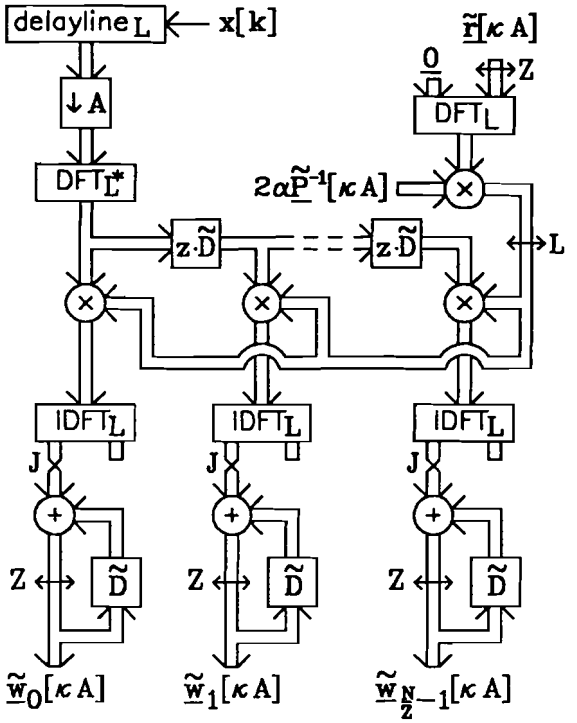


Figure 2-6. Update part of DPBFDAF

For small value of the block length A the filter coefficients are often updated. A large value of the partition length Z results in a large number of frequency bands. Choosing a large value of the partition length Z and a small value of the block length A implies a large complexity. If the block length A is chosen large, a low complexity can be realized.

### 2.4.3 Coupling of both parts

Two interfaces between filter and update part have to be defined for this implementation. One for the adaptive weight vector and the other for the residual signal vector.

The filter part generates a residual output signal vector $\underline{r}$[kB] containing B samples at a rate of $(B \cdot T)^{-1}$. In a delay line A/B such vectors can be collected to create a vector of length A. This residual signal is used to decorrelate the input signal in the update part by power normalization of its spectrum. It is useless to compute

the adaptive weights more often than it is needed, for this reason A≥B. In other words, the number of samples processed in the filter part is smaller than those of the update part. In this way a small processing delay and good convergence behaviour can be obtained. Furthermore for simplicity reasons, it is assumed that A is an integer multiple of B.

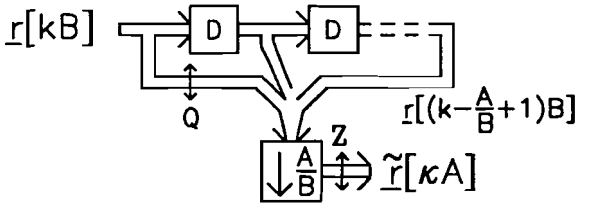


Figure 2-7. Interface of residual signal between filter and update part.

The adaptive weights are fixed to perform A/B filter operations of B samples each. During this period of A·T seconds the filter coefficients are updated. The new coefficients will be used in the following period. The interface, that couples the adaptive weight vector, first merges the separate vectors of the update part and split this vector into parts needed for the filter part. This interface is depicted in the figure 2-8. The hold box performs the updating of filter coefficients after filtering a total number of A samples, and performs the increase of sample rate from 1/(A·T) to 1/(B·T).

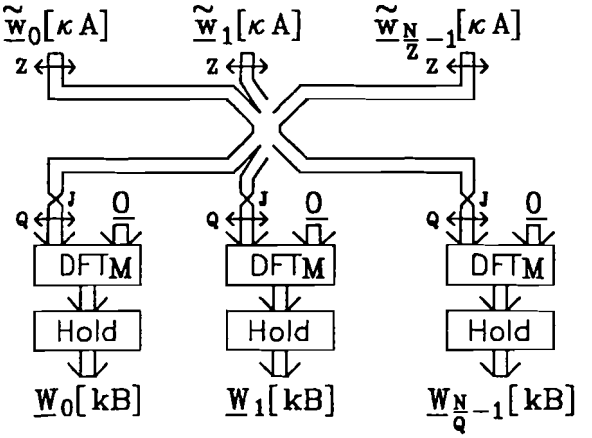


Figure 2-8. Interface of adaptive weight coefficients between update and filter part.

# 3 IMPLEMENTATION ASPECTS OF DPBFDAF

Implementation of an acoustic echo canceller, based on DPBFDAF, can be set up by communication of (assembly) routines. The main processing blocks in the functional diagram of DPBFDAF are the forward and inverse FFT, convolution, correlation, elementwise vector product, input/output and power normalization. These basic routines will be discussed. A study of fast algorithms is presented to obtain a near optimal realization of the discrete Fourier transform on the TMS320C30 DSP.

## 3.1 Fourier Transform

The discrete Fourier transform (DFT) converts information from time to frequency domain. The DFT of a discrete signal x[n], with length N, is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \qquad 0 \le k \le N-1$$

where $W_N = e^{-j(2\pi/N)}$ and $W_N$ is known as the twiddle factor.

The inverse DFT (IDFT) is defined as

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn} \qquad 0 \le n \le N-1$$

The summation in the DFT equation involves N terms and its computation requires (N-1) complex additions. Also, computation of each term in the summation requires one complex multiplication. Therefore, computation of the N-point DFT requires N(N-1) complex additions and $N^2$ complex multiplications, which is a heavy computational load in many applications of the DFT.

The Fast Fourier Transform (FFT) is the generic name for the class of computational efficient algorithms, that implement the DFT. The computation load of order $N^2$ is reduced to $N\log_2 N$. The FFT algorithm derives its efficiency by replacing the computation of one large DFT with that of several smaller DFTs. These FFTs are widely used in the field of digital signal processing, mainly to convolute and correlate signals efficiently.

Since early 1960s FFT algorithms have been developed. Considerable effort has been spent to implement the FFT efficiently on general purpose computers and digital signal processors. Implementation tradeoffs exists, because high-level languages (HLLs) do not provide all of the features required by FFTs. DSPs often include special hardware and software features to improve their efficiency. HLL implementations attempt to reduce the number of multiplications or additions. In DSPs the number of multiplies is not as important, because a multiply takes the

same amount of time as an addition. Limitations on DSPs may be data moves, pipeline restrictions, lack of registers, etc [18,27-30].

First some fast algorithms of the discrete Fourier transform are discussed. After this, methods of computing the real-valued and inverse FFT are presented. The reason for this is to decide, which FFT is most suitable to implement on the TMS320C30 DSP.

### 3.1.1 Radix-2 FFT

The radix-2 FFT divides the input sequence of length-N in two length-(N/2) sequences. This process is repeated, until length-2 sequences remain. It passes $\log_2 N$-1 stages. Two different ways of decomposition exist, called decimation-in-time (DIT) and decimation-in-frequency (DIF).

$$X[k] = \sum_{n=0}^{N/2-1} x[2n]W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x[2n+1]W_{N/2}^{nk} \qquad (radix\text{-}2\ DIT\ FFT)$$

$$X[k] = \sum_{n=0}^{N/2-1} x[n]W_N^{nk} + \sum_{n=N/2}^{N-1} x[n]W_N^{nk} \qquad (radix\text{-}2\ DIF\ FFT)$$

where $0 \leq k \leq N-1$. These length-2 sequences are combined by a butterfly.

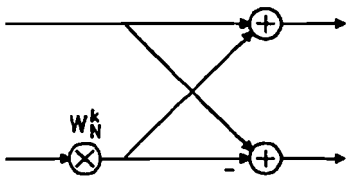


Figure 3-1 Radix-2 butterfly

The radix-2 FFT algorithm offers many freedom, all length (positive) powers of two can be processed. The algorithm can be computed in-place. This means the output data of a butterfly can be written at the input locations, the computation needs no extra memory locations.

### 3.1.2 Radix-4 FFT

The same decomposition, as for radix-2 FFT algorithms, applies to higher radixes as well. The main benefit of using higher radix algorithms is the reduced amount of arithmetic operations, required for the computation. The twiddle factor has special values, when the exponent corresponds to multiples of $\pi/2$. A radix-2

algorithm diagram can be transformed quite straightforwardly into a radix-4 algorithm diagram simply by changing the exponents of the twiddle factors [19].

The radix-4 DIT FFT is decomposed in the following way

$$X[k] = \sum_{n=0}^{N/4-1} x[2n]\, W_{N/4}^{nk} + W_N^k \sum_{n=0}^{N/4-1} x[2n+1]\, W_{N/4}^{nk}$$

$$+ W_N^{2k} \sum_{n=0}^{N/4-1} x[2n+2]\, W_{N/4}^{nk} + W_N^{3k} \sum_{n=0}^{N/4-1} x[2n+3]\, W_{N/4}^{nk}$$

The length-N transform is decomposed in four length-(N/4) transforms.

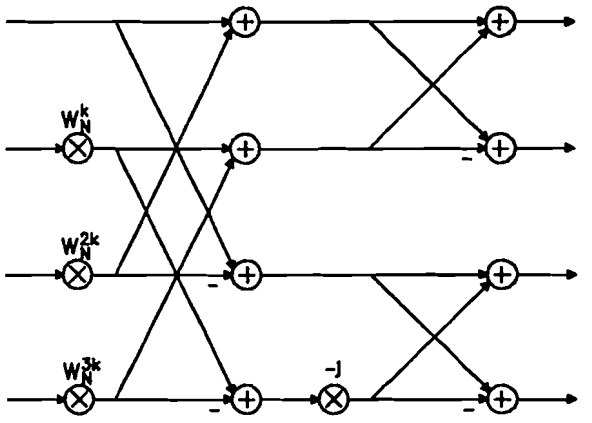


Figure 3-2 Radix-4 butterfly

The radix-2 and radix-4 algorithms are used in many practical applications. This is due to their simple structure, with a constant geometry of butterfly type and the possibility of performing them 'in-place', even if they are more costly in terms of multiplications than some other algorithms.

### 3.1.3 Split-radix FFT

One of the most efficient algorithms for computing the DFT is the split-radix FFT (SRFFT) [20]. The split-radix algorithm applies a radix-2 decomposition to the even indexed samples and a radix-4 decomposition to the odd indexed samples. It is based on the following combination of radix-2 and radix-4 decomposition.

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/4-1} x[4n+1] W_{N/4}^{nk}$$

$$+ W_N^{3k} \sum_{n=0}^{N/4-1} x[4n+3] W_{N/4}^{nk}$$

The first stage of split-radix decomposition replaces a DFT of length-N by one DFT of length-(N/2) and two DFTs of length-(N/4). The length-N DFT is then obtained by successive use of such decompositions up to the last stage, where some radix-2 butterflies are needed. In this way it is possible to obtain an algorithm for a length-$N=2^M$ sequence, using even fewer multiplications and additions than radix-$2^M$ FFTs (M=1,2,3). It can also be performed 'in-place' by repetitive use of a 'butterfly'-type structure given in figure 3-3.

