

Application Note 182

Getting Started with Porting Code to AudioDE

Document number: ARM DAI 182A

Issued: 2nd March, 2007

Copyright ARM Limited 2007



Application Note 182

Getting Started with Porting Code to the AudioDE

Copyright © 2007 ARM Limited. All rights reserved.

Release information

Change history

Date	Issue	Change
March 2007	A	First release

Proprietary notice

Words and logos marked with © and ™ are registered trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

<http://www.arm.com>

Table of Contents

1. Introduction	1-1
2. Acoustic Echo Cancellation.....	2-2
3. Least Mean Square Adaptive Filter	3-4
4. Overview of DEvelop LE.....	4-1
5. Implementation.....	5-2
5.1 System-Level Considerations	5-2
5.2 Code Porting.....	5-4
5.3 Dynamic Profiling.....	5-14
5.4 Why AudioDE does not Need Assembly	5-15
6. Summary.....	6-16
7. References.....	7-17

1. Introduction

Audio Data Engine (AudioDE) is a very low-power, synthesizable data engine core that can be programmed to support a wide-range of audio and speech codecs across all end-equipment targets. AudioDE has a number of functional units within its data path that efficiently execute audio algorithms. A combination of these audio-specific resources, general-purpose resources and bus infrastructure provides a data engine capable of targeting a diverse range of algorithms while being particularly efficient at audio processing.

Acoustic echo cancellation is one such algorithm that can be implemented on AudioDE to exploit the inherent advantages of AudioDE's architecture for audio processing. Acoustic echo cancellation is necessary in communication equipments such as wireless headsets, hands-free car kits, or mobile voice terminals.

By describing the implementation of an adaptive acoustic echo cancellation algorithm on AudioDE, this application note provides a "getting started" example of AudioDE code development flow as well as some important system considerations.

The application note comprises the following sections:

- Background on Acoustic Echo Cancellation
- Description of the Least Mean-Square Algorithm (LMS)
- Overview of the DEvelop LE compiler
- Implementation on the AudioDE.

2. Acoustic Echo Cancellation

Acoustic echo is caused by the existence of feedback paths between the far-end's speaker and microphone (Figure 1). Without acoustic echo cancellation, a person speaking into a microphone would hear an echo of his/her voice several milliseconds after having spoken. Because the feedback path is varying in time, an adaptive filtering method is required to estimate the echo, which can then be subtracted from the incoming speech signal.

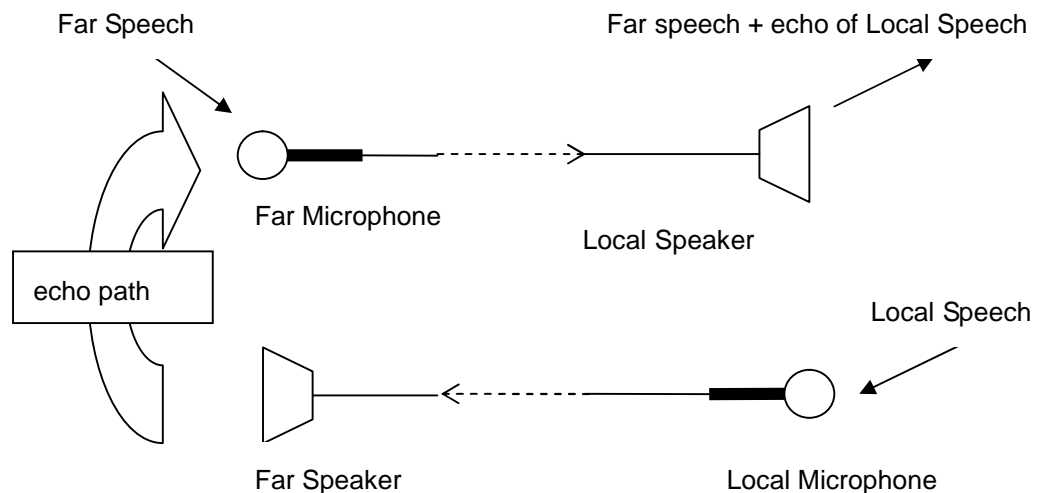


Figure 1. Simplified Speech Communication System with Acoustic Echo

There are multiple places where a feedback path can be set up. In a hands-free car phone system, the voice coming out of the far speaker can bounce off hard surfaces such as the windshield, side windows, and roof. In a wireless headset, the feedback path can be set up through the hard material holding the microphone and ear-piece together. In each application, the characteristics of the feedback path are different. One important characteristic of the feedback path is how much delay is introduced - a delay of greater than 16ms makes the echo detectable by the speaker; the larger the delay, the more noticeable the echo. The *tail length* of an echo represents the maximum delay length introduced by all echo paths and it dictates the tap size of the adaptive filter (time-domain processing) for effective echo cancellation. In a car environment, for example, a tail length of 32ms is typical. For speech sampled at 8 kHz, the tap size for the adaptive filter would be 256 (8000×0.032).

Today's digital communication systems pose an additional challenge to echo cancellation because voice coding schemes introduce additional delays in the order of 100ms to the echo path. A tail length of 270ms is a typical figure for such echo paths, which requires an adaptive filter with a tap size of 2048! With price pressures on wireless terminals, some of the echo cancellation burden is exported outside the handset.

More complex frequency-domain processing allows more efficient suppression of echo for large tap sizes. It also has the advantage of allowing better suppression of noise generated by non-linear systems (analog components). But for this application note example, we will stick with time-domain processing and illustrate how a Least Mean Square (LMS) adaptive filter can be implemented on AudioDE for acoustic echo cancellation.

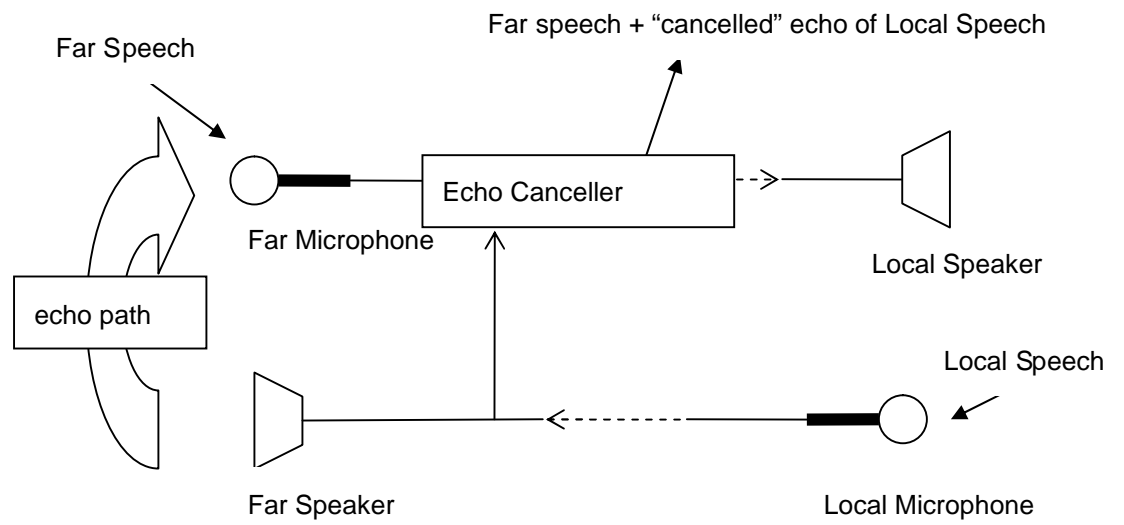


Figure 2 .Speech Communication System with Acoustic Echo Cancellation.

