AN1384
APPLICATION NOTE

Implementing an Acoustic Echo Canceller Algorithm using the ST120DSP

By Aude ANDOLFATTO

1 - INTRODUCTION

This application note describes an Acoustic Echo Canceller (16 bits, 8KHz, 256ms) implementation on the ST120 DSP.

The first chapter presents the general principle of an AEC based on the NLMS (Normalized Least-Mean-Squared) filtering.

The second, third, and fourth chapters detail the main parts of an AEC: nlms filtering, power estimation, and speech detection.

The sixth and seventh chapters provide results concerning C floating point and C fixed point implementation.

The last chapter describes the assembly implementation of the NLMS function and gives results in terms of MCPS (Mega Cycle Per Second - of speech) and code size.

TABLE OF CONTENTS

2 - AEC ALGORITHM

2.1 - General Purpose

The coupling between a loudspeaker and a microphone generates some specific problems. The loudspeaker signal is echoed back to the microphone and transmitted back to its origin. As a result, the far-end participant perceives this as an echo. The longer the transmission delay is, the more disturbing the acoustic echo is.

To eliminate the acoustic feedback, an Echo Canceller is introduced in the loudspeaker. By simulating the acoustic echo path, the echo cancellation filter synthesizes a replica of the echo signal, which is subtracted from the signal on the return path.

Figure 1 : Echo Canceller Configuration

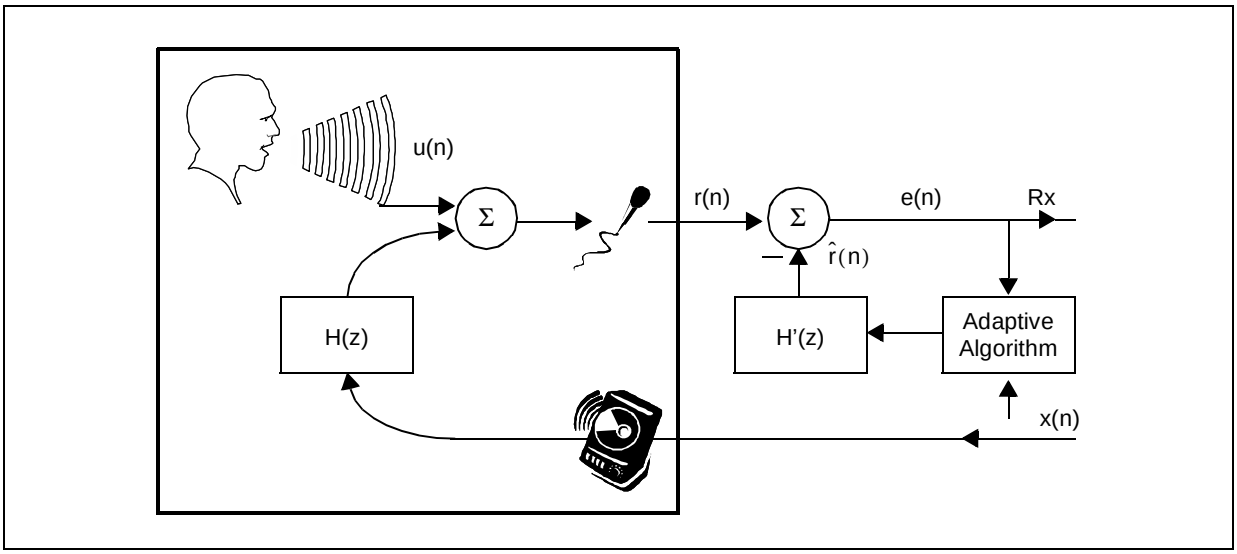


Table 1 : Echo Canceller Configuration Terminology

n	Sample time
x(n)	Far-end speech
u(n)	Near-end speech
r(n)	Signal received by the microphone, also called "near-end signal"
$\hat{r}(n)$	Estimated echo
e(n)	Residual echo
H(z)	Echo path
H'(z)	Estimated echo path

2.2 - NLMS Algorithm

Several adaptive algorithms have been proposed for acoustic echo cancellers. One of them is the NLMS algorithm, which is widely used.

This algorithm attempts to minimize the expected value of the squared error (residual echo). It starts from the initial (arbitrary) value for the tap weight vector, which is improved with the number of iterations.

See the NLMS implementation section for more information.

2.3 - Restrictions

Problems with AEC are double talk conditions, where both operators are speaking at the same time. If not detected, DT(Double Talk) can cause divergence of the adaptive algorithm.

Moreover, when the far-end operator is silent and the near-end operator is talking, the filter must not be adapted because the near-end operator is no longer an echo.

The following table summarizes the AEC principle:

Table 2 : AEC conditions and actions

Conditions	Actions
far-end speech alone	nlms algorithm: filtering and update
double talk	Filtering BUT NO update
near-end speech	Residual echo = near-end signal

3 - NLMS ALGORITHM

The following steps constitute the NLMS algorithm:

– Adaptive filter output: $\hat{r}(k) = w(k)x(k)$

– Estimation error: $e(k) = r(k) - \hat{r}(k)$

– Tap weight filter: $w_k(n+1) = w_k(n) + \frac{\mu e(n)}{P(n)} \times x(n)$

where $\frac{\mu e(n)}{P(n)}$ is NLMS constant at given sample time n

and where μ is the step size.

and *P(n)* is the estimated power of far-end speech at sample time n.

The short window power estimate of far-end speech (fes_short_pwr) is used to normalize the step size.

The generic equation for estimating the average power is:

$P(n) = (1-\rho)P(n-1) + \rho \times x^2(n)$, where ρ constant.

NLMS not only performs FIR filtering, but also updates the filter coefficients.

4 - SIGNAL POWER ESTIMATION

The signal power estimation is used in AEC to normalize the loop gain (step size). In addition the power estimation outputs are used by the speech detection function to determine the sequence of operations to be performed in that function.

Following equation uses the input squared to calculate the signal power.

$$P(n) = (1-\alpha) \times P(n-1) + \alpha \times X^2(n)$$

where

$\alpha = \frac{1}{32}$ for a very short window power estimate, 4ms

$\alpha = \frac{1}{128}$ for a short window power estimate, 16ms

$\alpha = \frac{1}{16384}$ for a long window power estimate, 2048ms

A different value is chosen for each different window-size power estimate.