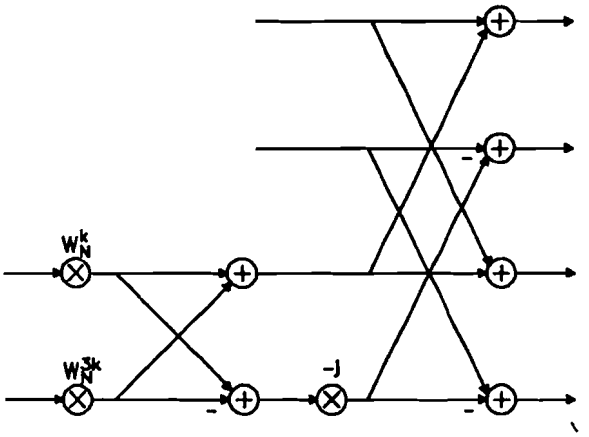


Figure 3-3. Butterfly used in split-radix algorithm.

At first sight it seems a good candidate for fast Fourier transformation on DSPs [21], since it has the lowest number of arithmetic operations for length $N=2^M$. However, the split-radix FFT does not progress stage by stage. To calculate the SRFFT stage by stage it needs an indexing scheme. This is not simple, because of the mixture of radix-2 and radix-4 indexing. For example, the first stage will always calculate N/4 butterflies and associated twiddle factors. The next stage, however, only operates on N/2 data points since the butterflies of the first stage have already calculated half of that stage. Keeping track of which part of a particular stage has already been calculated, is the indexing problem to be solved. This indexing is not straightforward. It is even that complicated, an implementation performs slower than a radix-2 implementation on the TMS320C30 [13].

### 3.1.4 Real-valued FFT

In practice, many signals are real functions of time, whereas the FFT algorithms has been derived for complex signals. It is widely known, that a length-(N/2) complex algorithm can be used to compute the DFT length-N real-valued sequence. It is less well known, that algorithms of greater efficiency can be developed by exploiting symmetries within the complex-valued algorithms. General methods, for constructing fast algorithms to compute the DFT in this way, are reviewed in [22]. The DFT length-N is assumed to be even.

There are three different ways to use an FFT algorithm for complex-valued data to compute the DFT of a real-valued sequence.
The simplest approach is to expand the sequence with zero imaginary part to obtain a complex sequence and then apply the complex FFT (CFFT). This is also the most expensive approach in terms of operational complexity. Exactly the same number of operations and the same amount of storage are required.
A second technique uses the symmetries of the DFT to transform two real-valued sequences simultaneously by computing one CFFT.
The third technique for computing the DFT of a real-valued sequence using a CFFT is useful, when only one real-valued transform is needed. The final stage of a DIT radix-2 FFT algorithm combines two independent transforms of length-(N/2) to compute a length-N transform. If the data are real-valued the two half length DFTs of the even and odd-indexed inputs are also transforms of real-valued data. They can be computed by combining the two sequences into one length-(N/2) complex sequence ($z[n]=x[2n]+jx[2n+1]$).

Symmetries, due to the real-valued nature of the data, should be exploited at every stage of the FFT algorithm to remove redundant operations. If x[n] is real, then X[k] has complex conjugate (or Hermitian) symmetry. It is only necessary to compute X[k] for $0 \le k \le N/2$. If x[n] is real, the real part of its Fourier transform is even symmetric, and the imaginary part is odd symmetric. The Fourier transform coefficients satisfy

$$X[0] \text{ and } X[N/2] \text{ are real, and}$$
$$X[k]=X^*[N-k] \text{ for } 1 \le k \le N/2-1.$$

A reduction in computation is achieved by only computing one of the two output points, that are complex conjugates of each other. To compute the RFFT in-place the redundancies must be utilized to halve the storage requirements compared to the complex FFT. If a real-valued input is applied to a complex algorithm, more and more of the values turn complex, as more and more stages are computed. At first sight this might seem to make in-place computation impossible. However, for each value that turns complex at a given stage, it has a conjugate that will not be computed. Hence the memory location of the conjugated value is never needed, and it can be used to store the imaginary part of the computed complex variable. So, if the real part of a complex value is stored in the $k^{th}$ memory location, the corresponding imaginary part is stored in the $N-k^{th}$ memory location.

A decimation-in-time (DIT) algorithm should be chosen for the computation of a real-valued sequence. The shorter sequences are also real-valued. The symmetry of their transforms can also be exploited at every stage, to reduce operations and storage. Although a decimation-in-frequency (DIF) algorithm yields a spectrum with the same symmetry at the end, the redundancies are not apparent within the algorithm. These redundancies can not easily be exploited to reduce complexity from that required for complex data.

As can be seen in figure 3-4, representing a length-8 DIT and DIF algorithm. After the first stage of DFT there are five real and three complex numbers. So, a total of eleven real numbers have to be stored. In this case, an increase of five locations

compared to the number of input locations. An in-place computation will be impossible, it also scrambles the order of data. If the data of a DIF FFT algorithm is in normal order, the output order is shuffled. In the DIT FFT algorithm the opposite is true.


(a) DIF FFT (b) DIT FFT

Figure 3-4. Propagation of realness in FFT. The symbol * indicates a complex value.

For the IDFT, that transforms the complex conjugate symmetric spectrum into a real-valued time sequence, the DIF algorithm is the best choice. If the even and odd indexed frequency samples are separated into two half-length sequences, symmetry of both sequences is maintained. At every stage of the DIF IDFT algorithm, the redundancies are explicit and easily removable. This corresponds to flow-graph reversal [14] of the DIT algorithm for real-valued series.

These symmetries allow a reduction of the memory and computational requirements by about a factor 2. Since the symmetries imply, that the coefficients X[k], $(N/2+1) \leq k \leq N-1$, and the imaginary parts of X[0] and X[N/2] need not be computed or stored.

### 3.1.5 Other FFT algorithms

FFTs can be implemented efficiently, if the block length is a power of two. However this is not necessary to perform an FFT. Significant time savings can be obtained as long as N is a highly composite number [16]. The selection of FFT algorithm should depend on the requirements of a particular application. In practice the class of $2^M$ FFT algorithms are often preferred, because of the flexibility in block length.

The radix-$2^M$ FFT algorithms are a special class within other FFT algorithms. The Fourier transform can be calculated directly or indirectly by these other algorithms, based on prime factors, Winograd, Hartley or tensor product to mention some types. Both the Hartley transform and tensor product factorization will be discussed, because they are based on alternative ways of computing the Fourier transform.

The Hartley transform is a potential substitute for the Fourier transform in general applications [25]. It works with real in stead of complex numbers.

The discrete Hartley transform (DHT) for a real-valued length N sequence x[n], $0 \le n \le N-1$

$$H[k] = \sum_{n=0}^{N-1} x[n] cas(\frac{2\pi}{N} kn)$$

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} H[k] cas(\frac{2\pi}{N} kn)$$

where cas(x) = cos(x)+sin(x)

The fundamental similarities of the DHT and the DFT [24] make most special techniques equally applicable to fast algorithms for computing either transform.

The DFT can easily be calculated from the DHT [23], as follows

$$Re\ \{DFT(x[n])\} = \frac{1}{2}\{DHT(x[N-n]) + DHT(x[n])\}$$

$$Im\ \{DFT(x[n])\} = \frac{1}{2}\{DHT(x[N-n]) + DHT(x[n])\}$$

where x[N]=x[0].

The inverse Hartley transform has the same form as the forward, except for a scaling factor. The fast Hartley transform is faster than complex FFTs and requires less storage. A fast Hartley transform (FHT) is not faster than a real input FFT based on the Hermitian property. The speed advantage of the FHT is only apparent in the inverse transformation, in which a complex FFT must be used. Use of the FHT, in stead of the real valued FFT, requires only a few more operations. In some situations the greater regularity and the equivalence of the forward and inverse may justify this cost.

An implementation of FHT on the TMS320C30 DSP [13] results in a performance decrease, compared to the radix-2 FFT. The total program code of forward and inverse transform is reduced in this way.

A new way of implementing FFT algorithms on DSPs is based on a tensor product factorization of the DFT [26,27]. The tensor product descriptions have much more degrees of freedom than the standard techniques. This extra freedom is not only

used to describe the required operations as the standard techniques, but also describes the needed data-flow. Traditionally FFT algorithms have been developed trying to minimize the number of arithmetic operations. For many processor architectures the data-flow is equally, if not more important. Hence the tensor product factorization allows to specialize algorithms to the given architecture by closer matching between the algorithm and hardware. A tutorial [27] has been written on this subject.

The tensor product factorization not only controls the break-down into short-length DFTs, but also shows the data-flow between the various blocks. This allows a better scheduling of operations, which gives a better utilization of the DSP pipelining/parallel capabilities. Leading to algorithms with significantly lower overhead than traditional methods. A description of implementing this FFT algorithm in assembly code is given in [26] for the TMS320C30. The presented algorithm can unfortunately not be computed in-place. An in-place computation is crucial for this application on the C30. The reason for this is the restricted fastest internal memory. Probably further research will result in an algorithm, that can be computed in-place. If the algorithm can also take benefit of real-valued input data, it will offer a very efficient implementation.

### 3.1.6 FFT algorithm on TMS320C30

Available registers and internal memory restrict the possible performance of the FFT on the TMS320C30 DSP. Saving of variables in external memory decreases performance. An in-place computation is necessary to compute the FFT fast in restricted internal memory. Both input data and table of twiddle factors, to calculate a 1024-RFFT should fit in internal memory. The algorithm should also take benefit of the real input data, to reduce the computational complexity.

Up till now an algorithm, based on the tensor product factorization, is not known with mentioned properties. Although the total program length of forward and inverse transform is reduced in DHT, the processing time is assumed to be more important. This needed processing time is more, than that to calculate a radix-2 FFT [13]. The split-radix FFT is one of the most efficient algorithms in terms of multiplications, but the indexing mixture of radix-2 and radix-4 can not be realized in an efficient way. An implementation of the SRFFT performs slower than an algorithm of radix-2 or radix-4 [13]. The eight available CPU registers are not enough to calculate the radix-4 faster than the radix-2 [30].

An in-place DIT radix-2 FFT for real input data is the most efficient way to implement an FFT on the TMS320C30. The radix-2 FFT can calculate all lengths, that are powers of 2. Furthermore, in the implementation of an acoustic echo canceller the length of FFT has a constraint of minimum value. A radix-2 FFT may be able to fulfil this constraint by half the size of a radix-4 FFT, resulting in some reduction of processing time.

### 3.2 Basic Routines

In this section the routines and implementation will be described in general terms. Details have been added in appendix C and programs. The computational load of the routines in the adaptive filter (DPBFDAF) is given in appendix D. An user-interface will be necessary to adjust the configuration by initialization of parameters. Interfaces to high-level programming languages will be helpful to speed up execution time of programs.

#### 3.2.1 Setting parameters

The adaptive filter (DPBFDAF) in echo canceller configuration is realized in assembly on a single DSP. All parameters can be initialized by a PC input and check program. This option gives a flexible and friendly interface to the underlying assembly program. In this way the parameters can be changed easily. Before the assembly program is executed, the validity of parameters is verified and some conversions are performed. The boundaries of variables are declared in a separate file and can be modified, without affecting the program code. This initialization program shows also default settings. The generated data file is linked together with the code of the assembly program, representing the echo canceller. A more technical description of the parameters setting program is included in appendix B.