3. Least Mean Square Adaptive Filter

The least mean-square adaptive filter is an effective acoustic echo cancellation scheme that adapts to changing echo paths. An estimate of the echo to be cancelled (y) is computed by passing the input signal (x) through a standard finite impulse response (FIR) filter (W). This estimate is then compared to the actual echo heard from the far end (d) and an echo estimate error (e) is computed. This error is then used to update the FIR filter coefficients in order to reduce the error. (Figure 3) Conceptually, the adaptive filter is trying to form an image of the impulse response of an unknown system, H , that is varying in time. The estimated echo, y , is then subtracted from the incoming signal to suppress the noticeable echo.

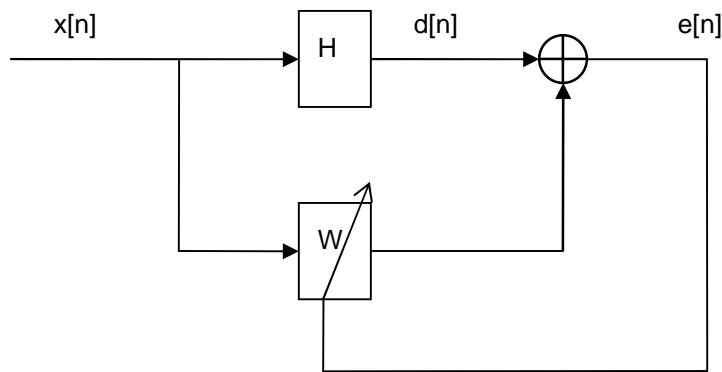


Figure 3. Block Diagram of an Adaptive Filter

Without illustrating the derivation, the equations involved in computing the new FIR coefficients of the adaptive filter W are:

$$y[n] = \sum_{k=0}^{N-1} w_k[n] \cdot x[n-k]$$

$$e[n] = d[n] - y[n]$$

$$w_k[n+1] = w_k[n] + \mu e[n] x[n-k]$$

where:

$y[n]$: estimate of echo at instant n

$w_k[n]$ is the k^{th} coefficient at instant n

$x[n]$: local input speech sample at instant n

$d[n]$: received speech sample (containing local speech echo) at instant n

$e[n]$: error on echo estimation

μ : adaptation rate

Note that in practice, the Normalised Least Mean Square Algorithm (NLMS) is commonly used instead of LMS. The NLMS is an extension of the LMS where the adaptation rate is scaled by the input signal energy (x^2).

4. Overview of DDevelop LE

DDevelop LE is the IDE (Integrated Development Environment) for data engines such as AudioDE. It is the “**L**ocked **E**dition” of the generic re-targetable compiler (**DDevelop**) whose main component is a powerful optimizing compiler that generates executable microcode as well as reports and graphical views at several compilation stages. The reports and views are used by the programmer to fix and fine-tune the code in order to achieve optimal performance.

The compilation of a program happens in three steps:

1. The Check step verifies the C/C++ syntax and converts the original code to an internal representation.
2. The Map step combines the internal representation of the source code with the AudioDE microarchitecture. It maps the individual operations from the code onto the available execution units, registers, and memories.
3. The Compile step performs three distinct tasks: it schedules operations; it assigns variables to registers and performs code compaction.

During each step, DDevelop LE generates feedback and intermediate results that enable the programmer to analyze the compilation process and tune it where necessary or desired. This feedback comes in the form of reports, views, and profiling data. Additionally, test benches are generated by DDevelop LE to allow convenient regression test on modified code.

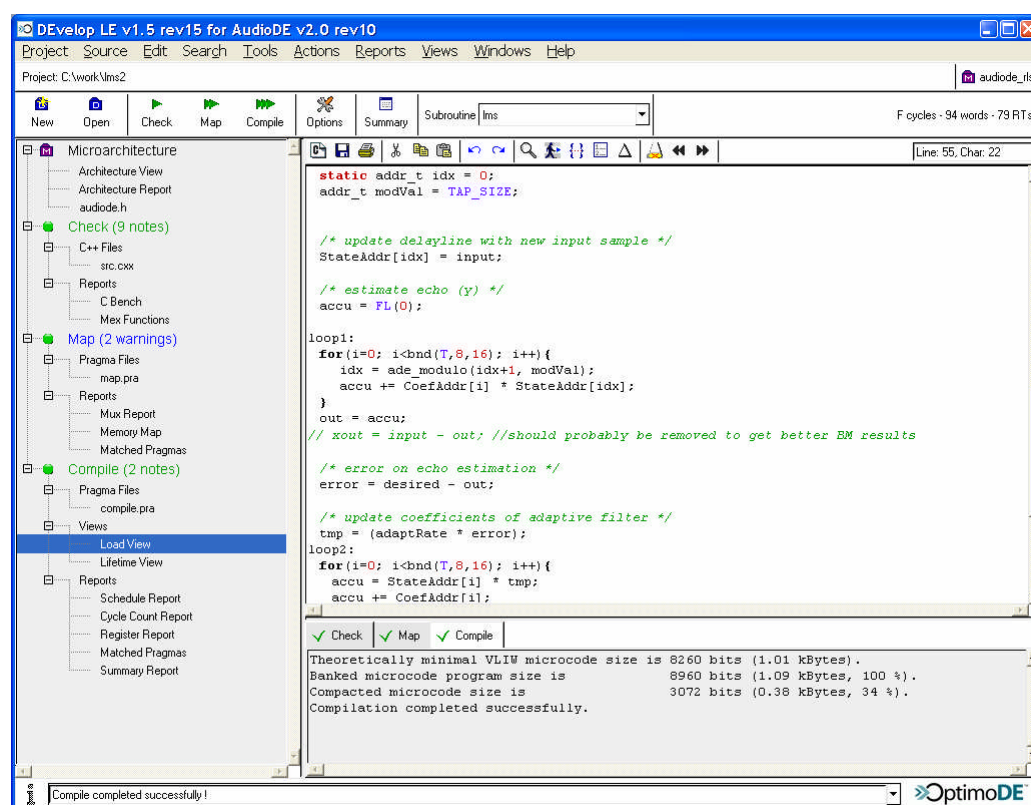


Figure 4. DDevelop LE for AudioDE Main Window View

5. Implementation

5.1 System-Level Considerations

Before we start porting the LMS algorithm to the AudioDE, let us first consider a few important aspects about the overall system implementation.

5.1.1 System Context – Data-Driven Model

The AudioDE is typically used as a sub-routine coprocessor to a Host processor. The Host processor is responsible for setting up task descriptions necessary for the initial launch of AudioDE execution. Once launched, AudioDE can run independently to the host processor and use the availability of new data to trigger its execution.