The far-end signal power (fes_short_pwr) is estimated by using a short window size of 16ms. This estimate is used in the NLMS algorithm to normalize the step size. A window of 4 ms is used to determine the very short power estimates of near-end power (nes_vshort_pwr) and far-end power (fes_vshort_pwr). These estimates are used in far-end and near-end speech detection.

It is important that the functionality of the speech detectors be accurate to avoid erroneous detection, which could lead to an unstable system.

5 - SPEECH DETECTION

5.1 - Introduction

Speech detection is a very important part of AEC. It must be done before the software can determine whether to filter, update or freeze the adaptive filter. There are three speech detectors:

– Far-end speech detector

– Double-talk detector

– Near-end speech detector

The speech detection software always checks for the presence of the far-end speech first, then it goes to double-talk detection. It performs double-talk detection even if it does not detect far-end speech. This avoids false detection due to small signal level of far-end speech. If the software does not detect either far-end speech or double-talk, it goes to near-end speech detection. All detection is based on the signal power estimate algorithm, which is discussed in detail in Signal Power Estimation section of this report.

5.2 - Far-end Speech Detection

Far-end speech means that only the far-end speaker is active. This is the only time the AEC program performs both filtering and updating. The very short power estimates of the far-end and near-end speech signals are used to determine if far-end speech is present. So, far-end speech is detected only if $fesvshortpwr + FESMARGIN > nesvshortpwr$.

where FESMARGIN is a threshold constant.

The value of the threshold constant must be chosen carefully from real-time experiments. If the threshold value is too small, the background noise picked up by the microphone results in a false detection. On the other hand, if the threshold value is too large, part of the speech is not transmitted when the speech signal levels are low.

5.3 - Double Talk Detection

In AEC algorithms, the presence of both far-end speech and near-end speech is known as double-talk. The following equation implements and defines a double-talk detector based on echo return loss enhancement (ERLE):

$$ERLE = 10\log\left(\frac{P^2_u(n)}{P^2_e(n)}\right)$$

where ERLE equals to 8 dB (chosen from real-time experiments), $P_u(n)$ is the short window power estimate of the near-end signal and $P_e(n)$ is the short window power estimate of the residual error signal.

So, double-talk is detected if

where $C = 10^{((ERLE)/10)}$ and D is a threshold constant, determined from real-time experiments.

The higher the value of the D constant, the less double-talk is detected, but the more the coefficients will be updated. In addition, a higher threshold constant results in a greater difference between the echo and the echo replica which means less echo cancellation, but it also results in less noise interference.

After double-talk is detected, the program freezes the FIR filter's coefficients updates; however, filtering is still done, and the double talk hangover counter is reset to high (see the Hangover Counters section for more information).

5.4 - Near-end Speech Detection

Near-end speech exists when there is no far-end speech and no double-talk. Near-end speech is detected by calculating the very short power estimate and the very long power estimate of the near-end signal as follows $nesvshortpwr + NESMARGIN > neslongpwr$.

where NESMARGIN is a threshold constant chosen from real-time testing.

If near-end speech is detected, the program sets the near-end speech mode bit and freezes the LMS adaptive filter function. Indeed, when the far-end operator is silent and the near-end operator is talking, the filter must not be adapted because the near-end operator is no longer an echo.

5.5 - Hangover Counters

Two hangover counters are used in the speech detection algorithm:

– DT_HANG

– NES_HANG

Each hangover counter is set to a hangover time of 400 samples or 50 ms after its corresponding type of speech is detected (assuming that the sampling frequency is 8 kHz). If a type of speech is not detected, its hangover counter is decreased by one. Table 3 shows how the counters determine when to do filtering, updating, filtering and updating, or nothing.

Hangover counters play a very important role in AEC algorithms. After each different speech is detected, its corresponding mode bit is set. For example, the FAR_SPEECH mode bit sets to 1 to indicate that only far-end speech is detected. The AEC_UPDATE mode bit is not set to 1 until the double-talk and near-end hangover counters are both less than zero. This method avoids erroneous detection and gives some buffer time to turn on the adaptive filter.

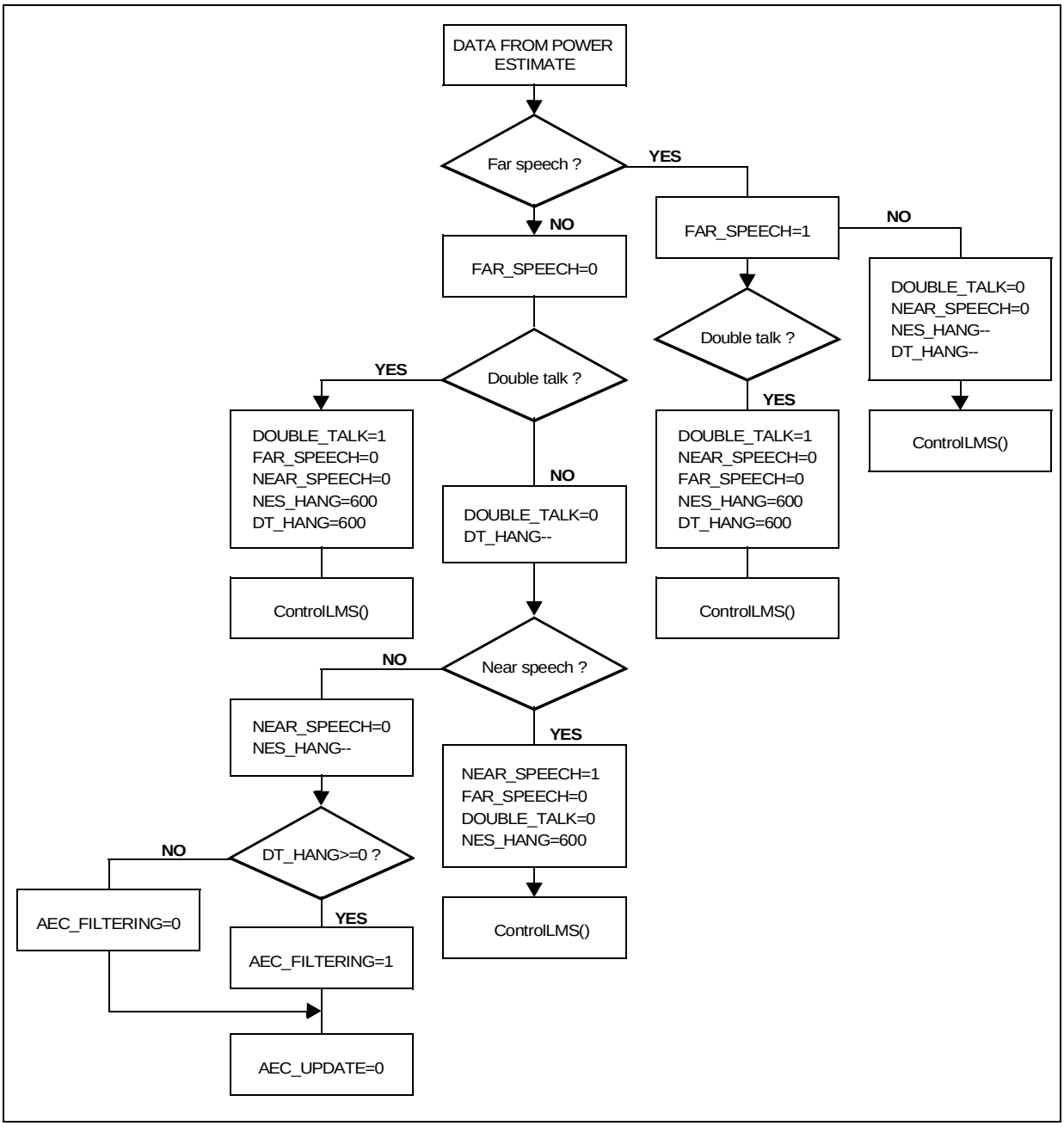
Table 3 : Hangover Counters and LMS Mode Bit Settings