#### 3.2.2 Input-Output

Although digital signal processing provides greater flexibility and noise immunity than its analog alternatives, the world remains analog. Signals have to be converted from analog to digital before processing, and converted back to analog afterwards.

The input and output signals, as well as the characteristics of filters themselves, are all defined at discrete instants of time. A continuous time signal may be represented by a sequence of samples, that is derived by observing the signal at uniformly spaced instants of time. No information will be lost during this conversion process provided, that the sampling theorem is satisfied.

An interrupt service routine is suitable to establish the input and output communication. A dedicated timer of the DSP generates an interrupt at the sample frequency. The program will stop processing, and switch to the interrupt service routine. Two ADCs and two DACs are available, which offer 16-bit precision and a sample rate up to 200 kHz. The digital representation of the input signal can be converted to a float representation by one type conversion instruction. It is also

possible to convert a float into an integer value. Both ADCs are necessary to obtain the input signals x[k] and $\tilde{e}$[k], one DAC performs the conversion of the discrete filter output signal r[k] to an analog signal.

### 3.2.3 (I)FFT

A radix-2 in-place DIT FFT for real-valued data will be implemented, as stated in §3.1.6. The FFT implementation is based on [22].

Symmetries, due to the real-valued nature of the data, can be used at every stage of the FFT algorithm to remove redundant operations. The reduction in computation is achieved by computing only half of the complex values. The other are known by complex conjugate symmetry. It is possible to compute X[0] through X[N/2], but a better choice is to compute X[0] through X[N/4] and X[N/2] through X[3N/4]. The second choice is preferred, because these values are computed by the first (N/4+1) complex butterflies in the standard complex-valued algorithm.

To compute the RFFT in-place the redundancies must be utilized to halve the storage requirement, compared to the corresponding CFFT. If the real part of the $k^{th}$ complex-valued coefficient is placed in the $k^{th}$ location of the real input data, the imaginary part can be stored in the redundant $(N-k)^{th}$ location. The butterfly operations for the RFFT and the CFFT are identical, only the memory locations, of which the data are fetched, are different.

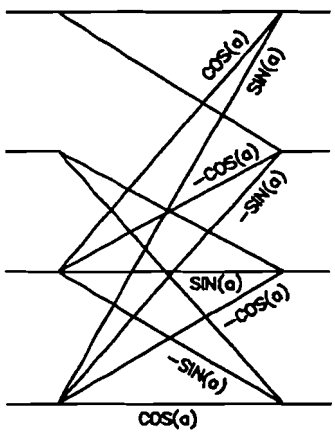


Figure 3-5 Real-valued butterfly.

A special butterfly in the final stage computes X[0] and X[N/2] separately, because they are real and it requires no multiplications. This reduces the computational complexity. A further saving of four additions and a complex multiplication in the last stage is achieved, by noting that the butterfly computing X[N/4] and

X[3N/4] requires no multiplications or additions (only a sign change) for real-valued data. These two complex values are computed separately in a special butterfly. This approach can be applied at every stage by recognizing that the half-length transforms of the even and odd indexed data samples are again DFTs of real-valued data. The same approach can be applied recursively to shorter DFTs to compute the algorithm.

The FFT code, delivered by Texas Instruments (TI), is not very efficient. A more efficient implementation can only be realized, if both the code of FFT and IFFT are reprogrammed.
For example, the parallel instruction set of the TMS320C30 DSP can be utilized better. Two instructions can be executed in parallel, resulting in one execution cycle in stead of two. However, the parallel instruction set is restricted, compared to the single instruction set. Therefore programming in parallel is more complicated. Conflicts in the pipeline operation can affect the total performance, especially in short repeated code. A delay of one clock cycle is inserted, in stead of executing two instructions, if a pipeline conflict is encountered. Sometimes these conflicts can be prevented, without extra processing time. In most cases the solution is to load previously the content of an auxiliary register in an CPU register.
Furthermore, the first two stages of radix-2 DIF FFT can be performed simultaneously, with a special radix-4 butterfly, to enhance execution speed. The special radix-4 butterfly consists of four separate radix-2 butterflies.
Another optimization can be realized by using the repeat block instruction efficient. TIs RFFT calculates the third stage in a loop construction, with a repeat block instruction inside. The loop was programmed as a jump (compare and branch), the repeat block instruction is only executed once. The third stage can be executed completely within one repeat block instruction, that is more efficient. It uses a constant twiddle factor of $0.5\sqrt{2}$, that can be exploited. Especially in loop constructions, it is worthwhile to reduce the number of instructions by rearranging of code.

The performance of the optimized FFT routine for real input data and IFFT routine for complex conjugate symmetry input data (real output data) are given in table 3-1 and 3-2, respectively. The number of clock cycles, needed to compute the same transform, equals the cycle counter of the simulator. One clock cycle is executed in 60 ns. The input data and table of twiddle factors have been stored in internal memory, other variables have been stored in external memory (zero wait states). The mentioned FFT routines of Texas Instruments can be found in appendices C1 and C3 of [13]. The code of the RFFT is also included in [32] as example 12-38. An inverse RFFT program can easily be derived by modifying a DIF inverse CFFT program, in which a complex conjugate symmetric spectrum is transformed to a real-valued sequence. This corresponds to a flow graph reversal of the DIT algorithm for real input data.

Table 3-1. Computational performance RFFT.

| FFT length | RFFT TI | RFFT optimized | profit |
|---|---|---|---|
| 16 | 267 | 172 | 35.5 % |
| 32 | 619 | 392 | 36.6 % |
| 64 | 1382 | 898 | 35.0 % |
| 128 | 3073 | 2079 | 32.3 % |
| 256 | 6796 | 4701 | 30.8 % |
| 512 | 14935 | 10571 | 29.2 % |
| 1024 | 32610 | 23554 | 27.7 % |

Table 3-2. Computational performance RIFFT.

| IFFT length | RIFFT TI | RIFFT optimized | profit |
|---|---|---|---|
| 16 | 295 | 194 | 34.2 % |
| 32 | 671 | 432 | 35.6 % |
| 64 | 1515 | 974 | 35.7 % |
| 128 | 3399 | 2204 | 35.1 % |
| 256 | 7571 | 4970 | 34.1 % |
| 512 | 16735 | 11128 | 33.5 % |
| 1024 | 36715 | 24710 | 32.6 % |

The optimized code computes a (I)FFT up to a length 1024 about 30% faster than the routine of Texas Instruments, the smaller the FFT size the better the result.

### 3.2.4 Convolution

A convolution in time-domain can be calculated in frequency domain by multiplication. In terms of DPBFDAF this convolution can be represented by the following formula, which is schematic shown in figure 3-6.

$$\sum_{i=0}^{N/Q-1} (\underline{X}_i[kB] \otimes \underline{W}_i[kB])$$

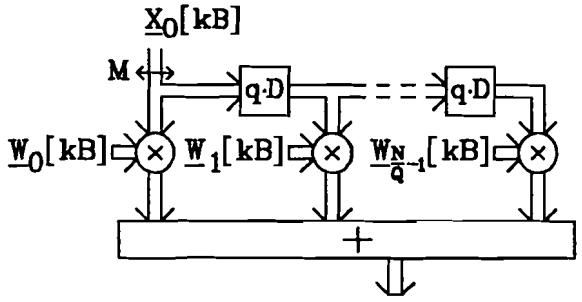


Figure 3-6. Convolution in frequency domain

Rearranging of the direct way (figure 3-6) will save (N/Q-1)·M words of memory space, as shown in figure 3-7. This implies multiplication of two vectors and addition of intermediate results, in stead of calculating all products before addition takes place.

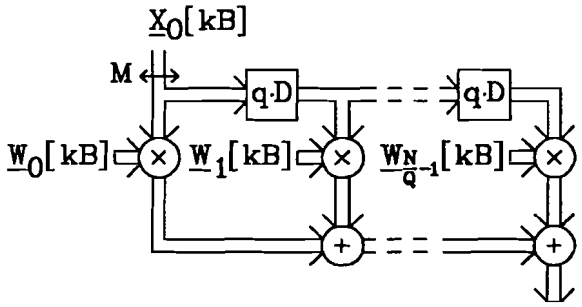


Figure 3-7. Convolution in frequency domain (implementation)

Implementing this convolution in assembly results in a total number of instruction cycles of

$$\frac{N}{Q}(9+4M)+29-M$$

if all vectors can be stored in internal memory. If this is not possible, the number of instructions will be increased.

For example, if a partition factor (N/Q) of 36 and a FFT length M of 64 is chosen, this means it will take 9505 instructions (≈0.0571 ms) to compute the convolution in frequency domain on the TMS320C30. If all 2N/Q+1 vectors of length M can not be stored in internal memory, the computational performance is decreased. Extra wait states are automatically inserted to avoid pipeline conflicts, because external memory accesses do not allow execution of parallel instructions. A better solution will be to calculate the convolution of input and coefficient vectors in parts. During computation DMA can be used to load other (remaining) parts of those vectors in internal memory. The convolution result and its intermediate

results can be stored in M memory locations, both input and coefficient vectors can be loaded partly in the remaining space (2K-M words) of internal memory.

This approach leads also to savings in program space. The reason for this is the better parallel instruction set utilization, compared to the separated implementation of vector multiplication and adding of all results afterwards.

### 3.2.5 Vector product

Both the filter and update of the adaptive filter use elementwise vector products. It is part of the convolution, correlation and normalization in frequency domain. This vector product computation in normalization and convolution is quite straightforward. The computation in correlation needs a closer look.

In the correlation computation the input signal x[kB] must be transformed to frequency domain in a complex conjugate way. Correlating two signals is an equivalent operation as to convolve two signals, except for an extra mirroring. This mirroring is carried out in frequency domain by using the conjugate (*) operator.


(a) direct (b) indirect

Figure 3-8. Implementations of complex conjugated Fourier transform.

Three different ways can be followed to obtain a correct result, a direct, indirect and hidden implementation. The first two implementations are schematic shown in figure 3-8.
The direct implementation of complex conjugated Fourier transform needs rewriting of the Fourier transform. The last stage should be rewritten. An amount of extra program code will be introduced in this way.
The indirect calculation divides the problem in two parts, a Fourier transform followed by complex conjugation of data. Extra program code and processing time will then be offered.
The hidden implementation offers an alternative. Complex conjugating is hidden, because it is handled in the calculation of the elementwise vector product. At the expensive of a little more program code this can be established. Only a few

modifications to the normal elementwise vector product need to be done.

The hidden implementation is the most efficient way to realize the complex conjugation in terms of program code and processing time. For this reason, the hidden computation approach will be implemented.

### 3.2.6 Configuration

The adaptive filter can be realized by calling the developed routines. Starting with the implementation of the filter part, the update and coupling parts will be realized later, by another person in the second phase of this project. An interrupt service routine takes care for the input and output of signals at the sample rate. Samples of signals must be stored to process and buffer. During block processing to filter new samples will arrive, that must be stored to process, if a total block of samples is available. The filter coefficients are adjusted in the update part. These coefficients are fixed, for the time the update and coupling parts have not been implemented. The filter part has not been completed. Up till now it processes fixed data of restricted size. All basic routines of the filter part are called in correct order. The only purpose is to demonstrate the functioning of routines.

### 3.2.7 Interface to other languages

It will be useful to speed up the computation process by calling the assembly routines within high-level programming languages. Therefore an interface to the programming languages C and SPOX (Signal Processing Operating eXecutive) has to be defined.

