AVR336: ADPCM Decoder

Features

- **AVR Application Decodes ADPCM Signal in Real-Time**
- **Supports Bit Rates of 16, 24, 32 and 40 kbit/s**
- **More Than One Minute Playback Time on ATmega128 (at 16 kbit/s)**
- **Decoded Signal Played Using Timer/Counter in PWM Mode**

Introduction

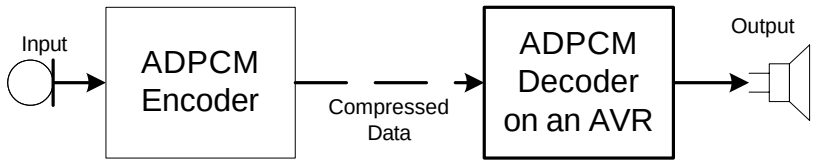
Adaptive Differential Pulse Code Modulation, or ADPCM, is a digital compression technique used mainly for speech compression in telecommunications. ADPCM is a waveform codec that can also be used to code other signals than speech, such as music or sound effects. ADPCM is simpler than advanced low bit-rate voice coding techniques and doesn't require as heavy calculations, which means encoding and decoding can be done in a relatively short time.

ADPCM is usually used to compress an 8 kHz, 8-bit signal, with an inherent flow rate of 64 Kbit/s. When encoded at the highest compression ratio, using only 2 bits to code the ADPCM signal, the flow rate is reduced to 16 Kbit/s, i.e. 25% of the original. Using 4-bit coding, the flow rate is 32 Kbit/s, i.e. 50% of the original, and the quality of the signal is fine for most applications.

This application note focuses on decoding the ADPCM signal. It uses the on-chip timer/counter to create a Pulse-Width Modulated (PWM) output signal, which is then passed through a simple, external filter. The filter consists of just a few external components and turns the digital signal into analog, suitable for connecting to speakers.

For further information about D/A conversion with Timer/Counter, please refer to Application Note *AVR335: Digital Sound Recorder with AVR ® and DataFlash ®*.

Figure 1. ADPCM decoding on an AVR microcontroller



8-bit AVR® Microcontrollers

Application Note

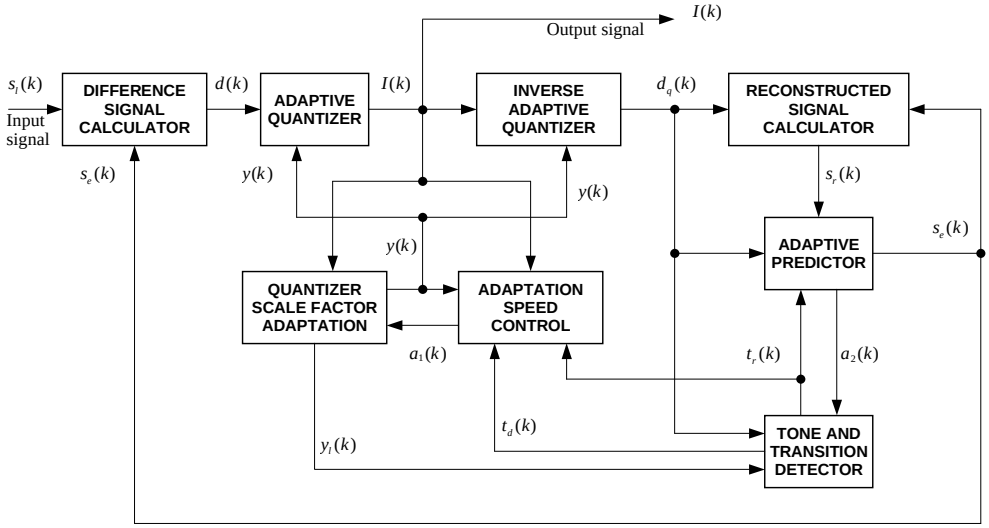
Rev. 2572A-AVR-11/04

1 Theory of Operation

The principle of ADPCM is to predict the current signal value from previous values and to transmit only the difference between the real and the predicted value. In plain Pulse-Code Modulation (PCM) the real or actual signal value would be transmitted. In ADPCM the difference between the predicted signal value and the actual signal value is usually quite small, which means it can be represented using fewer bits than the corresponding PCM value.

Depending on the desired quality and compression ratio, a difference signal is quantized using 4, 8, 16 or 32 levels. The block diagram of an ADPCM encoder is shown in Figure 1-1. For more information on how the encoder works see Reference 2.