HANGOVER COUNTERS		SET FILTERING MODE BIT	SET UPDATING MODE BIT
DT_HANG>=0	NES_HANG>=0	YES	NO
	NES_HANG<0	YES	NO
DT_HANG<0	NES_HANG>=0	NO	NO
	NES_HANG < 0	YES	YES

5.6 - Speech Detection Program Flow

The following flowchart shows the speech detection program flow.

Figure 2 : Speech Detection Flowchart



6 - MEASUREMENTS

6.1 - Algorithm Efficiency

6.1.1 - Input Test Files

Input test files consist of:

o. A reference wave file (far-end speech) and its associated near-end signal (addition of near-end speech and echo) in which the near-end speaker is silent so that near-end signal is only containing echo.

o. The same reference file and a new associated near-end signal in which, this time, near-end speaker is not remaining silent. In this file, double-talk is present.

6.1.2 - Performances Assessment

The assessment is quite similar to one recommended by the ETSI.

The terminal coupling loss (TCL) is computed as follows:

$$TCL = 10\log\left(\frac{\sum_{n} x^2(n)}{\sum_{n} e^2(n)}\right)$$

It computes the total attenuation between the receiving port and the sending port of the far-end side.

6.2 - Tools Description

For each compilation the Green Hills compiler used is the one present in GHS-ST100-2.1-01 MULTI 2000 environment . This compiler provides options that enable to improve performance and code size. The "speed optimization" (-OLAMI) and the "disable loop unrolling" (-Onounroll) options are used.

Using LAO (Linear Assembly Optimizer [1]) still improves performance: it is able to exploit instruction-level parallelism and automatically manages operator expansion, register allocation and loop optimization.

LAO is used as follows: the assembly file generated during the GHS compilation (with options) is recalled as file.lai (input file for LAO). Then the LAO is used to improve this file. A new "optimized" assembly file is generated and can be used for GHS compilation. Option used is : -Osliw -Ounroll

6.3 - Benchmark Parameters

6.3.1 - Number of (mega) cycles per second of speech (MCPS)

To measure performances, breakpoints were set in the assembly before and after the call of the functions. In the debugger window, the "cycle" command is then called at each breakpoint. The difference of both return values gives the number of cycles to perform the function (call and rts included). Then this number just has to be multiplied by 8000 (sampling rate equals to 8 kHz). The result is the number of MCPS.

6.3.2 - Code Size

The code size indicated in the further tables is the size given by the "gfunsize" command. This command (with as argument the object file) returns the number of bytes for each function.

7 - C FLOATING POINT IMPLEMENTATION

7.1 - Role

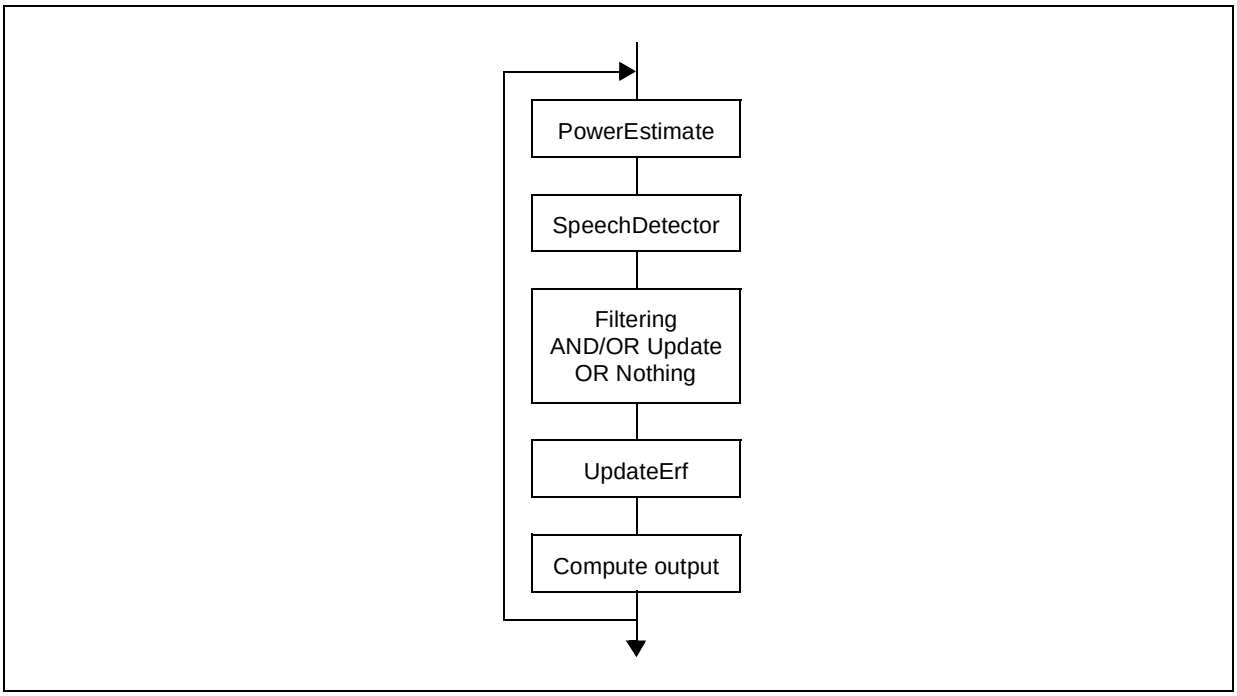
Regarding the dynamic precision range of floating variables, the aim of such an implementation is to obtain a reference system. All further fixed point (ASM, LAI, C ...) implementations should have the same properties (convergence, same output file for the same input files...).