It would be easy, if one interface could fulfil this. However some differences between C and SPOX make this option difficult or even impossible. SPOX features special digital signal processing functions, extended to the standard C programming language. Communication between PC and DSP is automatically taken care for. The PC is used as input and output monitor of the program running on the DSP. In C this communication between PC and DSP should be programmed. Therefore it is relative easy to call assembly routines within a SPOX program. No knowledge of assembly is necessary, the interface in assembly defines the order and type of arguments to be passed. An example of interfacing SPOX and assembly programs is added in appendix E. Separate modules of assembled code can be linked with a compiled SPOX program and the execution of this code is the same, as any other SPOX programs.

The assembly language function is called in the same way, as every function in a SPOX (or C) program. The only difference is the external declaration of the

function. The assembly function needs some kind of shell or interface to communicate with the SPOX program. The environment may not be disrupted by the context switch to the assembly program. To accomplish this the environment is saved on the stack during the execution of the assembly program. For communication purpose memory space must be reserved for arguments of the function and some global definitions are necessary to link both programs into executable code. A more detailed description, to interface without violation of conventions, can be found in appendix E and chapter 4 of [33].

A TMS320C30 High-Level Language Interface Library is available. These library routines are used to download TMS320C30 object code to the DSP board, to start execution of that code and to pass data between the program running on the TMS320C30 and on the PC. By using this library it is possible to create a communication environment, like that offered in SPOX. Assembly programmers can run and control programs in this way. Programmers, unfamiliar with assembly, should be confronted with a defined communication interface. To avoid errors a communication library should be offered, if assembly routines or even complete filters are accessed within C. Appendix E contains an example, it is the C equivalent of the SPOX example. A detailed description of each library function is given in [35].

The library functions transfer data directly to and from memory space of the DSP. By using these functions more knowledge of the assembly implementation is asked. The assembly routine, developed in the debugger or simulator, need some little changes. Before execution it should test, if the data is ready. Afterwards it should indicate, that it is ready. The assembly routine should be assembled into executable code. The C program is compiled in the way, it would be normal PC code, although the processing takes partly or almost entire place at the DSP. In the executable (object) code of the assembly routine variables are put at a linker defined place. The type, length and memory address of variables must be defined explicitly to guarantee a correct operation. An alternative could be a well defined interface to pass variables between PC and DSP in some special reserved memory space.

At this moment it seems likely, that this type of interfacing will only be used by programmers familiar with assembly. In general SPOX will be used, which already has a well defined interface to assembly. By using SPOX programmers do not need any hardware implementation knowledge.

# 4 CONCLUSION

During the set-up basic routines have been developed to realize a real-time implementation of the Decoupled Partitioned Block Frequency Domain Adaptive Filter on the TMS320C30 Digital Signal Processor. The implementation will be applied as an acoustic echo canceller. This library of basic routines will be used to implement the adaptive filter.

Typical routines, such as input and output communication, Fast Fourier Transform, elementwise vector product and convolution in frequency domain have been programmed efficiently. The code has been written to minimize the processing time in the assembly language of the DSP. The assembly routines can also be called in a high-level programming language, such as C and SPOX. Therefore the (near) optimal code is not only suitable for assembly programmers. Users of the TMS320C30 DSP can take benefit of this too.

Software tools are essential to support the development of an efficient implementation. In appendix F features of two tools have been compared to develop software.

An user interface has been made to change the filter parameters easily. The filter part has been set-up, but could not be completed. In the second phase of the project, to implement an acoustic echo canceller on a single DSP in real-time, this could be established. Furthermore, the power normalization routine must be programmed and communication of routines to obtain the update part and interfaces between filter and update part. Recommendations are described in the following chapter. The end of the first phase and start of the second phase had an overlap of three months. In this way experience and implementation knowledge, additional to this report, could be carried over. This transfer of project will reduce the initiation period of the second phase.

# 5 RECOMMENDATIONS

The highest priority in the next phase will be the availability of an application of the adaptive filter (DPBFDAF). A most efficient implementation will be less important. However, an inefficient implementation will not be possible. Restrictions of hardware are too strict. Later optimization of this implementation may be useful.

The update part of DPBFDAF can be composed by using the library of assembly routines for the TMS320C30, developed in the first phase of the project. Every module of the update part is available, except the power normalization. Also the interface between filter and update part has to be made, including the transformation of the residual and filter coefficients vector. Extra restrictions of parameters may be necessary to include in the parameter setting program. Additional constants, for example A/B and in the power estimation equation [4], could also be calculated in this setting program.

The computation of update and interface has to be accomplished without interrupting the filtering. In general, the processing of the filter part use a smaller block length (= number of samples) than the update part. Therefore the update of filter coefficients is less frequently computed, once every A samples. The time left after filtering B samples (and processing in and out coming samples), will be used to compute the other parts. This time remains A/B times. It corresponds to a number of instructions of

$$(\text{sample frequency})^{-1}(60.10^{-9})(\text{filter block length}) \; - \; (\text{number of instructions needed to compute filter part and input/output of samples})$$

A flexible approach, to compute the adaptive filter without interrupting the filtering and input and output of samples, is to save the context of all registers and by this the point of execution (program counter). In this way the routines do not have to be partitioned, but processing partitions automatically. Instructions in an interrupt service routine are able to redirect the program flow. If a new block of samples has arrived, this must be processed directly in the filter part. The filtering may not be disrupted by the update of coefficients and interfacing. For this reason, the filter part has a higher priority than the updating and interfacing. An interrupt service routine takes care for the input and output of samples at the sample rate.

The transformation of the residual signal can be implemented quite easy. The output signal of the filter part is passed to the DAC, and must also be stored for use in the update part. If the residual signal of each filter processing, including B samples, is appended to previous stored values, this transformation can be done automatically. This type of adaptive filter calculates a new update of filter coefficients after processing a total of A samples. A buffer of output samples must be twice the length A of one vector. At the start of a new conversion of A samples the pointers to the current buffer of A samples and the new buffer must be inter-

changed.

The new filter coefficient vector must be rearranged and transformed from time to frequency domain by the interface routine. The hold function in the schematic representation of the transformed filter vector (figure 2-8) can be implemented in a similar way as proposed for the transformation of the residual signal. Storage of both current and new filter coefficient vector will be needed.

The filter part must be optimized by use of internal memory. This means moving, loading and storing, of data from internal into external memory and vice versa. The current test configuration makes no optimal use of the internal memory. In most cases external memory is reserved, in this way no limits exists by boundaries of memory capacity. The available external memory is in the order of 192K words, in which a word corresponds to 32 bits or 4 bytes. Both input and filter coefficient vectors, $\underline{x}_i$[kB] and $\underline{w}_i$[kB] for i=0 to N/Q-1, must be stored in internal memory, to compute its convolution as fast as possible. However both vectors can only be stored partly, because this type of memory is restricted. For reasons of optimal performance downloading of these vectors must be accomplished by using DMA during computation.

This implementation of adaptive filter must be configured as an acoustic echo canceller. Practical problems will probably appear, if the project enters the phase of measuring. The implementation must be extensively tested to ensure a correct functioning, and of course well documented. Furthermore the accuracy of floating-point signal processing must be looked at. This type of processing introduces errors as most computational methods on computers. Be sure its effects may be ignored, otherwise measures should be taken to prevent this. For this purpose [31] should be studied, also [13] and [32] discuss the subject of floating-point. Maybe a study in literature is necessary to have a better insight in this problem.

Research guideline in second phase

- Optimal computation of filter part by adding DMA
- Extra check number of routine instructions
- Power normalization routine
- Set up of update part, out of assembly library routine
- Coupling of separate parts, way of data storage and rearranging of filter coefficient vector
- Adjustment of interrupt service routine to filter without interruption
- Definition of parameter restrictions, tested in set parameter program
- Recommendations of further improvement
- Extensively testing
- Documentation during development
- Optimization of library assembly routines.
  The convolution routine could be optimized by implementing of [5]. This new method offers considerably reduction of the computational complexity. Optimization of other routines will lead to less reduction of computational complexity.

**ABBREVIATIONS**

| | | |
|---|---|---|
| ADC | : | Analog-to-Digital Converter |
| BFDAF | : | Block Frequency Domain Adaptive Filter |
| BLMS | : | Block Least Mean Square |
| CFFT | : | Complex Fast Fourier Transform |
| DAC | : | Digital-to-Analog Converter |
| DAT | : | Digital Audio Tape |
| DFT | : | Discrete Fourier Transform |
| DMA | : | Direct Memory Access |
| DPBFDAF | : | Decoupled Partitioned Block Frequency Domain Adaptive Filter |
| DSP | : | Digital Signal Processor |
| FDAF | : | Frequency Domain Adaptive Filter |
| FFT | : | Fast Fourier Transform |
| FHT | : | Fast Hartley Transform |
| HLL | : | High-Level Language |
| IFFT | : | Inverse Fast Fourier Transform |
| LMS | : | Least Mean Square |
| NLMS | : | Normalized Least Mean Square |
| PBFDAF | : | Partitioned Block Frequency Domain Adaptive Filter |
| PC | : | Personal Computer |
| RFFT | : | Real-valued Fast Fourier Transform |
| SPOX | : | Signal Processing Operating eXecutive |
| SRFFT | : | Split-Radix Fast Fourier Transform |
| TI | : | Texas Instruments |

## REFERENCES

### ADAPTIVE FILTERS

#### GENERAL


[ 1] Haykin, S.
ADAPTIVE FILTER THEORY.
Second Edition.
Englewood Cliffs: Prentice-Hall, 1991.
ISBN 0-13-005513-1


#### (D)PBFDAF


[ 2] Sommen, P.C.W.
ON THE CONVERGENCE PROPERTIES OF A PARTITIONED BLOCK FREQUENCY DOMAIN ADAPTIVE FILTER (PBFDAF).
In: Signal Processing V: Theories and Applications.
Ed. by L.Torres, E. Masgrau and M.A. Lagunas.
Amsterdam: Elsevier, 1990.

[ 3] Sommen, P.C.W.
ADAPTIVE FILTERING METHODS: ON METHODS TO USE A PRIORI INFORMATION IN ORDER TO REDUCE COMPLEXITY WHILE MAINTAINING CONVERGENCE PROPERTIES.
Eindhoven: Doctoral Disseration, 1992.
ISBN 90-9005143-0

[ 4] Egelmeers, G.P.M.
DECOUPLING OF PARTITION FACTORS IN PARTITIONED BLOCK FDAF.
In: Proceedings of the 11th European Conference on Circuit Theory and Design - ECCTD '93, Davos, 30 August - 3 September 1993.
Amsterdam: Elsevier, 1993.
P. 323-328.

[ 5] Egelmeers, G.P.M. and P.C.W. Sommen
A NEW METHOD FOR EFFICIENT CONVOLUTION IN FREQUENCY DOMAIN BY NON-UNIFORM PARTITIONING
To appear in: Proceedings EUSIPCO-94, Edinburgh, September 1994.

[ 6] Park, S. and G. Hillman.
ON ACOUSTIC ECHO CANCELLATION IMPLEMENTATION WITH MULTIPLE CASCADABLE ADAPTIVE FIR FILTER CHIPS.
In: Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, Glasgow, 23-26 May 1989.
New York: IEEE, 1989.
P. 952-955.