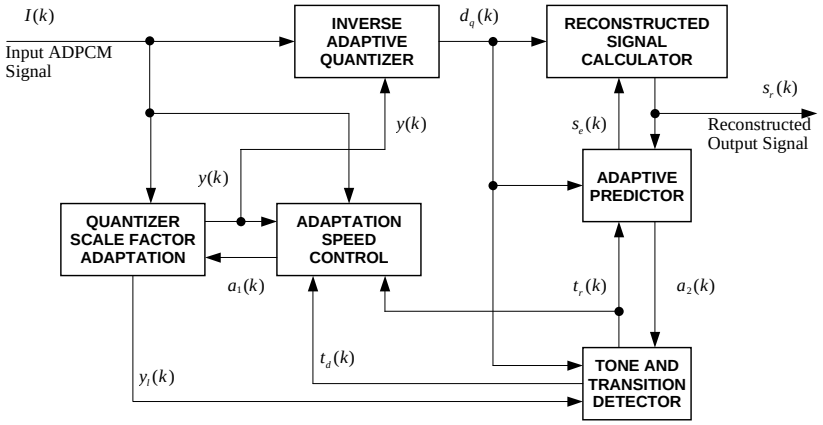
Figure 1-1. Block Diagram of ADPCM Encoder.



The decoder shown in Figure 1-2 takes the quantized value, performs an inverse quantization, and subtracts the result from the predicted signal to get the decoded signal.

The decoder is described in more details below. Headings refer to block names in Figure 1-2. For exact formulas, see “Appendix B: Formulas”.

Figure 1-2. Block Diagram of ADPCM Decoder.



1.1 Inverse Adaptive Quantizer

$$d_q(k) = 2^{d_{q\ln}(k)+y(k)}$$

This block calculates the linear quantized difference signal $d_q(k)$ from the logarithmic quantized difference signal $d_{q\ln}(k)$ and the adaptation factor $y(k)$. The logarithmic quantized difference signal $d_{q\ln}(k)$ is obtained from a static look-up table using the ADPCM codeword $I(k)$ as index. The linear domain difference signal is calculated by raising 2 to $d_{q\ln}(k) + y(k)$.

1.2 Quantizer Scale Factor Adaptation

The scale factor $y(k)$ used in the inverse adaptive quantizer is computed here. The scale factor $y(k)$ consists of two other factors; the fast (unlocked) scale factor $y_u(k)$ and the slow (locked) scale factor $y_l(k)$. Two scale factors are needed to handle different types of signals; a fast scale factor allows adaptation to signals with large fluctuations (e.g. speech), while a slow scale factor is needed when the signal changes slowly (e.g. tone signals). The scale factor is a linear combination of these two. The ratio is determined by speed control parameter a_l, described in the next section.

1.3 Adaptation Speed Control

The adaptation speed is controlled by the parameter a_l. It approaches unity with speech signals and zero with audio band data signals. To obtain a_l two measures of average magnitude (long- and short-term) of ADPCM codeword $I(k)$ are calculated. The difference between these two indicates how the average magnitude of $I(k)$ is changing. If the difference is small, the magnitude of $I(k)$ is constant; if the difference is large, the magnitude of $I(k)$ is changing, as indicated by the speed control parameter a_p. The parameter also takes into account the special cases where transitions are detected, or when the signal is idle. Finally, the limited speed control parameter a_l, is obtained from a_p (ranging from zero to two) by limiting it to range between zero and one. This is done asymmetrically to delay the transition from fast to slow adaptation mode.

1.4 Tone and Transition Detector

Tone and transition detections are included to improve the codec's response when handling data signals instead of speech.

If the signal uses only a narrow frequency band (e.g. tone signals), the quantizer is set to fast adaptation mode.

If transitions are detected, the quantizer is set to fast adaptation mode $(t_r = 1)$ and the coefficients of the adaptive predictor are set to zero.

1.5 Adaptive Predictor

$$s_e(k) = s_{ep} + s_{ez}$$

The adaptive predictor calculates the signal estimate $s_e(k)$ from the quantized difference signal $d_q(k)$. It uses two adaptive structures; a second-order structure that models the poles (s_{ep}), and a sixth-order structure that models the zeroes (s_{ez}). The coefficients in both structures are updated with a simplified gradient algorithm. In order to ensure stability the coefficients in the pole-modeling structure must be limited. The formulas for updating the coefficients in the zero-modeling structure are slightly different for different bit rates (5-bit vs. 2-, 3-, or 4-bit).