7.2 - Program Flow

The following figure presents the process developed.

As described in the previous chapters, the PowerEstimate function is used to estimate powers with different window lengths. These powers are used first as parameters for the SpeechDetector function and then to update the step size (UpdateErf function). The SpeechDetection function determines what kind of signal the system is receiving (far-end signal, near-end speech, double-talk...) and if it has to filter and/or update.

Figure 3 : Program Flow



7.3 - Convergence Properties

Note: Following results have been determined with constants and threshold fixed to give the bests results with THESE files. It is obvious that testing the system on a wide range of test files is needed to validate completely the values of constants and threshold.

7.3.1 - Tests in Quiet Environment

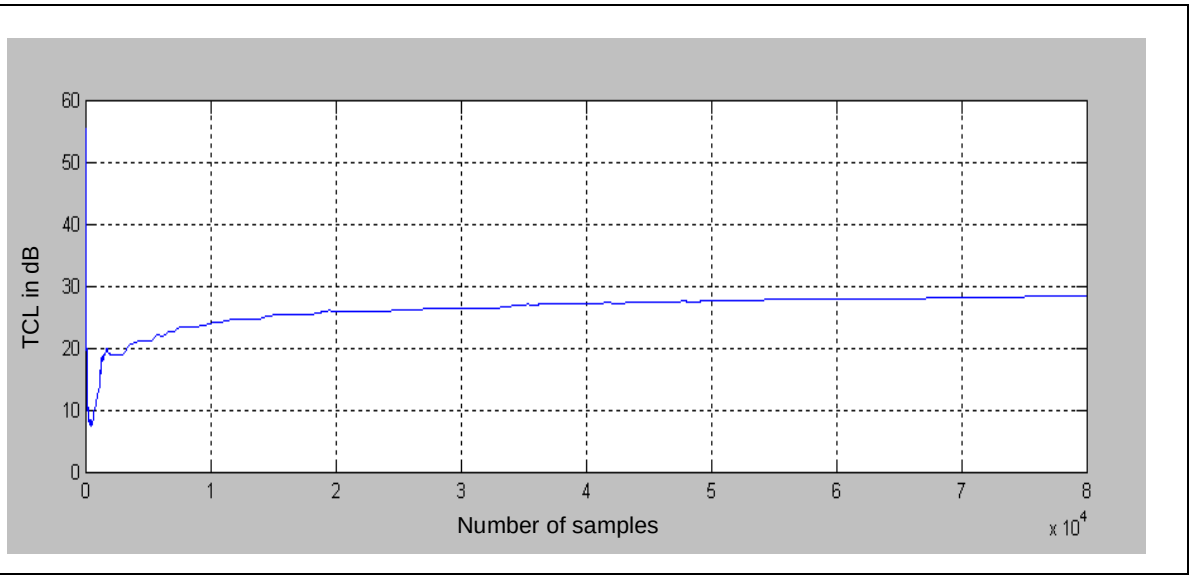
As specified by the ETSI, the measurements were taken after 10 seconds of speech to allow algorithm to reach a steady state.

The result is 28.45 dB.

The initial convergence time is good. The TCL almost reaches its maximum value after two seconds.

The following figure represents the TCL(in dB) versus the number of samples.

Figure 4 : TCL(in dB) versus the number of samples in Quiet Environment



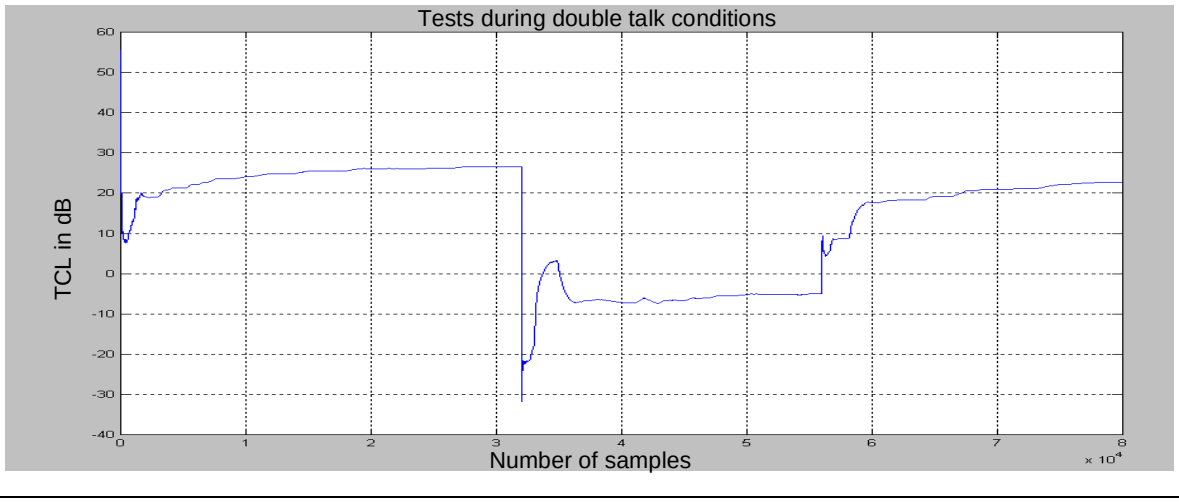
7.3.2 - Tests During Double-talk Conditions

The file used for the test was composed of a single talk period of 4 seconds, followed by a succession of double talk and near-end speech for 3 seconds, then the single talk condition returned.