[ 7] Reusens, P. and P. Reynders, P. Guebels.
ECHO CANCELLERS FOR TELEPHONE APPLICATIONS BASED ON PROGRAMMABLE DIGITAL SIGNAL PROCESSORS.
In: Signal Processing V: Theories and Applications.
Ed. by L. Torres, E. Masgrau and M.A. Lagunas.
Amsterdam: Elsevier, 1990.
P. 1475-1478.

[ 8] Borralo, J.M.P. and M.G. Otero
ON THE IMPLEMENTATION OF A PARTITIONED BLOCK FREQUENCY DOMAIN ADAPTIVE FILTER (PBFDAF) FOR LONG ACOUSTIC ECHO CANCELLATION
In: Signal Processing 27,
Amsterdam: Elsevier, 1992
P. 301-315.

[ 9] Jensen, S.H.
ACOUSTIC ECHO CANCELLER FOR HANDS-FREE MOBILE RADIOTELEPHONY.
In: Signal Processing VI: Theories and Applications.
Ed. by J. Vandewalle, R. Boite, M. Moonen and A. Oosterlinck.
Amsterdam: Elsevier, 1992.
P. 1629-1632.

[10] Rife, D.D. and J. Vanderkooy
TRANSFER-FUNCTION MEASUREMENT WITH MAXIMUM-LENGTH SEQUENCES
In: Journal of the Audio Engineering Society, Vol.37, No.6, June 1989
P. 419-443.

DIGITAL SIGNAL PROCESSOR TMS320C30

[11] Lee, E.A.
PROGRAMMABLE DSP ARCHITECTURES: PART I.
In: IEEE ASSP Magazine, October 1988.
P. 4-19.

[12] Lee, E.A.
PROGRAMMABLE DSP ARCHITECTURES: PART II.
In: IEEE ASSP Magazine, January 1989.
P. 4-14.

[13] DIGITAL SIGNAL PROCESSING
APPLICATIONS WITH THE TMS320 FAMILY
Volume 3.
Edited by P. Papamichalis.
Englewood Cliffs: Prentice Hall, 1990.
ISBN 0-13-212960-4

DIGITAL SIGNAL PROCESSING

[14] Rabiner, L.R. and B. Gold.
THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING.
Englewood Cliffs: Prentice-Hall, 1975.
ISBN 0-13-914101-4

[15] SIGNAL PROCESSING HANDBOOK.
Ed. by C.H. Chen.
New York: Marcel-Dekker, 1988.
ISBN 0-8247-7956-8

FAST FOURIER TRANSFORM

GENERAL

[16] Brigham, E.O.
THE FAST FOURIER TRANSFORM.
Englewood Cliffs: Prentice Hall, 1974.
ISBN 0-13-307496-X

[17] Nussbaumer, H.J.
FAST FOURIER TRANSFORM AND CONVOLUTION ALGORITHMS.
Berlin: Springer-Verlag, 1981.
ISBN 3-540-11825-X

[18] Brigham, E.O.
THE FAST FOURIER TRANSFORM AND ITS APPLICATION.
Englewood Cliffs: Prentice Hall, 1988.
ISBN 0-13-307547-8

RADIX-4

[19] Allen, G.H.
PROGRAMMING AN EFFICIENT RADIX-FOUR FFT ALGORITHM.
In: Signal Processing, Vol.6, No.4, August 1984.
P. 325-329.

SPLIT-RADIX

[20] Sorensen, H.V. and M.T. Heideman, C.S. Burrus.
ON COMPUTING THE SPLIT-RADIX FFT.
In: IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol.ASSP-34, No.1, February 1986.
P. 152-156.

[21] Duhamel, P.
IMPLEMENTATION OF SPLIT-RADIX FFT ALGORITHMS FOR COMPLEX, REAL, AND REAL-SYMMETRIC DATA.
In: IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol.ASSP-34, No.2, April 1986.
P. 285-295.

REAL VALUED

[22] H.V. Sorensen and D.L. Jones, M.T. Heideman, C.S. Burrus.
REAL-VALUED FAST FOURIER TRANSFORM ALGORITHMS.
In: IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol.ASSP-35, No.6, June 1987.
P. 849-863.

HARTLEY TRANSFORM

[23] Sorensen, H.V. and D.L. Jones, C.S. Burrus, M.T. Heideman
ON COMPUTING THE DISCRETE HARTLEY TRANSFORM
In: IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol.33, No.4, October 1985.
P. 1231-1238.

[24] Buneman, O.
CONVERSION OF FFTs TO FAST HARTLEY TRANSFORMS
In: SIAM Journal on Scientific and Statistical Computation, Vol.7, No.2, April 1986.
Philadelphia: Society of Scientific and Applied Mathematics,1986.
P. 624-638.

[25] Le-Ngoc, Tho and M.T. Vo
IMPLEMENTATION AND PERFORMANCE OF THE FAST HARTLEY TRANSFORM
In: IEEE Micro, October 1989.
P.20-27

TENSOR (Kronecker) PRODUCT DECOMPOSITION

[26] Johnson, J.R. and R.W. Johnson, D. Rodriquez, R. Tolimieri.
A METHODOLOGY FOR DESIGNING, MODIFYING AND IMPLEMENTING FOURIER TRANSFORM ALGORITHMS ON VARIOUS ARCHITECTURES.
In: Circuits, System Signal Process, Vol.9, No.4, 1990.
P. 449-500.

[27] Sorensen, H.V. and C.A. Katz, C.S. Burrus.
EFFICIENT FFT ALGORITHMS FOR DSP PROCESSORS USING TENSOR PRODUCT DECOMPOSITIONS.
In: Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, Albuquerque, 3-6 April 1990.
New York: IEEE, 1990.
P. 1507-1510.

DIGITAL SIGNAL PROCESSOR (TMS320C30)

[28] Li, Z. and H.V. Sorensen, C.S. Burrus.
FFT AND CONVOLUTION ALGORITHMS ON DSP MICROPROCESSORS.
In: Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, Tokyo, 7-11 April 1986.
New York: IEEE, 1986
P. 289-292.

[29] Meyer, R. and K. Schwarz.
FFT IMPLEMENTATION ON DSP-CHIPS - THEORY AND PRACTICE.
In: Proceedings IEEE International Conference on
Acoustics, Speech, and Signal Processing,
Albuquerque, 3-6 April 1990.
New York: IEEE, 1990
P. 1503-1506.

[30] Papamichalis, P.E.
FFT IMPLEMENTATION ON THE TMS320C30.
In: Proceedings IEEE International Conference on
Acoustics, Speech, and Signal Processing,
New York, 11-14 April 1988.
New York: IEEE, 1988
P. 1399-1402.

FLOATING-POINT ERRORS

[31] Lacroix, A.
FLOATING-POINT SIGNAL PROCESSING-ARITHMETIC, ROUNDOFF-NOISE AND LIMIT CYCLES
In: IEEE International Symposium on Circuits and Systems,
7-9 June 1988, Espoo.
New York: IEEE, 1988
P. 2023-2030.

PROGRAMMING

[32] THIRD GENERATION TMS320.
User's guide.
Digital Signal Processor Products.
Dallas: Texas Instruments, August 1988.

[33] TMS320C30 C COMPILER.
Reference guide.
Digital Signal Processor Products.
Dallas: Texas Instruments, December 1988.

[34] TMS320C30 ASSEMBLY LANGUAGE TOOLS.
User's guide.
Digital Signal Processor Products.
Houston: Texas Instruments, 1988.

[35] TMS320C30 PC SYSTEM BOARD.
User guide & Technical reference, Version 1.01.
Loughborough: Loughborough Sound Images, September, 1990.

# APPENDIX A MEASUREMENT OF REVERBERATION TIME IN TEST ROOM

The signal estimation model, as described in chapter 1, is used to model the acoustic echo canceller. The adaptive filter W needs to imitate the unknown system H some a priori information. It is assumed, that the unknown system can be modelled as a transversal filter including N coefficients. The unknown system is the acoustic coupling channel between loudspeaker and microphone in a room. Ideally, cancelling acoustic echoes requires a delay line which provides a delay equal to the impulse response time of the echo. The impulse response between loudspeaker and microphone characterizes the echo signal in the room.

The reverberation time of the test room has been measured to estimate the number of coefficients in the model of the unknown system H. This number equals the length of delay line.

The most direct approach in measuring the impulse response of a linear system is to apply an impulsive excitation to the system and observe the response. An impulsive acoustical excitation can be produced by pistol shots or exploding balloons. However it is difficult to assure, that the energy is equally distributed over all frequencies of interest. Because the duration of the impulse is very short, by definition, it is difficult to deliver enough energy to the system to overcome the noise that is present.

An impulse function is simulated by puncturing a balloon. This causes a detonating signal, followed with reverberations. The sounds are recorded on a DAT recorder, which samples the incoming signal at a frequency of 48 kHz. After recording a number of impulse responses, these signals are directed to a DSP, which is only used to store the samples in internal memory and finally to disk. The sources of noise, such as lighting of TL, computers and sounds coming from the outside, are being limited as much as possible. Advantages of the above mentioned measurement, in stead of the direct way without a DAT recorder, are the absence of the noise caused by the computer and an increased time to measure.

Better ways of measuring a room impulse response exists, but this experiment was set up to verify widely used parameters and to obtain (real) data of an impulse response. A more professional measurement of the transfer function is based on Maximum-length sequences [11]. This approach is said to be preferable for many applications over other techniques, including dual-channel FFT, periodic pulse testing and time-delay spectrometry.

Some impulse responses of the echo path in a 3x3x4 $m^3$ test room are shown in figure A-1. The length of impulse responses have been truncated at 2000 samples. Assuming a sample frequency of 8 kHz the reverberation time equals 250 msec. This means it will take 2000 taps processing, within a period of 125 µsec, to cancel echoes. After 2000 samples the echo reaches a level, that will not be audible. The

remaining echoes are neglected in the implementation of echo canceller. These responses are measured in almost the same situation, not disturbing the given environment. Even in this case major differences occur.