If a transition is detected, all predictor coefficients in both structures are set to zero (see previous block).

1.6 Reconstructed Signal Calculator

$$s_r(k) = s_e(k) + d_q(k)$$

The reconstructed signal $s_r(k)$ is calculated by simply adding the signal estimate $s_e(k)$ to the difference signal $d_q(k)$.

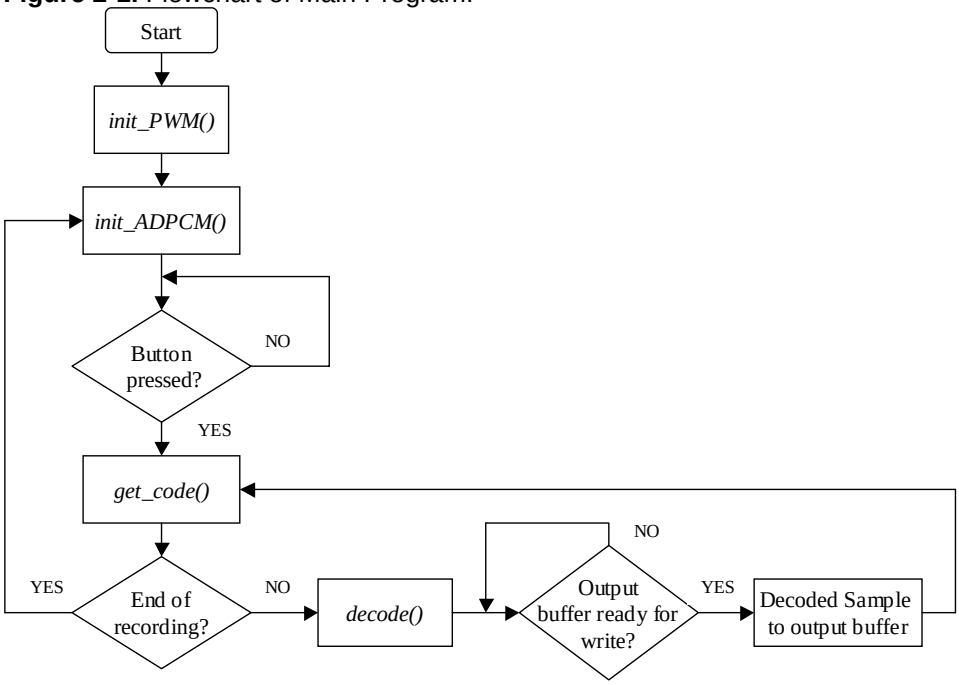
2 Implementation

This section describes the software implementation of the decoder.

2.1 Software

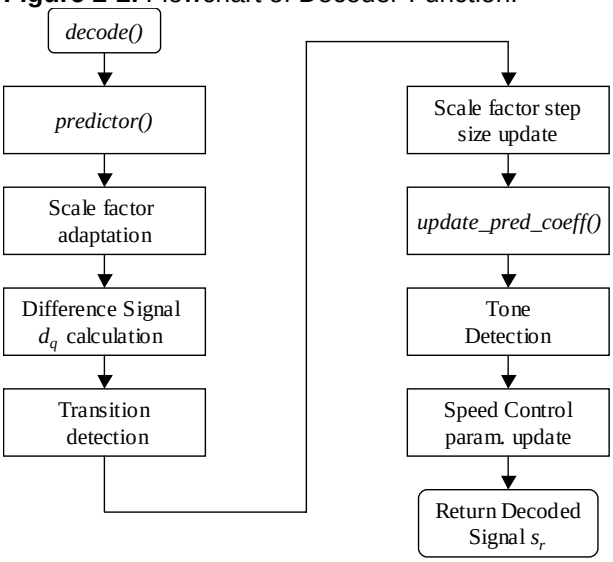
The decoder is implemented using IAR Systems C compiler, with some sections written in assembler. The main structure of the decoder shown in Figure 2-1 is simple; the program waits for a user to push a button, and then starts to decode pre-encoded samples from the Flash memory. It first calls *get_code()* which reads the data from the Flash and extracts the ADPCM codeword from it (one byte contains several codewords).

Figure 2-1. Flowchart of Main Program.