When double-talk occurred, the TCL computation was initialized.

Following figure shows that the filter is no more adapted at all to the signal during double-talk period. An excellent recovery time is realized after double-talk. Following double-talk, the TCL resumes its average value within 2-3 seconds.

Figure 5 : TCL(in dB) versus the number of samples in Double-Talk conditions



7.4 - Bench Results

In this paragraph the best results are presented. GHS options are -OLAMI -Onounroll and LAO options are -Ounroll=1 .A library of floating point intrinsics was used (fast float addition, fast float mac, fast float multiplication and division).

It is important to note that the call and the rts instructions are included in all MCPS results. However, the number of MCPS is sufficient to hide the cycles due to the CALL/RTS instructions impact.

Table 4 : Bench Results

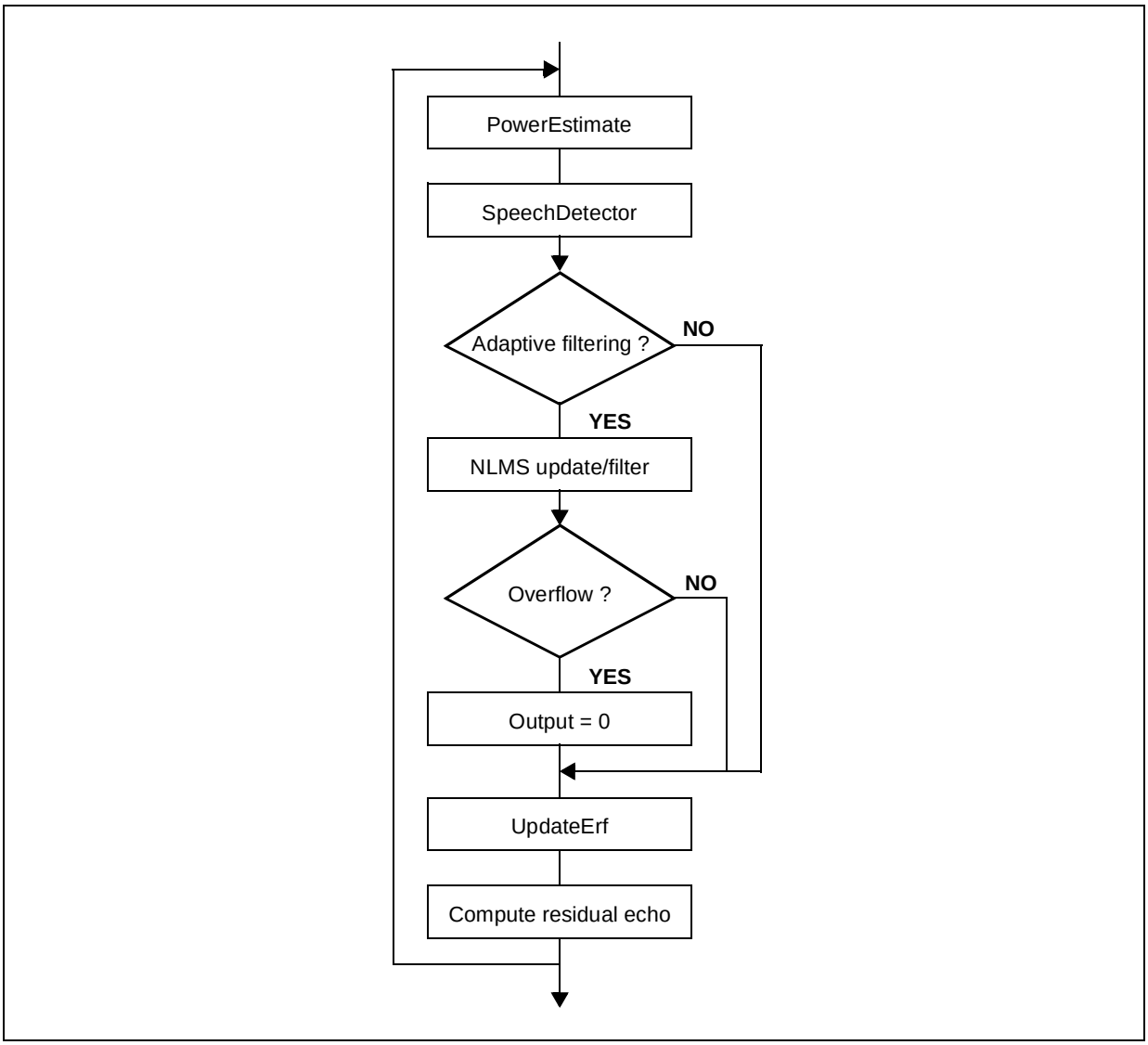
Function Name	Nb of Cycles/New Sample	Code Size
InitOutPower	458	236
PowerEstimate	2163	376
SpeechDetector	1133	976
ControlLMS		144
NLMSFiltering	40258	176
CalcUpdate	393	88

8 - C FIXED POINT IMPLEMENTATION

8.1 - Program Flow

The following flowchart describes the AEC process.

Figure 6 : AEC Flowchart



The difference with the floating point version is the test of overflow. It is important to check overflows as they could make the algorithm diverge, if not treated.

8.2 - C Prototypes

Table 5 through Table 8 show the protoypes used for the LMS adaptive filter function, the power estimate function, the speech detection function and the calc erf function.

Table 5 : Prototype for AEC NLMS Adaptive Filter Function

AEC NLMS Adaptive Filter Function Prototype	
Syntax	Void NLMS_filtering(short *input, short erf, short out)
Parameters	*input : table containing last 255 input samples + new input sample erf: normalized error times stepsize out: output sample
Description	This functions is called to adapt the filter coefficients and calculate an FIR filter output by using samples input and previous error

Table 6 : Prototype for AEC Power Estimate Function

AEC Power Estimate Function Prototype	
Syntax	void PowerEstimate(short FE, short NE, short Error, int *FEpwr, int *NEpwr, int Errpwr)
Parameters	FE : new far-end signal sample NE: new near-end signal sampleError: new error signal sample *FEpwr : point to far-end powers (short, vshort, long power estimates) *NEpwr : point to near-end powers (short, vshort, long power estimates) Errpwr : short error power estimate
Description	This function is called to estimate the signal power which is used in speech detection and to normalize the step size.