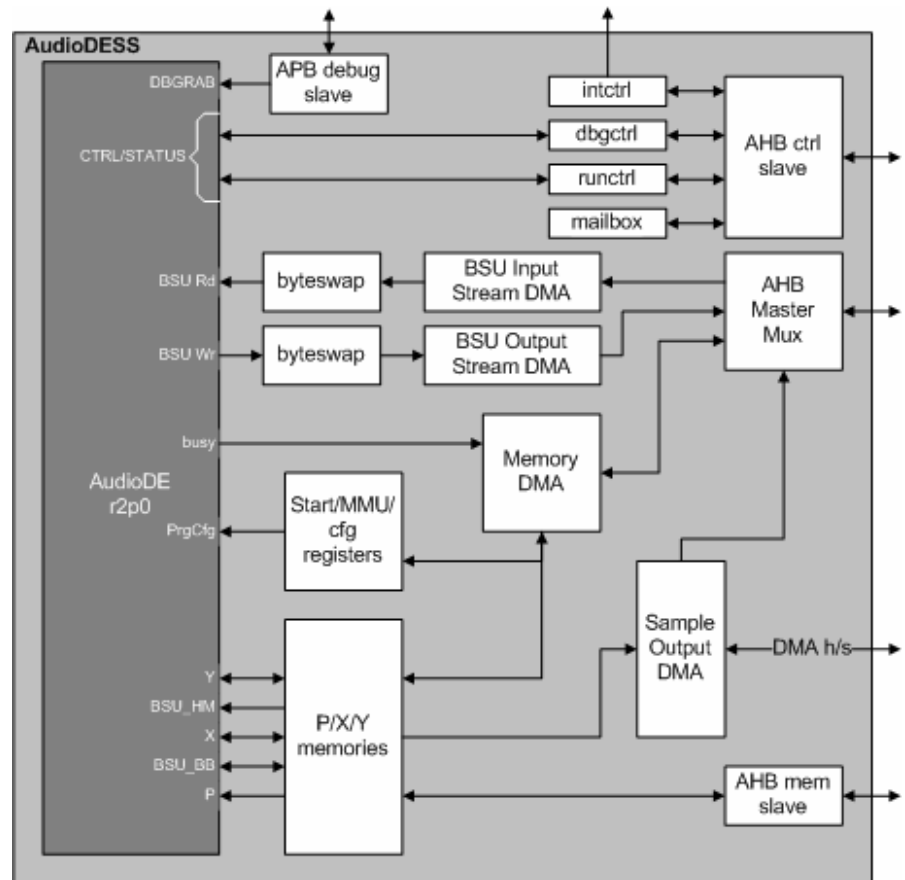


Figure 5. AudioDE and AudioDESS Block Diagram

Data input can be fed into the AudioDE's data memory space in several ways; either done by DMA (DMA is configured by Host or by AudioDE) or by direct data transfer from the Host.

The AudioDE Subsystem (AudioDESS) is intended to decouple all necessary external memory and I/O transfers from the actual processing on AudioDE. The Subsystem is equipped with five main interfaces:

1. AHB slave interface for accessing control registers by Host
2. AHB slave interface for accessing internal memories by Host
3. AHB master interface for internal DMA
4. Clock control interface for clock gating and frequency control
5. Hand-shake interface for miscellaneous IO (e.g. I²S)

It is important to note that AudioDE execution is not interruptible. As soon as a block of input data is available, AudioDE starts execution from the program start address. During task execution, a ‘busy’ signal informs AudioDESS that AudioDE is busy until task execution completes. When the task has completed, AudioDE is again ready to be triggered by availability of new data.

5.1.2 16-bit or 24-bit Mode?

Native data types in AudioDE are categorized in both *short* and *long* types. *short* types can be configured as either 16-bit or 24-bit wide (default 24-bits wide) and *long* types can be configured to be 32-bits or 48-bits wide (default 48-bits wide). Mode switching is done through a HW configuration pin in the AudioDE microarchitecture, which is controlled by the Host CPU. Microcode executing on AudioDE must have been generated by DEvelop LE with the correct corresponding mode set – by defining (or not) “ADE_16BIT” in the DEvelopLE Tool Options dialog box (Figure 6). By default, this MACRO definition is absent and the compiler assumes the target architecture is set to 24-bit mode.

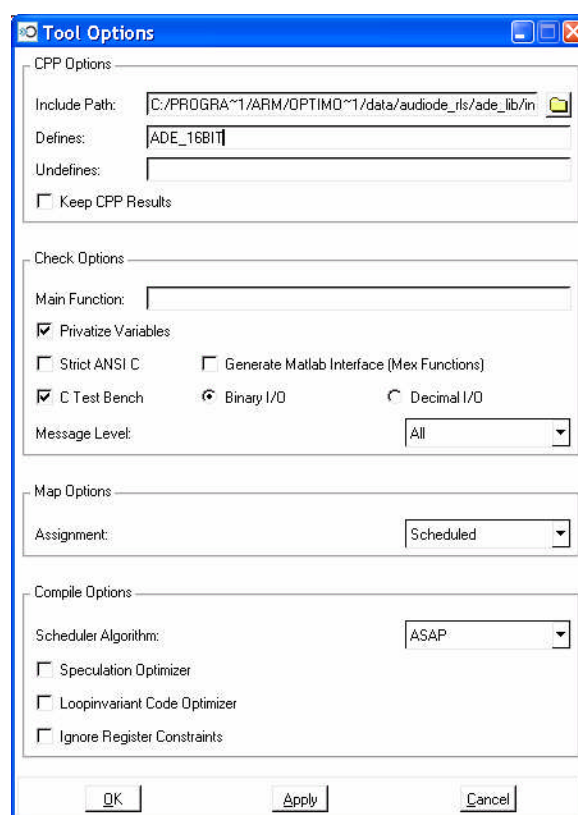


Figure 6. DEvelop LE Tool Options Dialog Box

The option to choose between 16-bit or 24-bit modes is a feature of AudioDE that facilitates the porting of code whose native short data type is 16-bit or 24-bit. A typical telephony/toll-quality voice application would rely heavily on 16-bit short data types, while an audio processing application would require 24-bit short data types to avoid double-precision operations. The echo cancellation algorithm described in this application note relies on 16-bit short data types so we must define the 16-bit mode for our implementation.

5.1.3 Inputs and Outputs

We shall assume that a new buffer of 16-bit sample input is delivered by DMA to a specific data location on AudioDE X or Y data memory. As soon as the new sample buffer is available, the

DMA signals a “start” to the AudioDE, which starts to execute the echo cancellation routine. The output “echo cancelled” sample would fill a circular buffer of samples, which would be encoded prior to transmission. In our example, we shall simply place the output “echo cancelled” speech samples in a specific data memory location.

5.2 Code Porting

Code development flow on AudioDE deviates somewhat from a “standard” DSP code development flow. The main difference lies in the absence of assembly instructions for AudioDE (see section 5.4). C/C++ code is compiled and intended operations are mapped to available resources (computation blocks, registers, and memory). These operations are then optimally scheduled to generate executable microcode. When inspecting “disassembled” microcode, the programmer sees a sequence of register transfers (RTs) rather than a sequence of assembly instructions. The RTs describe the operand(s) loaded into an execution unit, the output(s) from that unit, and optionally, a command associated with that unit.

Porting code to AudioDE involves two main phases (Figure 7). The goal of the first phase is to get the code to pass the three DEvelop LE compilation steps (*check*, *map*, and *compile*), and to check whether changes made to the code have altered its behaviour.