Next, the *decode()* –function is called, as shown in Figure 2-2. The function calculates the signal estimate (see *predictor())*, the quantizer scale factor *y*, and the difference signal *dq*. Possible transition signals are detected, and scale factor step size parameters and adaptive predictor coefficients are updated. Finally, the decoder detects possible tone signals and updates adaptation speed control parameters.

Figure 2-2. Flowchart of Decoder-Function.



The reconstructed signal *sr* is output through a simple D/A-converter, consisting of timer/counter1 in PWM mode (output to port B, pin 5) and an external filter.

2.1.1 Compliance with ITU-T G.726

The software follows quite closely what has been described above, but some changes have been made for more efficient computation. The ADPCM-standard ITU-T G.726 defines some calculations involving reconstructed signal $s_r(k)$ to be made with floating point arithmetic. These calculations are skipped in this implementation and are instead realized using fixed-point arithmetic. This has been done in order to save processor clock cycles. Using fixed point instead of floating point arithmetic could cause a small reduction in the quality of the reconstructed signal.

Two blocks have been left out completely, namely the μ-law and A-law conversion and synchronous coding adjustments. μ-law and A-law conversion are used to reduce the number of bits per sample. Since the output directly drives a D/A-stage, there's no need for μ-law/A-law -conversions.

The synchronous coding adjustment defined in the G.726 standard has also been skipped in this implementation. The purpose of the synchronous coding adjustment is to prevent distortion in synchronous tandem codings (e.g. ADPCM-PCM-ADPCM), which is not used in this application.

To achieve best possible quality, the encoding and decoding processes should use the same algorithms, i.e. the encoding should also be done with fixed-point arithmetic.

2.2 Hardware

To run the software an ATmega128 processor is used. To achieve better quality, it is highly recommended to use an active low-pass filter between the PWM output and the loudspeaker/headphone/amplifier to filter out the PWM base frequency. The PWM output and a suggested filter circuit has been described in Application Note AVR335. The schematics for a simpler, second order active filter implemented with a single-sided low-voltage operational amplifier is shown in Appendix C.

An STK500 development board with an STK501 expansion module is also recommended. When decoding signals with an 8kHz sample rate the AVR must run at 16 MHz and an external oscillator is required in such cases.

3 Implementation Example

This section gives step-by-step instructions on how to decode compressed ADPCM samples with an ATmega128 and the STK500/501 evaluation board. To decode an ADPCM signal with 8kHz sample rate the system clock frequency needs to be 16MHz.

3.1 Quick Start

To use the ADPCM-decoder:

- Copy one of the included encoded sound samples to the source directory and rename it to “data.c”.
- Start IAR Embedded Workbench, open the workspace file and configure the workspace; Edit the BITS and INPUTSIZE parameters in “adpcm.h” according to selected sound sample file (“data.c”) and rebuild the project.
- Download the compiled and linked program to the target AVR.
- Build and connect the output filter between the PWM output (PB5 on Atmega128) and the loudspeaker.

The signal is now ready to be decoded and played back.

3.2 ADPCM Sound Files

Some encoded sound samples are included with this application note. Copy one of the files to the source directory and rename it to “data.c”. Alternatively, use third-party software to create new ADPCM sound files. If using custom ADPCM data please note that the binary data must be copied into an array, as shown in the included sample files. See Appendix E on how to encode ADPCM files.

3.3 Compile & Link

To compile and link the source code in IAR Embedded Workbench:

- Open workspace “AVR336.eww”.
- Set output format to UBROF-8 (Project -> Options -> XLINK -> Output -> UBROF-8). IAR Embedded Workbench 3.10 generates UBROF 9 by default, but AVR Studio 4.10 currently supports UBROF 8, and previous. This may change as new releases are launched.
- In the project window, double-click on the file “adpcm.h” to open it. Update the ADPCM settings in the file to match the format of the ADPCM sound file. INPUTSIZE defines the length of the sound record and BITS defines the number of bits used for encoding.
- In the project window, left-click on the file “data.c” and select Compile from the pop-up menu. This is to make sure the compiler doesn’t use a precompiled version of an old sound file.
- Compile and link the project (Project -> Make).

3.4 Hardware Set-Up

Attach STK501 to STK500 and populate the top-module with an ATmega128. Connect push buttons to port D using a flat cable (connect the cable between SWITCHES and PORTD). Configure the target AVR to run at the desired frequency (default is using a 16 MHz external crystal). Connect the AVR and/or STK500 to a PC using direct cable connection or a choice of programmer/debugger.