Table 7 : Prototype for AEC Speech Detection Function

AEC Speech Detection Function Prototype	
Syntax	Void SpeechDetector(void)
Parameters	None
Description	This function is called to detect far-end speech, near-end speech and double-talk. After each detection the corresponding mode bit is set.

Table 8 : Prototype for AEC Update ERF Function

AEC Update ERF Function Prototype	
Syntax	Void UpdateERF(int *FEpwr, short out, short NE, short erf)
Parameters	*FEpwr: point to fe powers: only fes_short_pwr is used out: calculated output sample NE: new near-end sample ref: new normalized error times step size
Description	This function is used to calculate the new erf.L

8.3 - Convergence Properties

Note: Constants used first were short integers corresponding to floating point values used in floating point system. Results in term of TCL were quite identical than floating point results:TCL after 10 seconds for single talk was 28,42 dB and recover time after double-talk was 2-3 seconds.

Constants and thresholds have been modified to find bests results. The following results are obtained with those new values.

8.3.1 - Tests in Quiet Environment

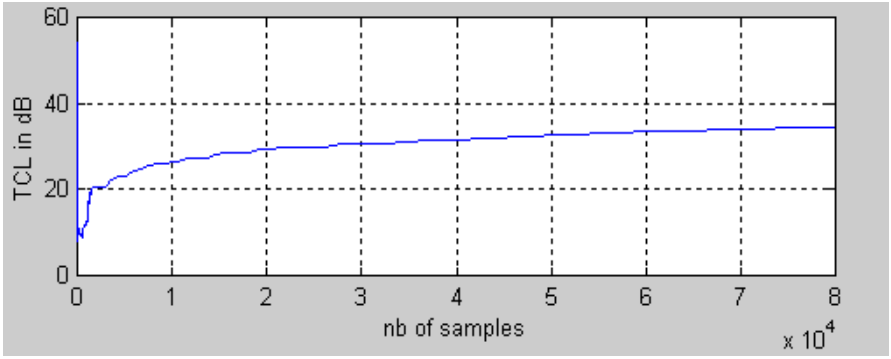
As specified by the ETSI, the measurements were taken after 10 seconds of speech to allow algorithm to reach a steady state.

The result is 34.48 dB.

The initial convergence time is also good. Indeed, the TCL nearly attains its maximum value after two seconds, like the floating point system result.

Following figure represents the TCL in dB versus the number of samples.

Figure 7 : TCL in dB versus the number of samples in Quiet Environment



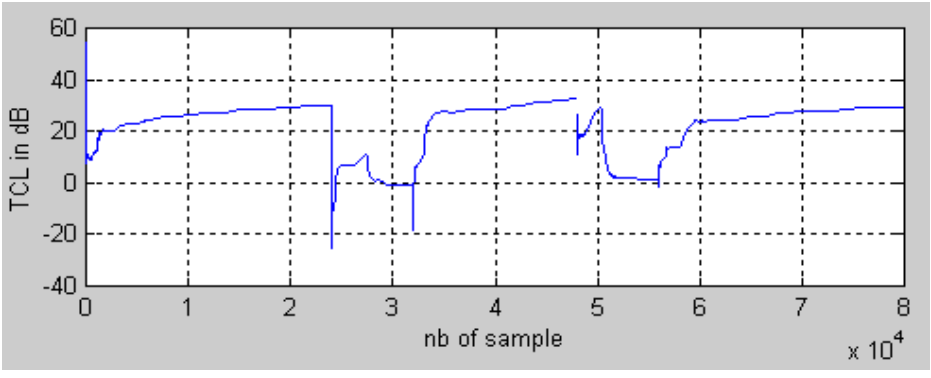
8.3.2 - Tests During Double-talk Conditions

The file used for the test was composed of a single talk period of 3 seconds, followed by a succession of double talk and near-end speech for 1 seconds, then the single talk condition returned for a period of 2 seconds. Double-talk then occurred during 1 second and single-talk then returned.

When double-talk occurred, the TCL computation was initialized.

Following figure shows that the filter is no more adapted at all to the signal during double-talk period. An excellent recovery time is realized after double-talk. Following double-talk, the TCL resumes its average value within 2-3 seconds.

Figure 8 : TCL in dB versus the number of cycles in Double-Talk Conditions



8.4 - Bench Results

In this paragraph the best results are presented. They are obtained with GHS-ST100-2.1-01 MULTI 2000 environment and LAO . GHS options are -OLAMI -Onounroll, as specified in chapter 6.

8.4.1 - Front End

This module contains all functions excepted Filtering: PowerEstimate, SpeechDetection, UpdateErf ... LAO options are -Ounroll=1.

It is important to remind that the call and the rts instructions are included in all MCPS results.

Since the number of cycle for each function is quite short, it is more appropriate to use the TraceViewer utility to have extremely right results.

The following results (in term of MCPS) are obtained by measuring the number of cycles from the decode step of the first instruction to the decode step of the poprts for each function.

Table 9 : Front End Bench Results

Function name	Nb of cycles/new sample	code size (bytes)
PowerEstimate	27	164
SpeechDetector	145	736
ControlLMS		124
CalcUpdate	118	104

Total number of cycles for front-end module is 27 + 145 + 118 = 290 cycles/sample.

It means that the maximum computational-time cost is 290* 8000 = 2.3 MCPS.

8.4.2 - NLMS Filtering

LAO options are:

– LAO1: -Ounroll=2

– LAO2: -Ounroll=2 -Osliw

Table 10 : Number of cycle of NLMS Filtering

	GHS+LAO1	GHS+LAO2
nb of cycle	1795	1286

Code size of the function is obtained with the "gfunsize" utility.

Table 11 : Code size of NLMS Filtering

	GHS+LAO1	GHS+LAO2
code size	216	332

Before explaining how to obtain the number of cycles, the filtering function behavior has to be reminded. After loading new data into data buffer, the function performs a multiplication and accumulation loop that updates the coefficients and shifts the entire data buffer down by one. It calculates each new coefficient value by multiplying the old data times erf. Next, it overwrites the old coefficient value with the new value. The updated coefficients then are used to calculate the filter output.