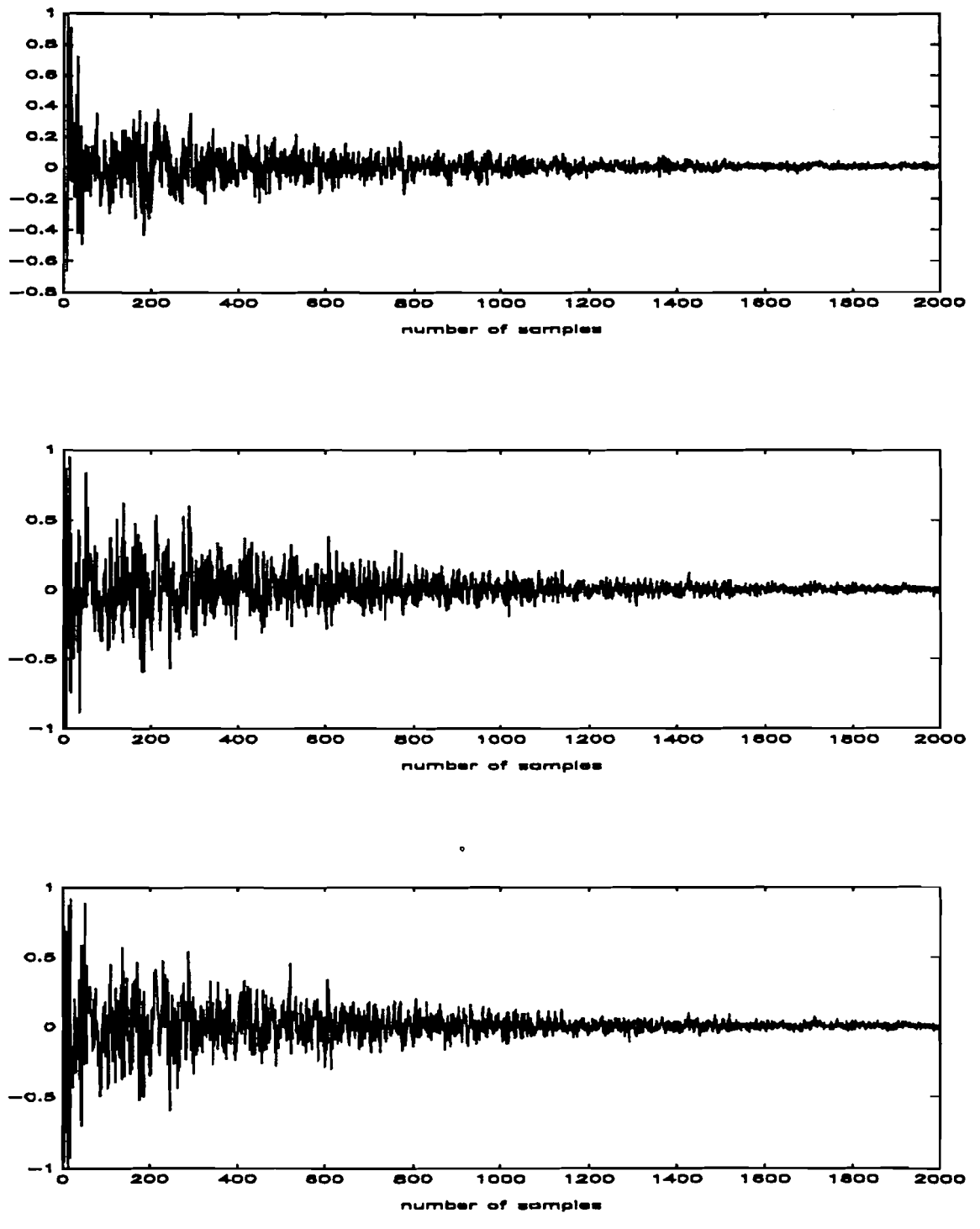


Figure A-1. Three different impulse responses of echo path sampled at 8 kHz.

## APPENDIX B DESCRIPTION OF PARAMETER SETTING PROGRAM

The parameters of the Decoupled Partitioned Block Frequency Domain Adaptive Filter are set by an user interface program. The echo canceller configuration can easily changed in this way. The interface takes care of a correct setting.

This user interface program runs on a PC, it consists of three files:

- SETPARAM.C - main C program to set parameters of DPBFDAF implementation.
- SETPARAM.DEF - include file with limits and initial settings.
- SETPARAM.CHK - include file containing all check procedures.

An executable SETPARAM.EXE can be obtained by using the following command

cl SETPARAM.C

During execution an assembly file, called SETTINGS.ASM, is created including all necessary parameters. This file will be linked with the object file of the adaptive filter and finally loaded onto the TMS320C30 DSP. A batch file DPBFDAF.BAT contains all program calls to establish an acoustic echo canceller, based on DPBFDAF.

The settings can be saved in a file SETPARAM.INI, the next run of SETPARAM will use these values as initial settings in stead of those in SETPARAM.DEF. If this file can not be found, the initial settings in SETPARAM.DEF are used.

Next restrictions of parameters are discussed, followed by a global program flow.

The parameters are restricted by the adaptive filter and implementation in the following way

CONSTRAINTS due to dpbfdaf

Number of adaptive weights N

Power estimation constant $\gamma$, $0 \leq \gamma \leq 1$

Parameters of filter part

| | | |
|---|---|---|
| Block length | B, | $1 \leq B \leq N$ |
| Partition length | Q, | $1 \leq Q \leq N$ |
| Partition factor | N/Q, | N/Q integer |
| DFT length | M, | $M \geq Q+B-1$ |

For efficiency reasons Q/B integer

Parameters of update part

| | | |
|---|---|---|
| Block length | A, | $1 \leq A \leq N$ |
| Partition length | Z, | $1 \leq Z \leq N$ |
| Partition factor | N/Z, | N/Z integer |
| DFT length | L, | $L \geq Z+A-1$ |

For efficiency reasons A ≥ B, Z/A and A/B integers

CONSTRAINTS due to implementation

DFT length DFT_min ≤ DFT length ≤ DFT_max and $\log_2$ (DFT length) integer

where DFT_min=16 and DFT_max=1024

REARRANGING of these CONSTRAINTS lead to

| | | |
|---|---|---|
| Power estimation constant | $\gamma$, | $0 \leq \gamma \leq 1$ |

Parameters of update part

| | | |
|---|---|---|
| Block length | A, | $1 \leq A \leq$ minimum (N, DFT_max/2) and N/A integer (additional) |
| Partition length | Z, | $A \leq Z \leq$ minimum (N, DFT_max-A+1) and N/Z integer |
| DFT length | L, | $L \geq$ maximum (DFT_min, Z+A-1), $L \leq$ DFT_max and $\log_2$ (L) integer |

Parameters of filter part

| | | |
|---|---|---|
| Block length | B, | $1 \leq B \leq A$ and A/B integer |
| Partition length | Q, | $B \leq Q \leq$ minimum (N, DFT_max-B+1), N/Q and Q/B integers |
| DFT length | M, | $M \geq$ maximum (DFT_min, Q+B-1), $M \leq$ DFT_max and $\log_2$ (M) integer |

Other limits on range of variables have been defined in SETPARAM.DEF. This include range limits of:

- sample frequency,
- filter length, and
- adaptation constant.

All these constraints have been implemented in the interface.

The user will not be aware of the following

- Parameters are automatically derived, such as

| | |
|---|---|
| partition factor | filter length / partition length |
| scale factor of IFFT | $(\text{FFT length})^{-1}$ |
| number of stages (I)FFT | $\log_2$ (FFT length) |

- Conversion of parameter

  sample frequency into sample period, a timer value to interrupt at sample rate

  sample period $\lfloor 10^6/(0.12*\text{sample frequency}) \rfloor$

  this number is truncated by float-to-integer conversion

Finally the program flow of the parameter setting program is represented at the following page.

PROGRAM FLOW OF SETPARAM

READ DEFAULT SETTING OF PARAMETERS

If setting of previous run has been saved in current directory (SETPARAM.INI exists), these default values will be used. Otherwise the initial settings defined in SETPARAM.DEF will be default.

INPUT AND CHECK OF PARAMETERS

| | |
|---|---|
| General | sample frequency, adaptation constant ($\alpha$), power estimation constant ($\gamma$) and number of filter coefficients (or length adaptive filter N) |
| Update part | block length (A), partition length (Z) and DFT length (L) |
| Filter part | block length (B), partition length (Q) and DFT length (M) |

SAVE CURRENT SETTING ?

If the answer to this question equals Y or y, the setting will be saved in a file SETPARAM.INI in the working directory. In this file names and values are stored, even as the purpose of this file. A former setting is deleted.

CONVERSION AND DERIVATION OF OTHER PARAMETERS

| | |
|---|---|
| Conversion | sample frequency into sample period |
| Derivation | partition factor, scale factor IFFT, number of stages (I)FFT |

PARAMETER SETTING ASSEMBLY FILE

The setting of parameters, necessary in the acoustic echo canceller, are saved in a file SETTING.ASM. This data file will be linked to the object file of the implementation.

DISPLAY CURRENT SETTING

## APPENDIX C DESCRIPTION OF ASSEMBLY ROUTINES

A more technical description of the assembly routines is included to explain restrictions, algorithms and some implementation details. These routines form a library to develop fast programs. By adding an interface the routines can be called in a high programming language, such as C and SPOX. Special implementation details are written inside the assembly routines.

### RFFT2B.ASM

Fast Fourier Transform (FFT) for real input data in bit reversed order. The FFT, based upon radix-2 butterflies, use decimation-in-time (DIT) in stead of decimation-in-frequency (DIF) decomposition to obtain an 'in-place' computation.

The minimum FFT length is 16, the maximum length equals 1024. The maximum is restricted by the table of twiddle factors. Enlargement of this table is possible, but will not be frequently used by reason of (internal) memory restrictions. The minimum size is restricted by 16, because of a separated implementation and testing after 4 stages. A minimum is 4, if the first two stages of radix-2 butterflies are still computed as radix-4 butterflies. In this case a test has to be included, that skips the other code.

If the length of the table of twiddle factors is changed the instructions to set the offset to the correct address of twiddle factors ($\pi/4$) in the FFT routines should also be changed.

The order of the output data of the Fourier transform is

R[0], R[1], R[2], .., R[N/2-1], R[N/2], I[N/2-1], .., I[2], I[1]

with N is the length of FFT,
R[k] represents the real part of X[k] and X[N-k], and
I[k] represents the imaginary part of X[k] and X*[N-k].

The complete transform can be composed out of this data by

X[0] = R[0]
X[1] = R[1]+jI[1]
X[2] = R[2]+jI[2]
..
X[N/2-1] = R[N/2-1]+jI[N/2-1]
X[N/2] = R[N/2]
X[N/2+1] = R[N/2-1]-jI[N/2-1]
X[N/2+2] = R[N/2-2]-jI[N/2-2]
..
X[N-1] = R[1]-jI[1]

FUNCTION Fast Fourier Transform

Description Radix-2 in-place DIT FFT for real input data

Variables length of FFT,
number of stages,
start address of (bit reversed input) data,
start address of table including twiddle factors.