3.5 Firmware Set-Up

Start AVR Studio and open the file “adpcm.dbg” from the IAR output directory (“Debug/Exe” or “Release/Exe”). Configure AVR Studio for the debug platform and AVR in use. The program is now ready to run.

3.6 Filter Circuit

The output filter circuit is not mandatory, but improves sound quality considerably. One possible filter circuit is described in application note AVR335. If using the circuit described in AVR335 please note that only the output part of the circuit is required; there is no need for the microphone pre-amplifier. Alternatively, a simpler filter circuit is shown in Appendix C.

4 Porting

The program can be ported to other AVR devices quite easily. Things that may need to be changed include reference type for packed data memory and PWM output pin. For example, to use the decoder on an ATmega32, replace all “__hugeflash” with “__flash” in files “main.c”, “adpcm.c” and “data.c”. The PWM output, OC1A, of ATmega32 is located on pin PD5 instead of PB5. Hence, the function “*initialize_pwm()*“ in file “adpcm.c” must be changed accordingly.

It is recommended to use an AVR with integrated multiplier.

5 Performance

The amount of Flash memory required for a sound recording depends on the sample rate, the ADPCM bit rate, and of course, the length of the recording. The table below shows memory requirements for one second of sound at different bit rates and sample rates.

Table 5-1. Flash Memory Required per One Second of Sound Data.

	Sample rate (Hz)		
Bits / Sample	**4000**	**6000**	**8000**
2	1,00 kB	1,50 kB	2,00 kB
3	1,50 kB	2,25 kB	3,00 kB
4	2,00 kB	3,00 kB	4,00 kB
5	2,50 kB	3,75 kB	5,00 kB

For example: a 10-second recording encoded with 4 bits and at a sample rate of 8 kHz requires 320 kilobits = 40 kilobytes (kB) of memory. This means that, for example, an ATmega128 running the ADPCM decoder (about 3kB) can play back up to (128-3) kB / (32/8) = more than 31 seconds of 8-kHz, 4-bit, ADPCM encoded data.

5.1 Clock Cycles

The table below shows clock cycle requirements per one sample. The bit rate has only a minimal effect on clock cycle requirements (the average values in the table are from the 4-bit version). To find out the required clock frequency of the AVR, multiply the total number of clock cycles by the sample rate.

Table 5-2. Average and Worst-Case Clock Cycle Requirements.

Function	Average Clock Cycles / Sample	Worst-case Clock Cycles / Sample
get_code()	130	140
decode()	1600	1860
T/C3 ISR	60	70
Other	140	150
Decoder Loop Total	**1930**	**2220**

For example: the sound recording is sampled at 7812 Hz, and the average total clock cycle requirement is 1930. The AVR needs to operate at 1930 x 7812 Hz = 15 MHz, which rounds to 16 MHz with headroom. The worst-case clock cycle requirement does exceed 16 MHz and therefore a small buffer is needed to prevent audible distortions caused by calculations taking too long. The length of the buffer can be adjusted (see source code). A long buffer reduces the risk of a buffer underrun but requires more memory (two bytes per element).

5.2 Memory

The memory requirements are listed in the table below. The source code includes the possibility to use all bit levels (2…5 bits), but in a typical application only one bit level is needed. Removing the support for other bit levels will reduce the program memory requirement.

Table 5-3. Memory Requirements (Excluding Sound Data).

Code Memory (bytes)	Data Memory (bytes)
< 3000 (+ sound data)	< 40 (+ output buffer)

6 Appendix A: Tables

This appendix shows tables needed in 4-bit ADPCM decoding. Corresponding values for other bit levels (2, 3, and 5) can be found directly from the source code ("*adpcm.c*"). In the source code some table values are represented using fixed-point decimal numbers. The number of bits reserved for the decimal part is indicated by Qn notation. For example, using Q4 notation the number 4.75 is written in binary as "0000 0000 0100.1100".

Table 6-1. Function W() and F() Values for 4-bit Coding.

|I(k)|	**W|I(k)| (Q4 notation)**	**F|I(k)| (Q9 notation)**
0	-0.75	0
1	1.13	0
2	2.56	0
3	4.00	1
4	7.00	1
5	12.38	1
6	22.19	3
7	70.13	7

Table 6-2. Inverse Quantizing Table for 4-bit Coding.

|I(k)|	**|DQLN(k)| (Q7 notation)**
0	-2048
1	4
2	135
3	213
4	273
5	323
6	373
7	425

7 Appendix B: Formulas

This appendix shows the mathematical formulas used in decoding the ADPCM signal. Headings refer to block names in Figure 1-2.