The C code is presented below:

```
void f_NLMSFiltering(
        short *psWork,
        short *s_erf,
        short *psOutSig
        )
{
        int             Temp;
        int             iInd;
        int             tempo;;

        for ( Temp = 0, iInd = 0; iInd < WIN_LEN; iInd ++ )
        {

         tempo = __mpfrch(psWork[iInd + 1], (*s_erf));
          sFilter[iInd] = sFilter[iInd] + (short)tempo;
          Temp = __mafcw ( Temp, sFilter[iInd],psWork[iInd]);
        }

        *psOutSig = (short)(Temp >> 16);

}
```

The main part is the loop of WIN_LEN (=256, 32ms at a frequency of 8kHz) iterations.

That's why MCPS-results are obtained with the command "cycle", called in the generated assembly before the beginning of the loop and at the end of the loop. The difference between both results gives the number of cycle to perform the entire loop (with 2-unrolling, it means 64 iterations). Consequently the numbers presented do not include neither call and rts instructions nor loop initialization.

Best case is 1286 * 8000 = 10.3 MCPS.

ASM generated is presented p. 20.

9 - IMPROVING PERFORMANCES

9.1 - First Step

GHS and LAO do not take into account packed arithmetic [2] that could be largely used in the NLMS Filtering module. It is necessary to re-write this function in assembly.

To take advantage of the sliw functionality of the ST120, an iteration of the nlmsFiltering loop processes 2 samples of the data buffer (instead of one). As a result, the operations described in paragraph 8.4 fit perfectly into 3 sliw groupings, which should lead to 3 cycles for two input data samples or 3/2 = 1.5 cycles for one input data sample. The **nlmsFiltering function should theoretically be performed in 1.5*N cycles**, N being the number of filter taps.

The nlmsFiltering function written in assembly [4] is presented below.

```
  .text
  f_NLMSFiltering:
    .align 8
    makec LC0 , 128
    setuls0 loop_start
    makea PR , %abs16to31(sFilter)
    setle0 loop_end-16
    addha P0 , P0 , 2
    morea PR , %abs0to15(sFilter)
    make R0 , 0
    make R15 , 0
    ldh R14 , @( P1 + 0 )
    .align 8
    .presliw
    sliwmd
  loop_start:
    ldf R12 , @( PR + 0 )
    ldw R1 , @( P0 !+ 4 )
    mafrcll R13, R12, R1 , R14
    nop

    ldh R2 , @( P0 - 6 )
    ldf SR, @( PR + 2 )
    mafrchl SR, SR, R1 , R14
    nop

    mafchl R0 , R0 , R13 , R2
    mafchl R15, R15, SR, R1
    sdf @(PR !+ 2), R13
    sdf @(PR !+ 2), SR
  loop_end:
    nop
    nop
    nop
    gp32md
    addcw R0 , R0 , R15
    subha P0 , P0 , 2
    sdf @( P2 + 0 ) , R0
    rts
    .leave f_NLMSFiltering
```

In order to check our 1.5-cycle loop, a breakpoint is set after the beginning of the loop, and another after the end of the loop. The command "cycle" is called after each breakpoint and the difference of both results gives us the number of cycles needed to perform the whole loop.

For 32 ms filter length (number of taps equals to N = 256), the difference is 521 cycles for 128 iterations (N/2). It means that an iteration needs in reality 4 cycles (521/128 = 4,03) so that the processing for one input data sample takes 2 cycles. **It makes the nlmsFIltering function needs 2 * N cycles**.

Table 12 : nlmsFIltering function Performance

Function	Nb Cycles/Sample	Code Size
nlmsFiltering	535	192

Filtering function takes 535*8000 = 4. 1 MCPS.

9.2 - Second Step

In this paragraph, a new way of writing the function is developed in order to get a real 1.5N cycle loop.

Let's remind that the output result is the addition of intermediate terms sFilter[i]*psWork[i]. The output is computed in the same time than the sFilter[i]. The first iteration of the loop is the following (Temp is initialized to zero):

$sFilter[0] = sFilter[0] + psWork[1] \times serf$

$Temp = Temp + sFilter[0] \times psWork[0]$ # Wait for the sFilter[0]

$sFilter[1] = sFilter[1] + psWork[2] \times serf$

$Temp = Temp + sFilter[1] \times psWork[1]$ # Wait for the sFilter[1]

To avoid latencies due to the two cycles needed for mac operations (cf #), the process is desynchronized. It means that sFilter[0] and sFilter[1] are computed outside the loop, Temp is set to sFilter[0]*psWork[0] and the first iteration of the loop performs the following operations: when sFilter[2] is computed, the second intermediate term of output is computed. When sFilter[3] is computed, the third intermediate term of output is computed, etc.

With such a system, sFilter[i] to be used during an iteration is ready when expected.

Initialisations

$sFilter[0] = sFilter[0] + psWork[1] \times serf$

$sFilter[1] = sFilter[1] + psWork[2] \times serf$

$Temp = Temp + sFilter[0] \times psWork[0]$

First iteration of the loop

$sFilter[2] = sFilter[2] + psWork[3] \times serf$

$Temp = Temp + sFilter[1] \times psWork[1]$ # sFilter[1] is ready

$sFilter[3] = sFilter[3] + psWork[4] \times serf$

$Temp = Temp + sFilter[2] \times psWork[2]$ # sFilter[2] is ready

At C level, the process is presented on pages 23 and 24

Like in the paragraph 9.1, an iteration of the loop fits perfectly into 3 sliw bundles but without any latencies, which should lead to a real 1.5N cycle loop. This time, the difference between the number of cycles returned by the "cycle" command called at the beginning and at the end of the loop equals to 390. It means that an iteration of the loop is performed in 390/128 = 3.04 cycles. **Then the processing for one input data sample really takes 1.5N.**