Input data order real input data in buffer X of size $N=2^M$,
in bit-reversed order

Output data order R[0], R[1], R[2], .., R[N/2-1], R[N/2], I[N/2-1], .., I[2], I[1]

Algorithm

Compute first two stages as radix-4,
so for i=0 to N-4 do
begin
X[i] = {X[i]+X[i+1]}+{X[i+2]+X[i+3]}
X[i+1] = X[i]-X[i+1]
X[i+2] = {X[i]+X[i+1]}-{X[i+2]+X[i+3]}
X[i+3] = -{X[i+2]-X[i+3]}
end (increase i by $2^2=4$)

Compute stage 3 separate as radix-2, this establish a faster loop
(in order words, a better repeat block instruction utilization)
so, for i=0 to N-8 do
begin
X[i] = X[i]+X[i+4]
X[i+2] = X[i+2]
X[i+4] = X[i]-X[i+4]
X[i+6] = -X[i+6]
X[i+1] = X[i+1]+0.5·√2(X[i+5]+X[i+7])
X[i+3] = X[i+1]-0.5·√2(X[i+5]+X[i+7])
X[i+5] = -X[i+3]-0.5·√2(X[i+5]-X[i+7])
X[i+7] = X[i+3]-0.5·√2(X[i+5]-X[i+7])
end (increase i by $2^3=8$)

Compute following M-3 stages as radix-2
so for K=4 to M
$begin_1$

$N_K=2^{K-1}$ and $angle_{stage}=2\pi/N_K$

for i=1 to $N-2\cdot N_K$

$begin_2$

$X[i] = X[i]+X[i+2\cdot N_K]$

$X[i+2\cdot N_K] = X[i]-X[i+2\cdot N_K]$

$X[i+3\cdot N_K] = -X[i+3\cdot N_K]$

$angle = angle_{stage}$

for j=1 to $(N_K-1)$

$begin_3$

$angle = angle+angle_{stage}$

$X[i+j] = X[i+j]+\{X[i+j+2\cdot N_K]\cdot\cos(angle) + X[i-j+4\cdot N_K]\cdot\sin(angle)\}$

$X[i-j+2\cdot N_K] = X[i+j]-\{X[i+j+2\cdot N_K]\cdot\cos(angle) + X[i-j+4\cdot N_K]\cdot\sin(angle)\}$

$X[i+j+2\cdot N_K] = -X[i+j+2\cdot N_K]-\{X[i+j+2\cdot N_K]\cdot\sin(angle) + X[i-j+4\cdot N_K]\cdot\cos(angle)\}$

$X[i-j+4\cdot N_K] = X[i+j+2\cdot N_K]-\{X[i+j+2\cdot N_K]\cdot\sin(angle) + X[i-j+4\cdot N_K]\cdot\cos(angle)\}$

$end_3$ (increase j by 1)

$end_2$ (increase $N_K$ by $4\cdot N_K$)

$end_1$ (increase K by 1)

RIFFT2.ASM

Inverse Fast Fourier Transform (IFFT) for real output data in bit reversed order.

The structure of this program is basically the same as the FFT, except the data flows in opposite direction. The IFFT routine needs more processing time due to scaling. The scaling factor of $(FFT\ size)^{-1}$ is included to obtain a correct result in stead of a scaled one. The order of output is bit reversed! The bit-reversed addressing mode establish the reordering without extra processing.

FUNCTION Inverse Fast Fourier Transform

Description Radix-2 in-place DIF FFT for complex conjugate symmetric input data

Input data order output order of RFFT

Output data order real bit reversed order

Variables length of FFT,
number of stages,
scale factor
start address of (bit reversed input) data,
start address of table including twiddle factors.

Algorithm

Compute first M-3 stages as radix-2
so for K=1 to M-3 do
$begin_1$
$N_K=2^{N-(K+1)}$ and $angle_{stage}=2\pi/N$

for i=1 to $N-2\cdot N_K$
$begin_2$
$X[i] = X[i]+X[i+2\cdot N_K]$
$X[i+N_K] = 2\cdot X[i+N_K]$
$X[i+2\cdot N_K] = X[i]-X[i+2\cdot N_K]$
$X[i+3\cdot N_K] = -2\cdot X[i+3\cdot N_K]$

$angle = angle_{stage}$

for j=1 to $(N_K-1)$ do
$begin_3$
$angle = angle+angle_{stage}$

$X[i+j] = X[i+j]+X[i-j+2\cdot N_K]$
$X[i-j+2\cdot N_K] = X[i+j+2\cdot N_K]+X[i-j+4\cdot N_K]$
$X[i+j+2\cdot N_K] = \{X[i+j]-X[i-j+2\cdot N_K]\}\cdot\cos(angle)$
$-\{X[i+j+2\cdot N_K]+X[i-j+4\cdot N_K]\cdot\sin(angle)\}$
$X[i-j+4\cdot N_K] = \{X[i+j]-X[i-j+2\cdot N_K]\}\cdot\sin(angle)$
$-\{X[i+j+2\cdot N_K]+X[i-j+4\cdot N_K]\cdot\cos(angle)\}$

$end_3$ (increase j by 1)
$end_2$ (increase i by 1)

$stage_{angle}=2\cdot stage_{angle}$
$end_1$ (increase K by 1)

Compute stage M-2 seperate as radix-2, this establish a faster loop
(in order words a better repeat block instruction utilization)
so for i=0 to N-8 do
begin

$X[i] = X[i]+X[i+4]$
$X[i+2] = 2 \cdot X[i+2]$
$X[i+4] = X[i]-X[i+4]$
$X[i+6] = -2 \cdot X[i+6]$
$X[i+1] = X[i+1]+X[i+3]$
$X[i+3] = -X[i+5]+X[i+7]$
$X[i+5] = 0.5 \cdot \sqrt{2}\{X[i+1]-X[i+3]\}-0.5 \cdot \sqrt{2}\{X[i+5]+X[i+7]\}$
$X[i+7] = 0.5 \cdot \sqrt{2}\{X[i+1]-X[i+3]\}+0.5 \cdot \sqrt{2}\{X[i+5]+X[i+7]\}$

end (increase i by $2^3=8$)

Compute first last stages (M-1 and M) as radix-4,
so for i=0 to (N-4) do
begin

$X[i] = X[i]+X[i+2]+2 \cdot X[i+1]$
$X[i+1] = X[i]+X[i+2]-2 \cdot X[i+1]$
$X[i+2] = X[i]-X[i+2]-2 \cdot X[i+3]$
$X[i+3] = X[i]-X[i+2]+2 \cdot X[i+3]$

end (increase i by $2^2=4$)

Scaling of output data

for i=0 to FFT size (N)
begin

$X[i] = X[i] \cdot (\text{fft size})^{-1}$, where $(\text{fft size})^{-1}$ is the scale factor

end

Table C-1: Number of instructions needed for R(I)FFT

| FFT length | RFFT TI | RFFT optimized | RIFFT TI | RIFFT optimized |
|---|---|---|---|---|
| 16 | 276 | 172 | 295 | 194 |
| 32 | 619 | 392 | 671 | 432 |
| 64 | 1382 | 898 | 1515 | 974 |
| 128 | 3073 | 2056 | 3399 | 2204 |
| 256 | 6796 | 4678 | 7571 | 4970 |
| 512 | 14935 | 10548 | 16735 | 11128 |
| 1024 | 32610 | 23554 | 36715 | 24710 |

Table C-2: Processing time needed for R(I)FFT [ms]

| IFFT length | RFFT TI | RFFT optimized | RIFFT TI | RIFFT optimized |
|---|---|---|---|---|
| 16 | 0.0166 | 0.0104 | 0.0177 | 0.0117 |
| 32 | 0.0372 | 0.0236 | 0.0403 | 0.0260 |
| 64 | 0.0830 | 0.0539 | 0.0909 | 0.0585 |
| 128 | 0.1844 | 0.1234 | 0.2040 | 0.1323 |
| 256 | 0.4078 | 0.2807 | 0.4543 | 0.2982 |
| 512 | 0.8961 | 0.6329 | 1.0041 | 0.6677 |
| 1024 | 1.9566 | 1.4133 | 2.2029 | 1.4826 |

Table C-3: Processing performance of optimized RIFFT, including scaling.

| length | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|
| cycles | 222 | 476 | 1050 | 2344 | 5238 | 11652 | 25746 |
| time | 0.0133 | 0.0286 | 0.0630 | 0.1407 | 0.3143 | 0.6992 | 1.5448 |

ELEMPROD.ASM

An elementwise vector product routine should use the symmetry of the Fourier transformed data. This routine will be used in the update part of the adaptive filter and to compute the first vector product of the non delayed input signal x in the convolution process.

The elementwise product of two vectors X and Y of length N

$$
\begin{aligned}
&X[0]\cdot Y[0] && = R_X[0]\cdot R_Y[0]\\
&X[1]\cdot Y[1] && = \{R_X[1]\cdot R_Y[1]-I_X[1]\cdot I_Y[1]\} + j\cdot\{R_X[1]\cdot I_Y[1]+R_Y[1]\cdot I_X[0]\}\\
&.. &&\\
&X[N/2-1]\cdot Y[N/2-1] && = \{R_X[N/2-1]\cdot R_Y[N/2-1]-I_X[N/2-1]\cdot I_Y[N/2-1]\}\\
&&& \quad + j\cdot\{R_X[1]\cdot I_Y[1]+R_Y[1]\cdot I_X[0]\\
&X[N/2]\cdot Y[N/2] && = R_X[N/2]\cdot R_Y[N/2]\\
&X[N/2+1]\cdot Y[N/2+1] && = \{R_X[N/2-1]\cdot R_Y[N/2-1]-I_X[N/2-1]\cdot I_Y[N/2-1]\}\\
&&& \quad - j\cdot\{R_X[1]\cdot I_Y[1]+R_Y[1]\cdot I_X[0]\\
&.. &&\\
&X[N-1]\cdot Y[N-1] && = \{R_X[1]\cdot R_Y[1]-I_X[1]\cdot I_Y[1]\} - j\cdot\{R_X[1]\cdot I_Y[1]+R_Y[1]\cdot I_X[0]\}
\end{aligned}
$$

In formula notation $X \otimes Y$

In this way the data can directly be transformed back to time domain, so no overhead is introduced.

FUNCTION Elementwise vector product of two vectors

Variables length of vector
start addresses of both input vectors and output vector

Input data order output order of FFT

Output data order output order of FFT or input order of IFFT

Algorithm

if i=0 to N/2 then R[i] = X[i]·Y[i]

for i=1 to N/2-1 do
begin
R[i] = X[i]·Y[i] - X[N-i]·Y[N-i]
I[N-i] = X[i]·Y[N-i] + X[N-i]·Y[i]
end

Modifications to elementwise vector product routine, that establish the computation of a complex conjugate vector multiplied with (normal) vector.

In formula notation $X^* \otimes Y$

FUNCTION Elementwise vector product of two vectors (one complex conjugated)

Variables length of vector
start addresses of both input vectors and output vector

Input data order output order of FFT

Output data order output order of FFT or input order of IFFT

Algorithm

if i=0 to N/2 then R[i] = X[i]·Y[i]

for i=1 to N/2-1 do
begin
R[i] = X[i]·Y[i] + X[N-i]·Y[N-i]
I[N-i] = X[i]·Y[N-i] - X[N-i]·Y[i]
end

This implies a interchange of add and subtract operations.

## CONVOLUT.ASM

The convolution in frequency domain consists of multiplication of two vectors and adding of result to previous (intermediate) result. A minimum of two input vectors is assumed, otherwise the test condition inside this routine should be changed or the elementwise product routine (elemprod.asm) can be used in stead.

FUNCTION Convolution in frequency domain

Variables length of vector
number of products (= partition factor of filter part in DPBFDAF)
start addresses of input vectors (X and Y) and result vector (Z)

Input data order output order of RFFT

Output data order output order of RFFT or input order RIFFT

Algorithm

$X_0 \otimes Y_0 = Z$

for i=1 to N/Q-1 do
begin
$X_i \otimes Y_i + Z = Z$
end

## RESIDU.ASM

The residual signal r[k] equals the unknown echo signal e[k], added to the desired speech signal s[k] minus the estimated echo signal ê[k]. In formula notation

$$r[k] = (s[k] + e[k]) - \hat{e}[k] = \tilde{e}[k] - \hat{e}[k].$$

The microphone signal $\tilde{e}[k]$ represents the unknown echo signal e[k], added to the desired speech signal s[k]. The estimated echo signal ê[k] is the convolution of the vectors x (input signal) and w (filter coefficients). The estimate ê[k] of the adaptive filter must be bit reversed, because of the data output order of the inverse FFT routine.