The *check* step involves making code compatible to the compiler by removing C/C++ features that are not supported. General code changes that help DEvelop LE generate more optimized result is also recommended. These recommended code changes are described in the following section.

The *map* step uses the data flow description generated by a successful *check* step as well as the AudioDE microarchitecture description - which is included in the DEvelop LE package - and attempts to map operations, constants, and variables to available resources on the AudioDE. This step may require replacing certain C operations with an equivalent API instruction. The output of a successful map step is a list of unscheduled operations, and memory maps.

The *compile* step optimally schedules operations generated during *map* step. Several compile options and pragmas exist in DEvelop LE that allows the programmer to influence the scheduling outcome.

Whenever code changes are made during this first phase, it is recommended to run C test bench in order to verify non-regression. The C test bench is automatically generated by DEvelop LE, and can be built and executed either on Linux or Windows.

In the second phase, reports and views generated by Develop LE provide static profiling information of the code, allowing identification of critical portions of code that need to be optimized. In addition to recommended code changes that help DEvelop LE generate more efficient code (see next section), DEvelop LE also allows the programmer to influence compilation outcome through the use of compile options and pragmas. Some of these options and pragmas are used in the echo cancellation example illustrated later on.

Again, it is highly recommended that changes to the reference (original) code be frequently tested for non-regression using the C test bench.

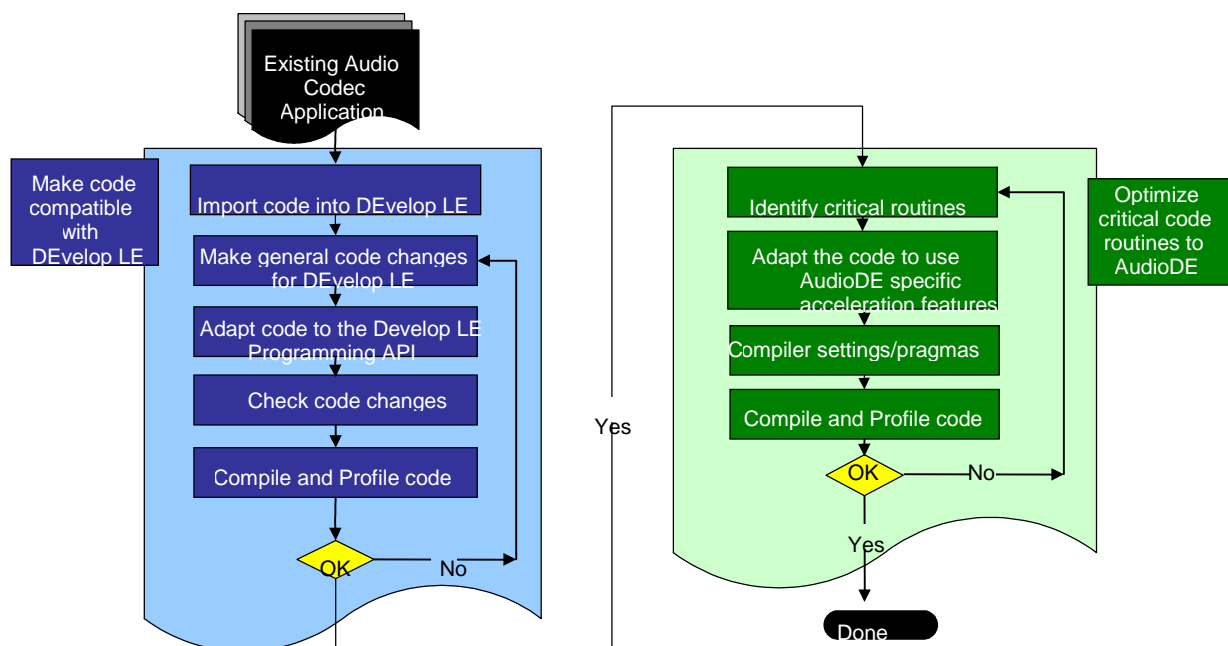


Figure 7. AudioDE Code Development Flow

5.2.1 Phase I: Making the Code Compatible with DDevelop LE

Have a Golden Reference

First, it is necessary that the reference C code has been white-box tested on a standard Host (Windows or Linux). We need a “golden” reference to which regression tests to ported code can be compared.

Create a Single Source File

Next, the code must be organised into a single source file (src.cxx). DDevelop LE analyses the entire data flow of an application to generate an optimal implementation. It requires full visibility of the whole code, which means external functions are not allowed.

Condition-out Code that is Not Part of Final Implementation

Code that should not be part of the AudioDE implementation, such as debug code, or a test harness, must not be compiled by DDevelop LE because it will prevent the compiler from properly analysing the real (implementation) data flow, leading to a less-than optimal implementation. The definition `__OPTIMODE__` is by default defined in DDevelop LE, and can be used to conditionally ignore such code.

Qualify Inputs and Outputs to All Functions

Inputs to and outputs from all functions need to be qualified as either inputs, outputs, or neither inputs nor outputs. Inputs are qualified using the `const` qualifier or the `IN` pragma; outputs using the `OUT` pragma; and the `IGNORE` pragma is used when a member of a class is neither an input nor an output. Using these pragmas enables DDevelop LE to perform better optimization of the code; however, the compiler has no way of verifying the qualifications made, so care must be taken when qualifying inputs and outputs.

Fully Describe Arrays

Arrays need to be defined such that their sizes are deterministic. DEvelop LE does not support arrays with incomplete type descriptions.

Convert Pointers to Arrays

And finally, the compiler treats pointers as arrays with predetermined size and location; as such, it is good practice to replace pointers by arrays.

Make Use of Different Cluster Types

Processing of AudioDE-specific type variables are more efficiently done in their respective clusters. Data types should be declared as `shortw_t`, `fshortw_t`, `longw_t`, `flongw_t`, etc. and address types should be declared as `iter_t` or `addr_t`. Beware that standard C types (`short`, `int`, `long`, `long long`) are interpreted differently (different data widths) by DEvelop LE from most other compilers. (More details can be found in the AudioDE Programmer's Guide).

Avoid Structs if Possible

It is common that pointers to structures are used in C as arguments for function calls. Because pointer handling is not supported, the compiler will pass all members of a structure as scalars during such function call even if only a few of its members are used by the called function. It is good practice to avoid passing pointer to structures and instead enumerate each member of the structure required.

5.2.2 Phase II: Optimize Code for AudioDE

AudioDE API Instructions

Several AudioDE API instructions are available to achieve better utilisation of microarchitecture resources. These resources include bit-stream access operations, Huffman table lookup operations, and various other Audio Functional Unit operations. Examples of these API instructions include: 'ade_getBits' and 'ade_decodeHuffWord'. Our simple echo cancellation algorithm does not contain any specific functions that can be replaced by AudioDE API instructions.

C Labels

C labels enable easy identification of a scope, which speeds up optimization by the compiler when compile pragmas are defined. It is good practice to place labels at beginnings of code blocks (e.g. for loop blocks).

Beware of Big Initialization Code

The compiler generates initialization code for static and global variables that are defined with C types, while those with SystemC (or AudioDE API Types) are not initialized. Code size can sometimes be dramatically reduced if the following best practices are followed:

- Avoid the use of global, static variables.
- Use AudioDE API types for variable definitions. API types do not require zero initialization. Only explicitly initialize those variables that need to be initialized.
- Avoid loading of immediate values to initialize a local variable array by storing the values in a const array and using an explicit initializing loop to load the initial values.

Make Efficient Loop Codes

It is recommended to have, where possible, bounded, well-defined loops. Unbounded loops, or loops with jump statements (`break`, `continue`, etc.) create scheduling barriers that prevent the compiler from optimally scheduling operations. *while* loops are also to be avoided as they require an extra if-state surrounding the loop.

In Figure 8 the ('Golden') reference C code for the adaptive echo canceller is presented with descriptions of needed changes. Figure 9 shows the resulting modified code that is now compatible with DEvelop LE.

```

/* Reference Acoustic Echo Canceller code */
#define TAP_SIZE 256
void lms(
    short x, //input sample (Q15 fractional format)
    short d, //received echo sample (Q15 fractional format)
    short k, //adaptation rate (Q15 fractional format)
    short xout) // echo cancelled sample
{
    static short coeff[];
    static short delayline[];
    short i;
    int accu;
    short y; //estimated echo (Q15 fractional format)
    short e; //error on estimated echo (Q15 fractional format)
    short tmp;
    short *coeffPtr;
    short *delaylinePtr;

    /* update delayline with new input sample */
    for(i=TAP_SIZE-1; i>0; i--){
        delayline[i] = delayline[i-1];
    }
    delayline[0] = x;

    /* estimate echo (y) */
    coeffPtr = &coeff[0];
    delaylinePtr = &delayline[0];
    accu = 0;
    for(i=0; i<TAP_SIZE; i++){
        accu += *coeffPtr++ * *delaylinePtr++;
    }
    y = accu>>16;
    xout = d - y;
    /* error on echo estimation */
    e = d - y;

    /* update coefficients of adaptive filter */
    coeffPtr = &coeff[0];
    delaylinePtr = &delayline[0];
    tmp = (k * e)>>16;
    for(i=0; i<TAP_SIZE; i++){
        accu = *delaylinePtr++ * tmp;
        accu >>= 16;
        accu += *coeffPtr;
        *coeffPtr++ = accu;
    }
}

```

qualify I/O's

fully define arrays

use modulo addressing to avoid updating every element of delayline

replace pointers by arrays

rely on automatic HW casting rather than explicitly shifting bits.

replace pointers by arrays

Figure 8. Reference C Code for Least-Mean Square Adaptive Echo Canceller

```

#define TAP_SIZE 256
void lms(
    const fshortw_t x, //input sample (Q15 fractional format)
    const fshortw_t d, //received echo sample (Q15 fractional format)
    const fshortw_t k, //adaptation rate (Q15 fractional format)
    fshortw_t& xout) //output echo cancelled sample (Q15 fractional format)
{
#ifdef __OPTIMODE__
#pragma OUT xout
#endif
    static fshortw_t coeff[TAP_SIZE];
    static fshortw_t delayline[TAP_SIZE];
    static addr_t idx = 0;
    iter_t i;
    flongw_t accu;
    fshortw_t y; //estimated echo (Q15 fractional format)
    fshortw_t e; //error on estimated echo (Q15 fractional format)
    fshortw_t tmp, tmp1;
    addr_t modVal = TAP_SIZE;

    /* update delayline with new input sample */
    delayline[idx] = x;

    /* estimate echo (y) */
    accu = FL(0);
loop1:
    for(i=0; i<TAP_SIZE; i++){
        idx = ade_modulo(idx+1, modVal);
        accu += coeff[i] * delayline[idx];
    }
    xout = accu.rndPinf() - d;

    /* error on echo estimation */
    e = d - accu.rndPinf();

    /* update coefficients of adaptive filter */
    tmp = (k * e);
loop2:
    for(i=0; i<TAP_SIZE; i++){
        accu = delayline[i] * tmp;
        accu += coeff[i];
        coeff[i] = accu;
    }
}

```

use `__OPTIMODE__` to compile DEvelop LE-specific code

C labels help DEvelop LE identify code portions where pragmas are applied

Figure 9. Modified Code Compatible with DEvelopLE and AudioDE

The various reports generated by the compiler can be used to help you identify areas of the code that can be optimized. One of these reports, the “Load View”, illustrates how resources on the AudioDE are utilized during each clock cycle. The “Load View” explicitly tells us which execution unit, which register, and which port is being utilized during a cycle.

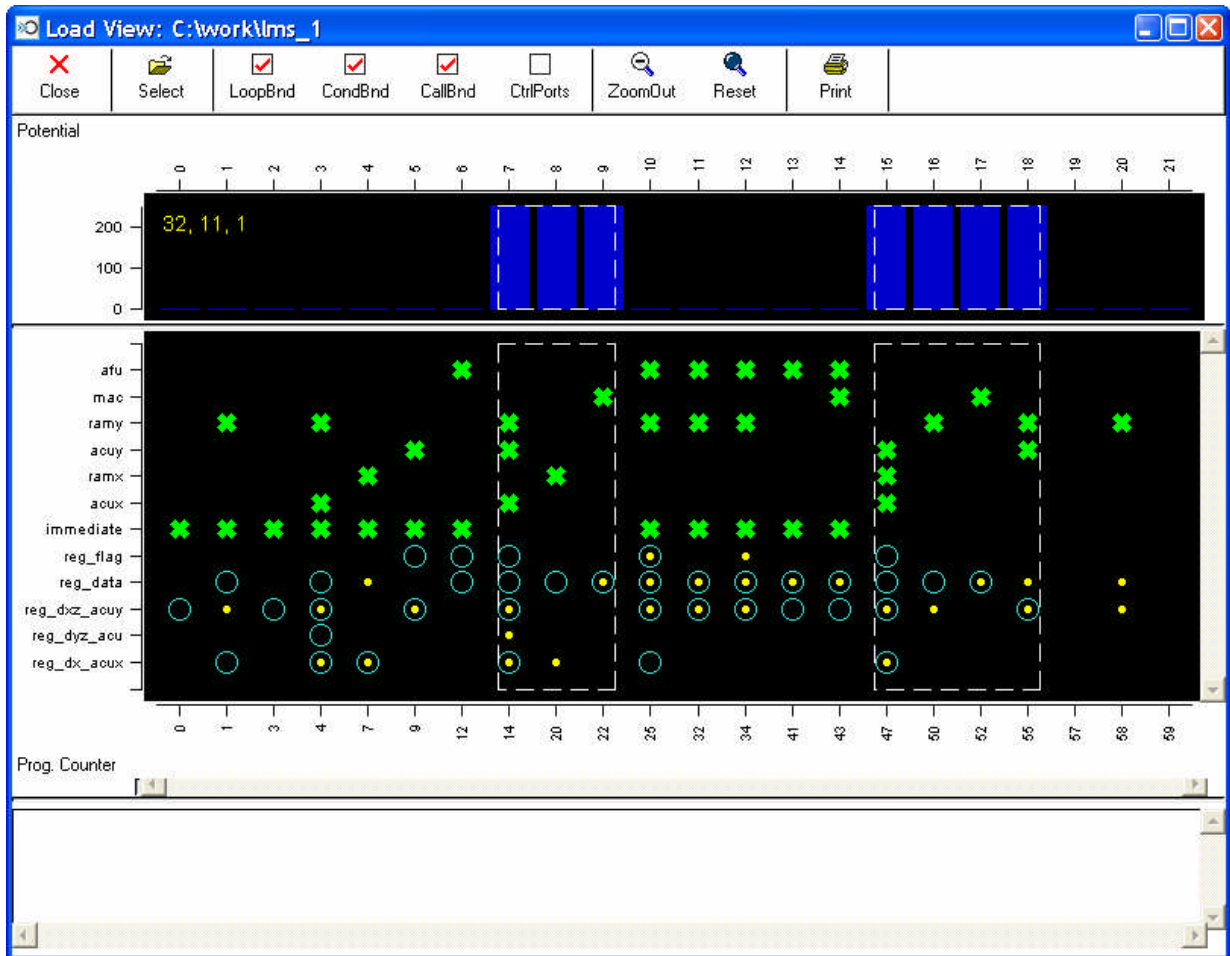


Figure 10. Load View before loop folding

The Load View also identifies loops in our code and outlines them with dashed lines. We can identify the first loop in our code requiring three cycles per iteration (see left-most dashed rectangle of Load View in Figure 10), while the second loop requires four cycles per iteration.

A powerful speed optimization technique is loop folding (aka software pipelining). Loop folding tries to reduce cycle count by executing during the same cycle operations that are normally executed during consecutive cycles. Loop folding is achieved by removing immediate data dependencies and peeling some operations out to the loop prologue and/or epilogue. Loop folding can significantly reduce cycle count; however, it typically increases code size.

By default, DDevelop LE optimizes for code size, thus it does not fold loops. The compile pragma *fold* is used to direct the compiler to fold labeled loops (see Figure 11). Note that the compiler allows usage of wild cards (*), which can be useful when setting pragmas for loops belonging to the same group (e.g. inner loops only). Another compile pragma, *additional_folding_feedback*, directs DDevelop LE to provide further feedback on what factors are limiting the extent to which the loop can be folded.

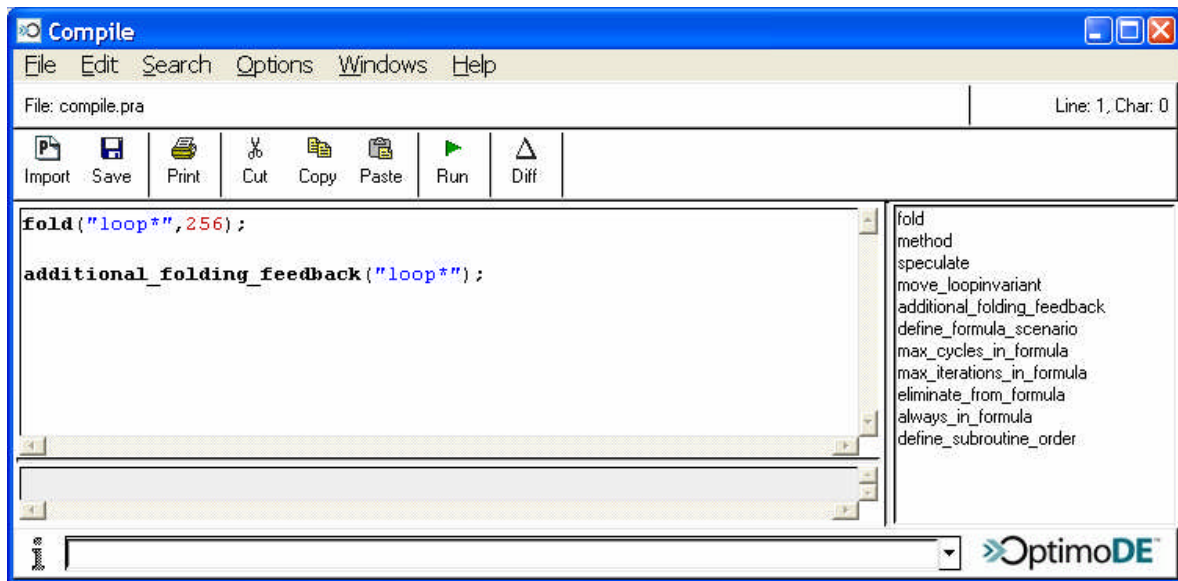


Figure 11. Compile Pragma Window

Another important consideration in MAC-type loops is the separation of multiply operands in ram X and ram Y in order to achieve single-cycle multiply-accumulate instructions. By default, all data is placed in ram Y and both of our operands in *loop1* (*delayline[]* and *coeff[]*) would normally be mapped to ram Y. By using the map pragmas, we can first define a memory sector in ram X, and then assign the variable *delayline* to this sector. (See Figure 12)

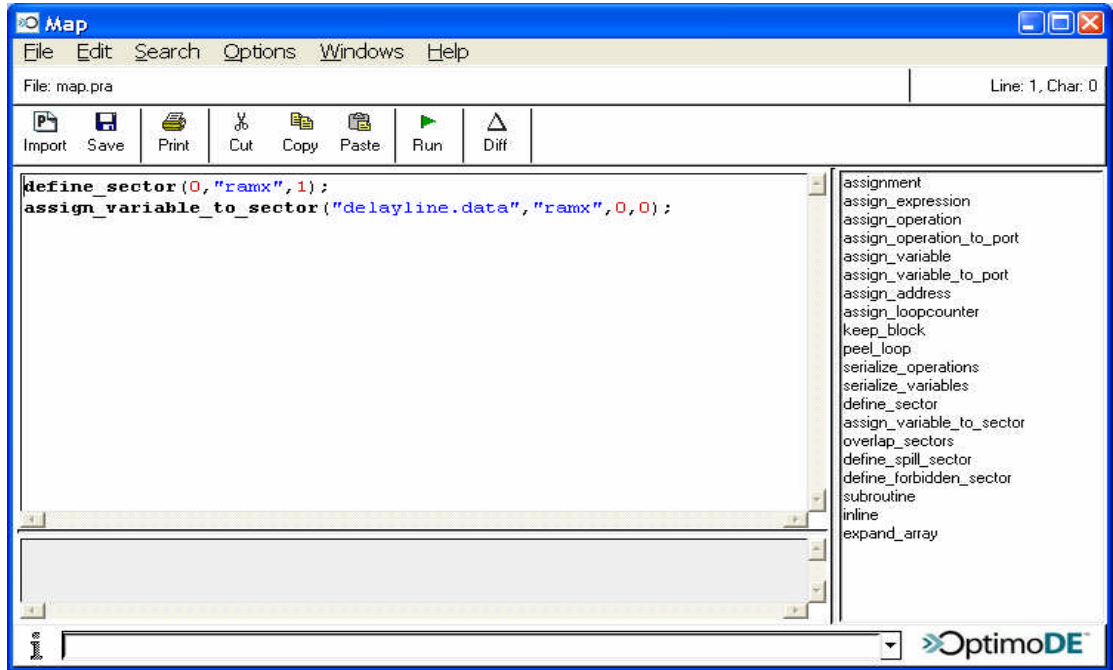


Figure 12. Map Pragma Window

After folding, the resulting load view shows that we have reduced cycles per iteration of both our loops by one. (See Figure 13)