7.1 Inverse Adaptive Quantizer

To convert from logarithmic domain to linear, the following formula is used:

$$d_q(k) = 2^{d_{q\ln}(k)+y(k)}$$

7.2 Quantizer Scale Factor Adaptation

Locked and unlocked scale factors are combined with the formula

$$y(k) = a_l(k)y_u(k-1) + (1-a_l(k))y_l(k-1)$$

where

$$0 \le a_l(k) \le 1$$

The unlocked (fast) scale factor $y_u(k)$ (see tables in Appendix A for definitions of function W values for different bit rates):

$$y_u(k) = \left(1-2^{-5}\right)y(k) + 2^{-5}W\left|I(k)\right|$$

The locked (slow) scale factor $y_l(k)$:

$$y_l(k) = \left(1-2^{-6}\right)y_l(k-1) + 2^{-6}y_u(k)$$

$y_u(k)$ is limited between 1.06 and 10.00.

7.3 Adaptation Speed Control

The adaptation speed is controlled with parameter a_l. It approaches 1 with speech signals and 0 with audio band data signals. The parameter is calculated by limiting the speed control parameter $a_p(k)$ to be less than or exactly 1:

$$a_l(k) = \begin{cases} 1, & if\ a_p(k-1) > 1 \\ a_p(k-1), & if\ a_p(k-1) \le 1 \end{cases}$$

Unlimited speed control parameter $a_p(k)$ is defined as:

$$a_p(k) = \begin{cases} \left(1-2^{-4}\right)a_p(k-1) + 2^{-3}, & if\ \left|d_{ms}(k) - d_{ml}(k)\right| \ge 2^{-3}d_{ml}(k) \\ \left(1-2^{-4}\right)a_p(k-1) + 2^{-3}, & if\ y(k) < 3 \\ \left(1-2^{-4}\right)a_p(k-1) + 2^{-3}, & if\ t_d(k) = 1 \\ 1, & if\ t_r(k) = 1 \\ \left(1-2^{-4}\right)a_p(k-1), & else \end{cases}$$

The following variables (short- and long-term averages of $F|I(k)|$) are needed in calculation of a_p:

$$d_{ms}(k) = (1-2^{-5})d_{ms}(k-1) + 2^{-5}F|I(k)|$$

$$d_{ml}(k) = (1-2^{-7})d_{ml}(k-1) + 2^{-7}F|I(k)|$$

Values for $F|I(k)|$ are fetched from static table (see Appendix A).

7.4 Tone And Transition Detector

Tone detection:

$$t_d = \begin{cases} 1, & if\ a_2(k) < -0.71875 \\ 0, & else \end{cases}$$

Transition detection:

$$t_r = \begin{cases} 1, & if\ a_2(k-1) < -0.71875\ and\ |d_q(k)| > 24 * 2^{y_l(k-1)} \\ 0, & else \end{cases}$$

7.5 Adaptive Predictor

Signal estimate:

$$s_e(k) = s_{ep} + s_{ez}$$

where

$$s_{ep} = \sum_{i=1}^{2} a_i(k-1)s_r(k-i)$$

and

$$s_{ez} = \sum_{j=1}^{6} b_j(k-1)d_q(k-j)$$

A-coefficient updating:

$$a_1(k) = (1-2^{-8})a_1(k-1) + 3 * 2^{-8} * \mathrm{sgn}(p(k))\,\mathrm{sgn}(p(k-1))$$

$$a_2(k) = (1-2^{-7})a_2(k-1) +$$

$$2^{-7} * \{\mathrm{sgn}(p(k))\,\mathrm{sgn}(p(k-2)) - f(a_1(k-1))\,\mathrm{sgn}(p(k))\,\mathrm{sgn}(p(k-1))\}$$

where

$$p(k) = d_q(k) + s_{ez}(k),$$

$$f(a_1) = \begin{cases} 4a_1, & if\ |a_1| \le 0.5 \\ 2\,\mathrm{sgn}(a_1), & if\ |a_1| > 0.5 \end{cases}$$