FUNCTION Residual output signal

Variables length of vector,
start address of input vectors ($\underline{\hat{e}}$[kB] and $\underline{\tilde{e}}$[kB]), and
output vector ($\underline{r}$[kB]).

Input data order $\underline{\hat{e}}$[kB] is bit reversed, $\underline{\tilde{e}}$[kB] in normal order

Output data order $\underline{r}$[kB] is bit reversed

Algorithm
skip M-B elements of vector $\underline{\hat{e}}$[kB]
$\underline{r}$[kB] = $\underline{\hat{e}}$[kB] subtracted from ($\underline{\tilde{e}}$[kB] in bit reversed order)

## SETTINGS.ASM

This file defines all initial settings, such as FFT size, number of stages and scaling factor.

Input and verification of parameters is realized in a high level programming language. The linker will link this file to the object file of program routines. This option offers a flexible and reliable user interface.

## TWIDTABL.ASM

The table of twiddle factors includes all necessary sine and cosine values to calculate a 1024-FFT. The cosine and sine values alternates starting at 0 up to $\pi/4$. It fits exactly in one of the two internal memory RAM blocks of each 1K words.

The following relationships of the twiddle factor $W_N$ hold

- $W_N^k = -W_N^{k+(N/2)}$ (symmetry property)
- $W_N^k = W_N^{k+N}$ (periodicity property)

## DPBFDAF.BAT

Batch file to create a new configuration of echo canceller, in which consecutively

- setting of parameters
- parameters are automatically linked into the executable code of the echo canceller
- execution of echo canceller configuration on DSP

## DPBFDAF.CMD

Linker command file, that specifies the total memory mapping of the implementation on the DSP.

## FILTER.ASM

Filter part of adaptive filter, containing calls of subroutines

- fast Fourier transform
- convolution (in frequency domain)
- inverse fast Fourier transform
- extraction of residual signal

## DPBFDAF.ASM

Main assembly program to

- definition of interrupt vectors
- definition of interrupt service routines
- definition of parameters
- initialization of DSP peripherals
- endless loop to compute filter part and update part of adaptive filter

## DSP_INIT.ASM

Initialization of TMS320C30 DSP, including

- initialize stack pointer to save and restore values during context switch in reserved data space
- initialize timer 1, dedicated to D/A and A/D converters, to generate an interrupt (interrupt 1) at sample frequency
- enable interrupts
- set status

## ADC_DAC.ASM

Interrupt service routine to read input samples of signals x[k] and $\hat{e}$[k] from ADCs and write output samples of signal r[k] to DAC. The analog signal is limited at ± 3 Volt. The analog-to-digital converter converts this signal level to a 16 bits digital representation, including a sign bit.

The interface to analog signals provides two channels for input and two for output. The interface is accessed through 16 bit registers, which are connected to the C30 expansion bus. These registers are accessed with two memory wait states. This results in an overall time of 180 ns to access each register. The A/D's and D/A's use the same register for output and for input. The A/D must be read before the D/A is written, if different samples appear on the same channel during the same time interval.

Timer 1 of the C30 is used for sample interval control. It initiates the A/D and D/A conversions at a rate, defined by the sampling period. The timer must be set with a value equal to the desired sampling period in µs divided by 0.12. The counter, inside the timer, is namely continuously incremented once every 120 ns. Interrupt 1 is reserved to read and write one or both A/D's and D/A's. The interrupt drives control to the interrupt service routine by using the interrupt vector table.

Type conversion of integer to float values has to be included. Storage at the correct memory location should be provided within this routine.

## APPENDIX D EXAMPLE COMPUTATIONAL LOAD OF ROUTINES

| | | | |
|---|---|---|---|
| Number of adaptive weights | N | 2016 | |
| Sample frequency | $f_s$ | 8 kHz | (telephony standard) |
| TMS320C30 DSP | $f_{clock}$ = 16,67 MHz | | (1 instruction/ 60 ns.) |
| Instructions/sample | 2083 | | |

Parameters of DPBFDAF

| | Filter Part | | Update Part | |
|---|---|---|---|---|
| Block length | B | 8 | A | 504 |
| Partition factor | Q | 56 | Z | 504 |
| FFT length | M | 64 | L | 1024 |

| | |
|---|---|
| Instructions/(filter block) | 16664 |
| Instructions/(update block) | 1049832 |

Computational load example of adaptive filter, without normalization, communication, and input and output routines.

| | | Routine | Implementation | |
|---|---|---|---|---|
| Filter Part | | | | |
| $DFT_M$ | ·1 | 900 | | |
| $IDFT_M$ | ·1 | 1050 | | |
| Convolution | ·1 | 9505 | | |
| Residu | ·1 | 95 | | |
| | | ——— + | | |
| Sub total | | 11550 | | |
| | ·A/B | | 727650 | (70%) |
| Update Part | | | | |
| $DFT_L$ | ·2 | 47110 | | |
| $IDFT_L$ | ·N/Z | 102990 | | |
| Correlation | ·1 | 13000 | | |
| | | ——— + | | |
| Sub total | | 163100 | | |
| | | | 163100 | (16%) |
| Interfaces | | | | |
| $DFT_M$ | ·N/Q | 32350 | | |
| | | | 32350 | ( 4%) |
| | | | ——— + | |
| Grant total | | | 923100 | (90%) |

Optimization profit of fast Fourier routine 148130 (14%)

# APPENDIX E INTERFACING ASSEMBLY AND C/SPOX

Accessing the assembly routines within a high-level language, such as C, can be useful to speed up the computation process.

The real FFT assembly routine is used to demonstrate the way of interfacing. A real and bit-reversed input sequence is transformed from time to frequency domain. Two examples have been programmed in C and SPOX.

It is relative simple to call assembly routines within a SPOX program. SPOX features communication routines between PC and DSP, that automatically uses the PC as input and output monitor. The interface in assembly defines the order and type of arguments to be passed, no particular knowledge of assembly is necessary.

The assembly language function is called as every function in a SPOX or C program. The only difference is the external declaration of the function. The assembly function needs some kind of shell or interface to communicate with the SPOX program. Separate modules of assembled code can be linked into code of a compiled SPOX program. Execution of this code is the same as other SPOX programs.

The environment may not be disrupted by the context switch to the assembly program. The assembly routine must be preceded and followed by some instructions, that make the context switch possible. In the interface the following conventions have to be met

- the name of the assembly routine (start point of execution) should begin with an underscore (_). The compiler appends an underscore to the beginning of all identifiers.

- the function must be declared with the .global directive. This defines the function as external and allow the linker to resolve references to it.

- variables passed as arguments to assembly must also be declared with the .global directive and the .bss directive is used to reserve memory for variables.

- the function call of the assembly routine must be preceded with
  (a) instructions to save some dedicated registers
  (b) move instructions of arguments into locations, matching the names of variables in assembly.

- the function call must be followed with instructions to restore the environment and ended with a return instruction.

A more detailed description, to interface without violation of conventions, can be found in chapter 4 of [34].

The following improvements could be included:

- Allocation and deallocation of memory dynamically during program execution, in stead of a fixed memory allocation. A memory manager will be useful for this purpose.

- Execution in internal memory will speed up the computation process, but data (already located in this memory) may need to be saved and restored. A function can be written quite easily to establish this.

In SPOX the communication between PC and DSP is automatically taken care for. However in C programs this communication must be set up by the programmer. A TMS320C30 High-Level Language Interface Library is available. These library routines are used to download TMS320C30 object code to the DSP board, to start execution of that code and to pass data between the program running on the TMS320C30 and on the PC. By using this library it is possible to create a communication environment, like that offered in SPOX. Assembly programmers can run and control programs in this way. Programmers, unfamiliar with assembly, should be confronted with a defined communication.

The library functions transfer data directly to and from memory space of the DSP. By using these functions more knowledge of the assembly implementation is asked. The assembly routine, developed in the debugger or simulator, need some little changes. Before execution it should test, if the data is ready. Afterwards it should indicate, it has been finished. The assembly routine should be assembled into executable code. The C program is compiled in the way, it would be normal PC code. Although the processing takes partly or almost entire place at the DSP. In the executable (object) code of the assembly routine variables are put at a linker defined place. These locations can be accessed directly by reading and writing.

To avoid errors a communication library should be offered, if assembly routines or even complete filters are accessed within C. The memory address and type of variables must be defined explicitly to guarantee a correct operation. An alternative could be a well defined interface in some special reserved memory space, to pass variables between PC and DSP.

In the final implementation the download and start (or reset) function have to be used. Once a program is written and debugged using the C interface library, it is possible to substitute the faster assembly library with only minor modification (necessary due to floating point representations). The C libraries have error checking logic included, which is not in the assembler code versions of the library.

# APPENDIX F SOFTWARE DEVELOPMENT SUPPORT

## Debugger

At this moment two debug monitors are available, called MON30 and SDS30. Such monitors can load programs to the board, run them and provide the tools to debug them. The debug process allows to execute the program under user control in different ways. After a stop of execution the state of the processor can be checked.

Advantage debug monitor

- programs runs on the TMS320C30

Disadvantages debug monitor

- TMS320C30 DSP board necessary
- cycle counting is not included
- cache utilization is unknown
- interface can not be user customized
- pipeline conflicts are not shown
  (wait states are inserted automatically)

## Simulator

A TMS320C30 Simulator is a software program, that simulates the TMS320C30 DSP for software development and program verification in non real-time. With the simulator it is possible to debug without the DSP. The simulator simulates the operation and allows monitoring of the state of the DSP. Simulation speed is typically on the order of hundreds of instructions per second. That will be fast enough at this stage of development.

Advantages simulator compared to debug monitor(s)

- freedom in development surrounding,
  resulting in faster development time
- cycle counting including wait states
- trace execution and display of cache memory
  and instruction pipeline
- simulation of cache utilization

- interface can be user customized
- interrupt generation at own defined intervals
- files can be associated with I/O parts, resulting in better test conditions
- execution of commands in journal file

The simulator offers a sophisticated development surrounding. This tool will be helpful to locate problems and to utilize the DSP efficiently. It is possible to develop programs at a personal computer. This reduces or even removes the dependency to other DSP users. The TMS320C30 DSP will be free for computational intensive applications. The debug monitors do not show the utilization of the DSP. If conflicts arise wait states are automatically inserted to solve this situation, without reporting this situation to the user.

A complete debug environment is necessary to allow a relative fast development and execution of assembly routines. The best way to develop software starts by dividing a problem in sub problems, describing them in general terms, translating it into an algorithm and writing it in the programming language (assembly coding). If data has been located in internal memory, then the code can be executed in parallel. An optimal code is more difficult to write, because of the restricted parallel instruction set. This assembly routine will probably include pipeline conflicts, that can be traced by the simulator. Rearranging of instructions can result in a (near) optimal processor implementation. The debug monitor must be used to test the whole implementation, preceded by testing parts. Programs, developed in the simulator, can not run directly on the DSP. Initialization of the DSP will be necessary, including definition of the interrupt vector table and setting of DSP bus control.