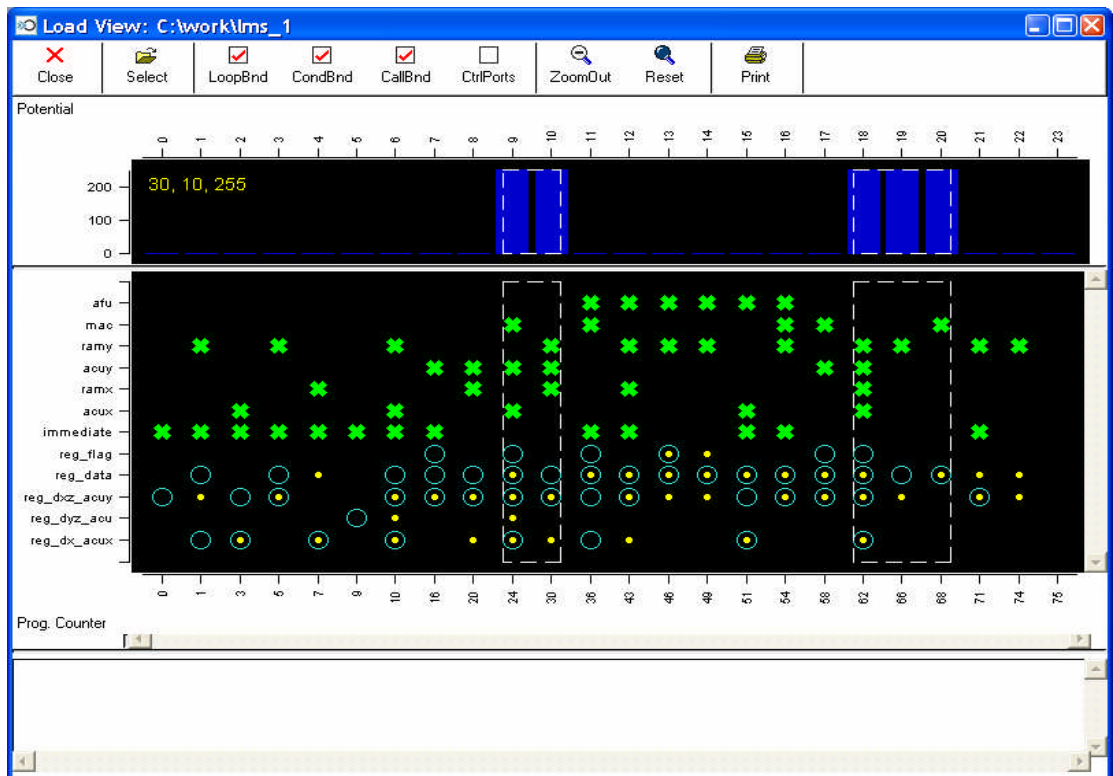


Figure 13. Load View after loop folding

At this stage of our optimization process, we have arrived at an implementation that executes to completion in 1294 cycles (see cycle count report in Figure 14); and requires a compacted microcode size of 0.32 Kbytes. (See Figure 15)

The screenshot shows a window titled "Cycle Count Report: C:\work\lms_1\ode_dev_cache\formula.tcl". The window contains a report with the following details:

```
// Report      : FORMULA
// Tool       : DEvelop LE
// Version    : v1.5 rev15
// Date      : Mon Jan 15 09:58:29 2007
// Project   : C:\work\lms_1
// Main Function : lms
```

The main function lms

☐ default scenario:

1294

☐ STRUCTURE scenario:

pot	pc	operation description
0		FUNCTION lms {
9		FOR (i = 254; i >= 0; i += -1) {
		// loop1, flag = flag(i<=0), /lms/loop1
		potentials : 2, cycles : 2, weight : 255
		}
18		FOR (i = 254; i >= 0; i += -1) {
		// loop2, flag = flag(i<=0), /lms/loop2
		potentials : 3, cycles : 3, weight : 255
		}
		}

Figure 14. Cycle Count Report

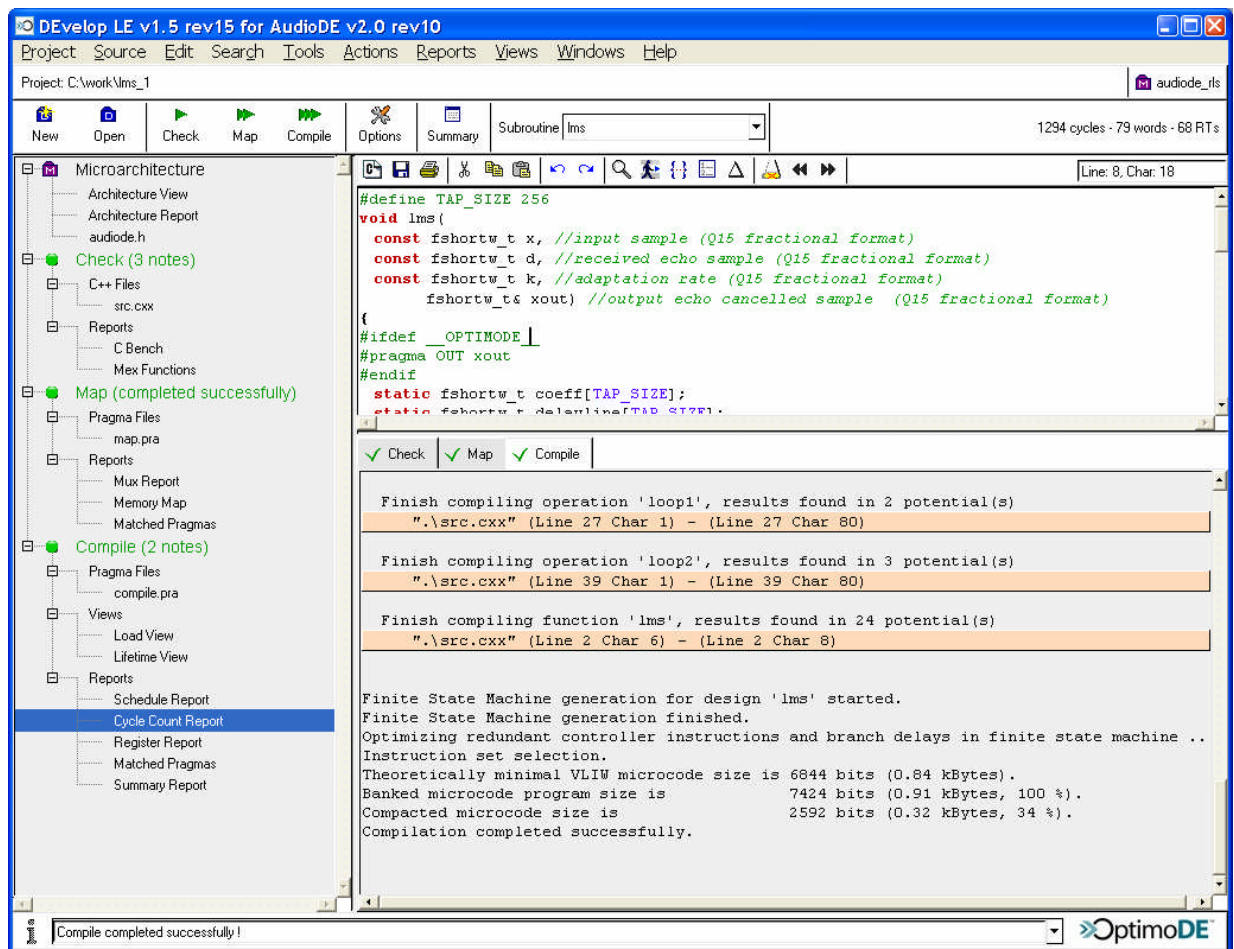


Figure 15. Compilation Report after loop folding

5.3 Dynamic Profiling

We've seen that Develop LE not only compiles your code but also provides a static profile of the application through views and reports. These views and reports provide feedback on where and how the programmer needs to adjust code and compile options to best optimize the code. When implementing large applications, it is also necessary to exercise the application with realistic stimuli to obtain real-time cycle count information.

The Instruction Set Simulator (ISS) model of the AudioDE and a corresponding ISS bench program are delivered as part of the AudioDE release. The ISS bench is a C++ program that is compiled and linked with your application code on a host platform (Windows XP or Linux). The resulting executable uses the ISS model to generate output data that can be compared to a reference output. It also generates a dynamic profile of the simulation run, which contains cycle count information for the whole test run as well as for each frame and sub-routine. It also provides statistics on executed processing units for each subroutine. (See Figure 16).

TOTAL SUMMARY						
=====						
Total cycles: 125919043						
Total coverage: 78.3 %						

Number of subroutine calls						

	Cycles in subroutine					
		Cycles in subroutine and called subroutines				
			Subroutine call site			
=====						
1	19364	19364	init			
152	40513384	125899679	AACPD_SessionHandler			
4864	10618112	10618112	AACPD_SessionHandler/DCTIV			
152	9120	9120	AACPD_SessionHandler/sbr_SqrtRatio			
2432	167808	167808	AACPD_SessionHandler/ps_HybridSynthesis20			
4864	12495616	33731840	AACPD_SessionHandler/AACD_SBR_synFilterbank2			
4864	10618112	10618112	AACPD_SessionHandler/AACD_SBR_synFilterbank2/DCTIV			

SUBROUTINE DCTIV						
pc	init	total	min/	avg/	max	
=====						
11408	0	24486	161/	161.1/	168	
11410	0	24486	161/	161.1/	168	

init	min/	avg/	max	% on DPU	% of cycles	instruction
=====						
0	34304/	35819.8/	35840	35.2	9.1	afu.cmd0.au_add
0	56/	115.2/	116	0.1	0.0	afu.cmd0.au_negx
0	28211/	29140.2/	29268	28.6	7.4	afu.cmd0.au_passx
0	34465/	35980.9/	36008	35.3	9.1	afu.cmd0.au_sub
0	322/	322.2/	336	0.3	0.1	afu.cmd0.lu_norm
0	483/	483.3/	504	0.5	0.1	afu.cmd1.fcu_passx
0	442/	556.7/	572	0.5	0.1	afu.cmd1.imm_00
0	12717/	12717.9/	12784	12.5	3.2	afu.cmd1.imm_01
0	165/	165.1/	176	0.2	0.0	afu.cmd1.imm_02
0	97358/	101378.2/	101568	99.5	25.7	afu.cmd0
0	13807/	13923.0/	14036	13.7	3.5	afu.cmd1
0	97841/	101861.5/	102072	100.0	25.8	afu.mode
0	20626/	21370.3/	21394	22.1	5.4	mac.maccmd.maddrs
0	38114/	39602.3/	39636	40.9	10.0	mac.maccmd.mltdp
0	26258/	27002.3/	27026	27.9	6.8	mac.maccmd.mrsubrs
0	8770/	8770.2/	8784	9.1	2.2	mac.maccmd.nmltdp
0	93768/	96745.1/	96840	100.0	24.5	mac.maccmd

Figure 16. Example dynamic profile report

5.4 Why AudioDE does not Need Assembly

Not having assembly instructions on AudioDE does not diminish the level of optimization attainable on the AudioDE, nor does it increase the effort of porting code to the AudioDE. On the contrary, the DEvelop LE compiler can optimize C/C++ code to achieve microcode that is at least as good in performance than optimized hand-written assembly code for a standard single-MAC 16-bit DSP.

To illustrate this, we compare simple code examples for sample FIR filter on AudioDE and a generic single-MAC DSP. Both are able to execute the FIR tap in a single cycle. (see Figure 17 and Figure 18)

```
SET <IMMEDIATE>, ModeRegister ;//configure modulo addressing mode
MULT [pointerXreg]+, [pointerYreg]+, accumulator_0
LOOP N
    MAC [pointerXreg]+, [pointerYreg]+, accumulator_0
    ADD product, accumulator_0
```

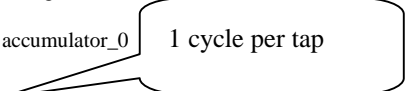


Figure 17. Sample FIR code example on Generic Single-MAC DSP

```
modval = TAP_SIZE;
idx = 0;
accumulator = 0;
fir_loop: for(i=0;i<N;i++){
    idx = ade_modulo(idx+1,modval); //uses ade_modulo API instruction
    accumulator += data[idx] * coeff[i];
}
```

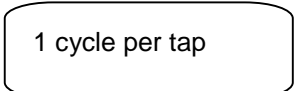


Figure 18. Sample FIR code example on AudioDE

Having a Very Long Instruction Word (VLIW) architecture allows the AudioDE to perform significantly better than standard single-MAC DSPs in code benchmarks that are not MAC-limited, for example the Viterbi decode butterfly, or the FFT butterfly; both which can rely on the API instruction *ade_addsub* to achieve efficient adding and subtracting of two operands during the same cycle.

6. Summary

This “Getting Started” application note used a simple adaptive echo cancellation implementation to illustrate the general steps in porting code to the AudioDE. The code development flow encompasses two phases: First, the code must be made compatible with the DDevelop LE compiler. Second, the code needs to be refined for the AudioDE architecture by making use of AudioDE architecture-specific features. The optimization process during the second phase relies on a comprehensive set of reports and views generated by the compiler. Usage of pragmas and compiler settings help achieve optimal code performance. Furthermore, an ISS model of the AudioDE, as well as an ISS bench program are provided to allow dynamic profiling of the ported application.

Code porting to AudioDE does not involve writing assembly code. The optimizing DDevelop LE compiler is able to generate efficient microcode directly from AudioDE C code. The use of API instructions can help achieve performances that are often better than equivalent-class single-MAC DSP.

Although programming the AudioDE deviates somewhat from standard DSP programming flow, learning how to influence the DDevelop LE compiler through code adjustments, compiler settings, and pragmas will enable implementations on the AudioDE to be best-in-class in terms of cycle count and code size.

7. References

1. ARM AudioDE Programmer's Guide, revision v2.0 rev10. Document number: DE800-DC-02001-DUI0339
2. Jones, Douglas. Adaptive Filters. Connexions. 12 May 2005
<http://cnx.org/content/col10280/1.1/>