Limitation:

$$|a_2(k)| \le 0.75 \quad and \quad |a_1(k)| \le 1 - 2^{-4} - a_2(k)$$

B-coefficient updating, when ADPCM-code is 2, 3, or 4-bit:

$$b_i(k) = \left(1 - 2^{-8}\right) b_i(k-1) + 2^{-7}\,\mathrm{sgn}\left[d_q(k)\right]\mathrm{sgn}\left[d_q(k-i)\right]$$

B-coefficient updating, when ADPCM-code is 5-bit:

$$b_i(k) = \left(1 - 2^{-9}\right) b_i(k-1) + 2^{-7}\,\mathrm{sgn}\left[d_q(k)\right]\mathrm{sgn}\left[d_q(k-i)\right]$$

If a transition is detected:

$$t_r = 0 \quad \Rightarrow \quad \begin{cases} a_i(k) = 0, & i = 1,2 \\ b_i(k) = 0, & i = 1..6 \end{cases}$$

7.6 Reconstructed Signal Calculator

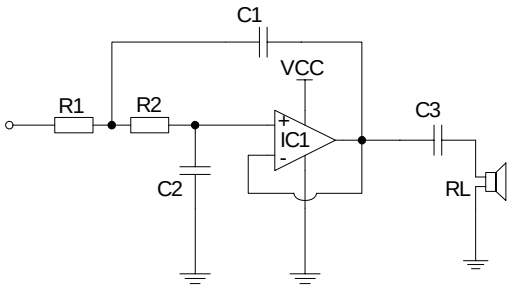
The reconstructed signal $s_r(k)$ is calculated by simply adding the signal estimate to the difference signal:

$$s_r(k) = s_e(k) + d_q(k)$$

8 Appendix C: Filter circuit

The filter illustrated in the figure below is a second-order, low-pass, Butterworth filter with Sallen-Key topology. The cut-off frequency is at 3000 Hz. The PWM base frequency, 31.25 kHz, is attenuated by about 40 dB. The filter is implemented using a single-sided low-voltage operational amplifier TS921, which can be used with operating voltages ranging from 2.7 to 12 V.

Figure 8-1. Filter Schematics.



Gain and operating voltage Vcc may need adjustments depending on the load, RL. A suitable operating voltage for this configuration is +5V.

Table 8-1. Filter Component Values.

Component	Value	Description
R1	1,69kΩ	Butterworth filter resistor
R2	3,4kΩ	Butterworth filter resistor
C1	0,022µF	Butterworth filter capacitor
C2	0,022µF	Butterworth filter capacitor
C3	1µF	DC coupling capacitor
IC1	TS921	Operational amplifier
RL	≥ 32Ω (1)	Miniature speaker, headphones, etc.

(1) Depends on IC1. See manufacturer's datasheet.

9 Appendix D: Files Included

This section lists the files distributed with the application note.

9.1 Decoder

The file “data.c” contains the encoded sound recording. To experiment with different recordings, delete data.c, copy one of the files from the “Samples”-directory to the source directory, rename it to “data.c” and change the BITS and INPUTSIZE parameters from the file “adpcm.h” accordingly. Then compile and link the program.

Decoder files are located in the Source\Decoder directory.

Table 9-1. Decoder Files.

File Name	Description
adpcm.c	ADPCM algorithms
adpcm.h	ADPCM header file
AVR336.eww	IAR workspace/project file
data.c	Encoded sound data file (see header for information)
Decoder.ewp	IAR workspace/project file
main.c	Main program
update_yl.asm	ASM routine to update scale factor

9.2 Encoder

The encoder files are located in the Source\Tools directory.

Table 9-2. Encoder Files.

File Name	Description
decode.c	Source code for decoder
decode.exe	Compiled decoder
encode.c	Source code for encoder
encode.exe	Compiled encoder
g7*.*	Supporting source code files for encoder and decoder
c-izer.cpp	Source code for text file to C code converter
c-izer.exe	Compiled text file to C code converter

Portions of software based on code released to public domain by Sun Microsystem®, Inc.

9.3 Sound Samples

Sound samples are located in the Source\Samples directory.

Table 9-3. Sound Samples.