Table 13 : nimsFiltering Function Performance

Function	Nb Cycles/Sample	Code Size
nlmsFiltering	402	198

Filtering function is performed in 402*8000 = 3. 2 MCPS.

```
void f_NLMSFiltering(
short *psWork,
short *s_erf,
short *psOutSig
)
{

intTemp;
intiInd;
inttempo;

short adaptRate;
short *ptr = NULL;

short FiltPrec1 , FiltPrec2;
short ptrPrec1 , ptrPrec2;

ptr = &psWork[1];
adaptRate = *s_erf;

tempo = __mpfrch(psWork[1], (*s_erf));
sFilter[0] = sFilter[0] + (short)tempo;

tempo = __mpfrch(psWork[2], (*s_erf));
sFilter[1] = sFilter[1] + (short)tempo;

Temp = __mpfcw ( sFilter[0] , psWork[0]);

FiltPrec1 = sFilter[1];
FiltPrec2 = sFilter[2];
ptrPrec1 = ptr[0];
ptrPrec2 = ptr[1];
```

```
for (  iInd = 2; iInd < (WIN_LEN); iInd =iInd +2 )
{
    tempo = __mpfrch(ptr[iInd], adaptRate);
    sFilter[iInd] = FiltPrec2 + (short)tempo;
    Temp = __mafcw ( Temp, FiltPrec1, ptrPrec1);

    tempo = __mpfrch(ptr[iInd + 1], adaptRate);
    sFilter[iInd+1] = sFilter[iInd+1] + (short)tempo;
    Temp = __mafcw ( Temp, sFilter[iInd], ptrPrec2 );

    FiltPrec1 = sFilter[iInd+1];
    FiltPrec2 = sFilter[iInd+2];
    ptrPrec1 = ptr[iInd];
    ptrPrec2 = ptr[iInd+1];


}

Temp = __mafcw ( Temp, sFilter[WIN_LEN-1], psWork[WIN_LEN-1]);

*psOutSig = Temp >> 16;


}
```

The assembly code is presented below.

```
f_NLMSFiltering:
  push R4-R11
  .align 8
  makea PR , %abs16to31(sFilter)
  ldf R5 , @( P1 + 0)
  makec LC0 , 127
  ldf R6 , @( P0 + 2 )
  morea PR , %abs0to15(sFilter)
  ldf R7 , @( PR + 0 )
  ldf R8 , @( PR + 2 )
  mafrchh R7 , R7 , R6 , R5
  sdf @( PR + 0 ) , R7
  ldf R6 , @( P0 + 4 )
  addha P1 , P0 , 6
  setuls0 start_loop
  mafrchh R6 , R8 , R6 , R5
  setle0 end_loop-16
  sdf @( PR + 2 ) , R6
  ldf R9 , @( PR + 4 )
  ldf R10 , @( P0 + 0)
  addwa PR , PR , 4
  ldf R11 , @( P0 + 2 )
  mpfchh R2 , R7 , R10
  ldf R8 , @( P0 + 4 )
  movehl R11 , R8
```

Assembly code (continued)

```
.align 16
  .presliw
  sliwmd
start_loop:
  ldhsw R4 , @( P1 !+ 4 )
  nop
  nop
  mafrchh R1 , R9 , R4 , R5
  ldf R9 , @( PR + 4 )
  ldf R10 , @( PR + 2 )
  mafchh R2 , R2 , R6 , R11
  mafrclh R6 , R10 , R4 , R5
  mafchl R2 , R2 , R1 , R11
  extw R11 , R4
  sdf @( PR + 2 ) , R6
  sdf @( PR !+ 4 ) , R1
end_loop:
  ldh R0 , @( P0 + 510 )
  ldh SR , @( PR - 2 )
  mafcll R1 , R2 , SR , R0
  nop
  nop
  nop
  sdf @( P2 + 0 ) , R1
  gp32md
  poprts R4-R11
  .leave f_NLMSFiltering
```

10 - CONCLUSION

During single-talk period, the computational-time cost is 2.32 + 3.2 = 5.6 MCPS (front-end + nlms filtering); during double-talk periods, it is reduced to 2.32 MCPS (front-end). Single talk periods represent the main part of a conversation, whereas double-talk periods are present only during brief and limited instances.

11 - ANNEX

11.1 - ASM Generated by GHS+LAO

NLMSFiltering function

```
f_NLMSFiltering:
  .align 8
.LAO2:
  G15? makec LC0 , 128
  G15? makea PR , %abs16to31(sFilter)
  setuls0 .L55
  G15? addha P0 , P0 , 2)
  setle0 .LAO5-4
  G15? morea PR , %abs0to15(sFilter)
  G15? make R0 , 0
  .align 16
.L55:
  G15? ldh R2 , @( P1 + 0 )
  G15? ldh R1 , @( P0 + 0 )
  G15? mafcll R1 , SR , R1 , R2
  G15? ldh R2 , @( PR + 0 )
  G15? shrw R1 , R1 , 16
  G15? addu R1 , R2 , R1
  G15? sdh @( PR !+ 4 ) , R1
  G15? ldh R2 , @( P0 + 2 )
  G15? ldh R12 , @( P1 + 0 )
  G15? mafcll R12 , SR , R2 , R12
  G15? ldh R2 , @( P0 - 2 )
  G15? mafcll R2 , R0 , R1 , R2
  G15? ldh R0 , @( PR - 2 )
  G15? shrw R1 , R12 , 16
  G15? addu R0 , R0 , R1
  G15? sdh @( PR - 2 ) , R0
  G15? ldh R12 , @( P0 !+ 4 )
  G15? mafcll R0 , R2 , R0 , R12
  .align 8
.LAO5:
  .align 8
.LAO10:
.L56:
  G15? subha P0 , P0 , 2
  G15? sdf @( P2 + 0 ) , R0
  G15? make R0 , 0
  G15? rts
.LAO1:
  .leave f_NLMSFiltering
```

ACRONYMS AND DEFINITIONS

– **AEC** : Acoustic Echo Cancellation/Canceller
– **DT** : Double Talk
– **ERLE** : Echo Return Loss Enhancement
– **ETSI** : European Telecom Standard Institute
– **GHS** : GreenHills
– **LAI** : Linear Assembly Input language
– **LAO** : Linear Assembly Optimizer
– **MCPS** : Megacycle per second
– **NLMS** : Normalized Least Mean Squared
– **TCL** : Terminal Coupling Loss

REFERENCES

[1] ST120 DSP TOOL SET USER'S GUIDE available on www.st.com/st100/.
[2] ST120 DSP-MCU Core Reference Guide available on www.st.com/st100/.
[3] ST120 DSP-MCU Instruction Set Reference Guide available on www.st.com/st100/.
[4] ST120 DSP-MCU Programming Manual available on www.st.com/st100/.

littlediode
COMPONENTS FOR ALL

Littlediode supplies new, hard to find or obsolete electronic components and semiconductors all over the world.

With over two million different components listed you are sure to find the part you need.

Feel free to visit us today at our online store:

LittleDiode.com

Looking forward to providing you with the best possible service.