File Name	Technical Details
Example_5bit.c	ADPCM, 5-bit, 7.8kHz
Example_4bit.c	ADPCM, 4-bit, 7.8kHz
Example_3bit.c	ADPCM, 3-bit, 7.8kHz
Example_2bit.c	ADPCM, 2-bit, 7.8kHz

10 Appendix E: Generating ADPCM Files

An ADPCM encoder is needed to produce the packed files for the decoder described in this application note. The decoder will decode data, which has ben packed with any standard-compliant ADPCM encoder but in order to obtain highest possible quality it is recommended to use a customised, non-standard encoder for generating the ADPCM sound files. This is because the decoder doesn't strictly follow the ITU-T standard (see chapter *Compliance with ITU-T G.726*).

The operating principle of the encoder is briefly explained in "Theory of operation" – chapter.

10.1 Using the Encoder

Included with this application note is a pre-compiled encoder, which can be run on a PC with Windows® operating systems. The encoder is run from the command line with the following parameters:

```
encode [-2|3|4|5] [-a|u|l] –i infile -o|c outfile

-2   Generate G.726 16kbps (2-bit) data
-3   Generate G.726 24kbps (3-bit) data
-4   Generate G.726 32kbps (4-bit) data [default]
-5   Generate G.726 40kbps (5-bit) data

-a   Process 8-bit A-law input data
-u   Process 8-bit u-law input data
-l   Process 16-bit linear PCM input data [default]

-i   Input filename (binary)

-o   Output filename (binary)
-c   Output filename (text file in format ready to be pasted into a
c-array)
```

For example: the following command line instructs the encoder to use 32kbps ADPCM to encode a 16-bit linear PCM file called "sound1.bin" to a text file named "encoded1.txt":

```
encode –4 –l -i sound1.bin –c encoded1.txt
```

The output file contains a hex listing, as follows (hex data is only an example of what the output may look like):

```
0xef, 0xc1, 0xff, 0xde, 0x7b, 0xff, 0xff, 0xff,
0xfe, 0xfb, 0xff, 0xff, 0xf7, 0xfe, 0x7f, 0xff,
...
```

The text file must then be formatted into a valid *.c file, which can be used when compiling the ADPCM decoder project. For this purpose, add lines shown in bold below (hex data is still just an example):

```
#pragma memory = __hugeflash
char packed_ [] = {

0xef, 0xc1, 0xff, 0xde, 0x7b, 0xff, 0xff, 0xff,
0xfe, 0xfb, 0xff, 0xff, 0xf7, 0xfe, 0x7f, 0xff,
…
0x10, 0xf1, 0x45, 0x21, 0x44, 0x8c, 0xff, 0xff,
0xff

};
#pragma memory = default
char __hugeflash *packed = packed_;
```

Note that the command line encoder generates hex output files that end with a comma. Make sure the array data does not end with a comma.

Save the file as data.c and copy it to the source directory of the ADPCM decoder. Finally, count the number of elements in the array and use this number to configure the INPUTSIZE parameter of file adpcm.h (see section “Implementation Example”). Each full line of elements contains eight (8) characters of data. To quickly count the total lines use a plain text editor that indicates on which line the cursor is when editing and move the cursor from the first line to the last.

A small application named C-IZER is also included. It performs all necessary operations described above for creating a valid C-file from the text file generated by ENCODE. The application also returns the value for the INPUTSIZE parameter.

10.2 Using WAV-Files

WAV is a sound format developed by Microsoft® and used extensively in Microsoft Windows®. Conversion tools are available to allow most other operating systems to play WAV-files. The encoder included with this application note can be used to convert WAV-files to ADPCM records, which can then be played back from the target AVR.

In order to use WAV-files with the ADPCM encoder, the files should be saved as 16-bit PCM in mono. The default settings of the decoder rely on a sampling frequency of 7.8 kHz.

WAV-files include a header, which the ADPCM encoder does not differentiate from sound data. This means that the encoded sound file may start with a few bytes that actually represent header data processed as sound samples. Typically, this is not audible and the WAV-file header can usually be simply disregarded.

10.3 Compiling the Encoder

The encoder must be recompiled if it is to be used on other platforms than Windows® PC. Source codes are included with the application note.

11 References

1. Atmel Application Note AVR335: Digital Sound Recorder with AVR ® and DataFlash ®.
   http://www.atmel.com

2. ITU-T Recommendation G.726: 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM).
   http://www.itu.int

3. ANSI-C Implementations of CCITT G.711, G.721 and G.723 Voice Compressions by Sun Microsystems, Inc.
   ftp://ftp.cwi.nl/pub/audio/ccitt-adpcm.tar.gz

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature