AS950 ARM Applications Library

Version 1.1

Programmer's Guide

ARM

ARM DUI 0081B

AS950 ARM Applications Library
Programmer's Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
Oct. 98	A	First release
Oct. 01	B	Second release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. 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.

Contents
AS950 ARM Applications Library Programmer's Guide

Preface

This preface introduces the AS950 ARM Applications Library Programmer’s Guide. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page x.

About this book

This guide is provided with the ARM Applications Library. It is assumed that the ARM Applications Library sources are available as a reference. It is also assumed that the reader has access to programmer guides for C and ARM assembly language.

Intended audience

This book is written for all developers who want to to evaluate, develop, and optimize software applications for the ARM RISC processor family. It assumes you are an experienced software developer, and that you are familiar with using ADS.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the ARM Applications Library.

Chapter 2 ***Adaptive Differential Pulse Code Modulation***

Read this chapter for details on implementation, and for function descriptions for *adaptive differential pulse code modulation* (ADPCM).

Chapter 3 ***G.711—A-law, µ-law, PCM Conversions***

Read this chapter for details on implementation, and for function descriptions for the G.711 standard for A-law and µ-law conversion of *pulse code modulation* (PCM) signals.

Chapter 4 ***Fast Fourier Transform and Windowing***

Read this chapter for details on implementation, and for function descriptions for the *fast Fourier transform* (FFT), and implementations of Hamming and Hanning windows that can be used with the FFT.

Chapter 5 ***Two-Dimensional Discrete Cosine Transform***

Read this chapter for details on implementation, and for function and macro descriptions for *two-dimensional* (2D) *discrete cosine transform* (DCT).

Chapter 6 ***Huffman and Bit Coding/Decoding***

Read this chapter for details on implementation, and for function descriptions of a Huffman coder/decoder that uses a general bit codec based on lookup tables.

Chapter 7 ***Filters***

Read this chapter for details on implementation, and for function and macro descriptions for *finite impulse response* (FIR), *infinite impulse response* (IIR), and *least mean square* (LMS) filters.

Chapter 8 ***IS-54 Convolutional Encoder***

Read this chapter for details on implementation, and for function and macro descriptions for the convolutional encoder from the IS-54 standard for digital mobile telephones in the United States of America.

Chapter 9 ***Multi-tone Multi-frequency Generation/Detection***

Read this chapter for details on implementation, and for function descriptions for the *multi-tone multi-frequency* (MTMF) detector and generator.

Chapter 10 ***Bit Manipulation***

Read this chapter for details on macro descriptions for performing common bit manipulation routines.

Chapter 11 ***Mathematics***

Read this chapter for details on macro descriptions for performing common mathematical functions.

Typographical conventions

The following typographical conventions are used in this book:

`monospace` Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.

`monospace` Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code and ARM processor signal names.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This book contains information that is specific to the version of the Applications Library supplied with the *ARM Developer Suite* (ADS). Refer to the books in the ADS document suite for information on other components of ADS.

In addition, refer to the following documentation for specific information relating to ARM products:

- ARM Reference Peripheral Specification (ARM DDI 0062)

- the ARM datasheet or technical reference manual for your hardware device.

Other publications

For information that may be useful when using the ARM Applications Library, refer to:

- *General Aspects of Digital Transmission Systems; Terminal Equipments, Volume III, Recommendations G.700-G.795*, International Telegraph and Telephone Consultative Committee (ISBN 92-61-033415)
- Y.Arai, T.Agui, and M.Nakajima. *A Fast DCT-SQ Scheme for Images. Trans. of the IEICE.* E 71(11):1095 (Nov. 1988)
- Mark Nelson, *The Data Compression Book* (ISBN 1-55851-214-4)
- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language* (ISBN 0-13-110370-8)

Feedback

ARM Limited welcomes feedback on both the ARM Applications Library, and its documentation.

Feedback on the ARM Developer Suite

If you have any problems with the ARM Developer Suite, please contact your supplier. To help your supplier provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

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

- the document title
- the document number
- the page number(s) to which you comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the AS950 ARM Applications Library. It contains the following sections:

- *About the ARM Applications Library* on page 1-2
- *Registers in macro arguments* on page 1-7
- *Building and running a demonstration* on page 1-8.

1.1 About the ARM Applications Library

The ARM Applications Library Version 1.1 is a suite of optimized ARM assembly language and C source code for commonly used *digital signal processing* (DSP), mathematical, and bit manipulation functions. It has been designed to help you to evaluate, develop, and optimize software applications for the ARM RISC processor family. The library should be used in conjunction with the *ARM Developer Suite* (ADS).

The following is provided with each of the code modules:

- comprehensive documentation
- example and test applications in two subdirectories:
  - `src` contains C routines and source files
  - `tstfiles` contains test data files (for example, a bitmap image for DCT)
- supporting utilities needed to build the examples using ADS.

A general C utility library, util_lib, is also provided to demonstrate each of the main components. You are free to incorporate any of the source code contained in the library into any products, subject to the terms of the license agreement.

The following sections briefly describe each of the code components, and give examples of potential applications.

1.1.1 Noncompliant adaptive differential pulse code modulation

Adaptive differential pulse code modulation (ADPCM) is a compression and decompression algorithm that uses adaptive quantizers and adaptive predictors to compress sound data, typically speech. The implemented encoder takes 16-bit *pulse code modulation* (PCM) data to produce a representation using 4-bit ADPCM values.

The ADPCM implementation is not compliant with the G.726 standard or any other ADPCM standard. Therefore, it cannot be used to interface to other ADPCM compressors/decompressors. It does not include error correction, so it is not suitable for transmission over an imperfect link. However, the implementation is suitable for reducing audio storage overheads.

See Chapter 2 *Adaptive Differential Pulse Code Modulation*.

1.1.2 G.711

G.711 is a recommendation defined by the *International Telegraph and Telephone Consultative Committee* (CCITT) (now the *Telecommunication Standardization Sector* of the *International Telecommunication Union* (ITU-T)).

Analog signals are digitized using *pulse code modulation* (PCM) and are then compressed from linear 16-bit PCM values to 8-bit A-law or µ-law values using logarithmic compression. Conversely, 8-bit A-law or µ-law values can be decompressed to linear 16-bit PCM values.

This compression is recommended for encoding voice-frequency signals because logarithmic compression only loses information that the human ear cannot hear.

Applications include telecommunications (cordless phones), multimedia (lower audio storage overheads), and inter-chip connections (reduces bus width to 8-bits).

See Chapter 3 *G.711—A-law, µ-law, PCM Conversions*.

1.1.3 Fast Fourier transform and windowing

The *fast Fourier transform* (FFT) is one of the main *digital signal processing* (DSP) algorithms. FFTs convert *one-dimensional* (1D) data, typically sound, between the time domain and the 1D frequency domain.

Windowing selects segments of data (pre-weighting of the data) for subsequent frequency domain analysis using the FFT. Both Hamming and Hanning windows are provided.

FFT can be used for:

spectral analysis

Identifying the magnitude of each frequency within a signal (the spectral components).

digital filtering

Applying FFT, removing a range of frequency components using a filtering technique and reconstructing the signal using the inverse FFT. Windowing can be used to prevent clicks at frame boundaries.

calculation of correlations

Increasing the speed of a traditional *finite impulse response* (FIR) filter that requires the calculation of large correlations.

calculation of convolutions

Common calculations in the construction of filters.

See Chapter 4 *Fast Fourier Transform and Windowing*.

1.1.4 Discrete cosine transform

The *discrete cosine transform* (DCT) converts a *two-dimensional* (2D) graphics image between the spatial domain and the 2D frequency domain.

The DCT is implemented using an 8x8 block and does not directly lead to compression. However, the DCT does produce frequency data that is suitable for use in graphics compression algorithms such as JPEG or MPEG.

See Chapter 5 *Two-Dimensional Discrete Cosine Transform*.

1.1.5 Huffman encoding/decoding

Huffman encoding/decoding exploits the fact that discrete amplitudes of a signal do not occur with equal probability. Huffman encoding creates variable length codes that are an integral number of bits. Symbols with higher probabilities have fewer bits in their codes. The codes can be uniquely decoded, despite being of different lengths, because each is created with a unique prefix.

Most compression algorithms use some form of Huffman encoding/decoding where performance is more important than compression rate. Applications include JPEG, font compression, and dictionary compression.

See Chapter 6 *Huffman and Bit Coding/Decoding*.

1.1.6 Filtering

Digital filters can be used to remove data from a sound signal, such as noise and unwanted frequencies, by attenuating or reducing certain bands of frequencies and allowing the others to pass. Digital filters are often classed into one of four types:

- low-pass
- high-pass
- band-pass
- band-stop.

A low-pass filter allows frequencies below a certain value to pass through as-is while attenuating frequencies above that point. Conversely, a high-pass filter lets frequencies above a certain value pass through as-is while attenuating frequencies below that point. For example, a low-pass filter may be used to remove high frequency noise from a speech signal.

A band-pass filter allows a selected frequency band in a signal to pass through as-is and reduces all frequencies outside of this band. This has the effect of emphasizing the selected band. For example, band-pass filters might be used in a graphic equalizer.

A band-stop filter reduces a selected frequency band in a signal and allows all frequencies outside this band to pass through unaffected. For example, it can be used to remove a 50Hz mains hum from a signal.

Three filters are provided:

- *finite impulse response* (FIR)
- *infinite impulse response* (IIR)
- *least mean square* (LMS).

The type of filter to be used, such as a low-pass or band-stop, is identified by the coefficients (weight values) that are used with the filter.

The FIR filter is a nonrecursive linear filter (no feedback) and performs a moving and weighted average on the input data. Finite refers to the fact that an input pulse results in energy at only a finite number of samples after which the output returns to zero. FIR filters are suitable for telecommunications use, such as distinguishing between the channels used in a V.22bis modem link.

The IIR filter is a recursive linear filter that employs feedback to allow sharper frequency responses to be obtained for fewer filter coefficients. Infinite refers to the fact that the output from a unit pulse input exhibits non-zero outputs for an arbitrarily long time.

The LMS filter is an adaptive FIR digital filter that is self-learning and adapts the impulse response of the FIR filter to a desired signal.

The IIR and LMS filters are suitable for echo cancellation and line equalization, compensating for errors introduced by transmission over a channel.

See Chapter 7 *Filters*.

1.1.7 IS-54 convolutional encoder

IS-54 is the *Telecommunications Industry Association/Electronic Industries Association* (TIA/EIA) interim standard for the Cellular System Dual-Mode Mobile Station-Base Station Compatibility Standard. This is the digital mobile telephone standard used in the United States of America.

The convolutional encoder is used to code 89 bits of data at a time:

- 77 class 1 bits from the IS-54 speech coder
- seven bits of cyclic redundancy check
- five bits of tail.

This algorithm also demonstrates how to optimally interleave two 16-bit words to generate a 32-bit word. See Chapter 8 *IS-54 Convolutional Encoder*.

1.1.8 Multi-tone multi-frequency and Goertzel algorithm

The multi-tone multi-frequency algorithm is used in detecting and generating tones such as *dual-tone multi-frequency* (DTMF), multi-frequency, busy, and dial tones.

DTMF transmission and reception is resistant to line noise and distortion, and is performed in applications such as answering machines, modems, telephones and PBXs. E1 applications (the European equivalent to T1) and US telephone frame equipment are applications that perform multi-frequency tone detection and generation functions. Busy and dial tone generation and detection capabilities are required by various telephony applications.

See Chapter 9 *Multi-tone Multi-frequency Generation/Detection*.

1.1.9 Bit manipulation

The bit manipulation routines perform a number of operations on words on a per-bit basis. The routines include bit and byte reversal over a word, binary coded decimal addition, finding the least and most significant bits set in a word, and population count over a set of words (determining the number of bits set).

See Chapter 10 *Bit Manipulation*.

1.1.10 Mathematics

The mathematics functions provide optimal algorithms to perform common mathematical tasks using differing precision and fixed-point input data. These algorithms are suitable for high-precision work such as encryption or vector manipulation (three-dimensional graphics).

The mathematics component also contains routines to perform addition of absolute values, c = a + abs(b), and signed-saturated addition of two 32-bit integers.

See Chapter 11 *Mathematics*.

1.2 Registers in macro arguments

In the ARM Assembler macro definitions, arguments are prefixed with a dollar sign ($). Except where otherwise noted, these arguments represent variable names for registers that must be substituted for actual registers when creating an instantiation of the macro.

1.2.1 Example

In the following example, an ARM assembly language macro `example_macro` could be defined in a file `examplem.h`:

```
    MACRO
    example_macro $c, $a, $b
        ADD $c, $a, $b
    MEND
```

The macro arguments `$a` and `$b` represent the two input registers, and `$c` is the output register

The macro file can be included in an assembly language source file using the `INCLUDE` directive. The following example shows how the macro might be instantiated:

```
    INCLUDE examplem.h
example_macro_start
    example_macro    R0, R0, R1
    MOV              pc, lr
```

The macro adds the values of the input registers r0 and r1, and stores the result in r0.

Refer to the *ADS Developer Guide* in the ADS documentation for information on register usage conventions and the *ARM Procedure Call Standard* (APCS).

1.3 Building and running a demonstration

This section describes how to build and run the Applications Library using ADS. For more information on using ADS, refer to the ADS documentation.

1.3.1 Variants

Each component of the Applications Library has a UNIX makefile and two CodeWarrior IDE `.mcp` project files, which are:

- the core, library components file
- the executable version of the core, library components file.

These files provide options for building different variants of the components. You can build variants for:

- little-endian and big-endian ARM versions
- little-endian and big-endian Thumb versions.

The variations are identified by the following variant names:

ArmBigDebug	ARM, big-endian, debug information
ArmBigRelease	ARM, big-endian, no debug information
ArmLittleDebug	ARM, little-endian, debug information
ArmLittleRelease	ARM, little-endian, no debug information
ThumbBigDebug	Thumb, big-endian, debug information
ThumbBigRelease	Thumb, big-endian, no debug information
ThumbLittleDebug	Thumb, little-endian, debug information
ThumbLittleRelease	Thumb, little-endian, no debug information.

1.3.2 Linker output

The output produced when building a variant is the default format defined by the version of the toolkit used to perform the build. To build a different output, add the appropriate command-line argument to the linker configuration. For information on the output formats supported by the linker and how to change the output produced by the linker, see the *ADS Linker and Utilities Guide* in the ADS documentation.

1.3.3 Building and running

This section describes how to build and run an Applications Library component. To compile components of the Applications Library on Windows with the supplied projects, ADS is required. Refer to *CodeWarrior IDE Guide* in the ADS documentation for details on building using project files.

1. Double click on the executable `.mcp` file located in the appropriate component folder, not the library `.mcp` file (identified by `lib` appended to the component name).
2. Build one of the variants of the project.
3. Run the binary under the Windows debugger, ensuring the byte-sex for the debugger is that of the selected variant.

1.3.4 Execution considerations

Each component allows you to enter values at run-time for options on testing and demonstrating the functionality. If you are prompted for an input data file, suitable test files can be found in the `tstfiles` folder for the appropriate component.

When running C-based test versions, some of the demonstrations (such as Discrete Cosine Transform and Fast Fourier Transform) require considerable time to complete if running under the ARMulator. For these cases, it is advisable to download the executable to an ARM Integrator, or other hardware, and run the test there. The ARM Integrator can be purchased from ARM. See `http://www.arm.com`.

Chapter 2
Adaptive Differential Pulse Code Modulation

This chapter describes an implementation of *adaptive differential pulse code modulation* (ADPCM). It contains the following sections:

- *Overview* on page 2-2
- *ADPCMState data structure* on page 2-7
- *Functions* on page 2-8.

2.1 Overview

This section provides general information on ADPCM encoding and decoding.

2.1.1 Implementation

This section describes the formulas that have been used in the implementation for the encoding and decoding ADPCM.

This implementation of ADPCM is noncompliant with G.726 or any other standard, including the earlier G.721 and G.723 standards. The ARM implementation is suitable only for use where bit errors cannot occur, such as in computer systems. If bit errors are introduced into the process, they propagate through the coder. Therefore, this implementation is not suitable for compression before transmission across channels that can add bit errors.

The encoder for the ADPCM has been implemented to take 16-bit PCM input values and produce a 4-bit compressed ADPCM output value for each input value. Conversely, the ADPCM decoder has been implemented to take 4-bit compressed ADPCM input values that were generated by the encoder and produce the corresponding 16-bit PCM output value for each input value.

The decoder is discussed before the encoder because the encoder makes use of the decoder.

Decoding ADPCM

For each ADPCM value y_i, where $0 \leq i < \mathrm{N}$, the corresponding PCM output, x_i, is given by:

$$step_i = stepSize[index_i]$$

$$delta_i = y_i$$

$$index_{i+1} = index_i + indexTable[delta_i]$$

$$delta_i = \begin{cases} delta_i - 8 & delta_i > 7 \\ delta_i & otherwise \end{cases}$$

$$x_i = \begin{cases} x_{i-1} - (2 \times delta_i + 1) \times {}^{step_i}\!/_{8} & y_i > 7 \\ x_{i-1} + (2 \times delta_i + 1) \times {}^{step_i}\!/_{8} & otherwise \end{cases}$$

where:

$x_{-1} = 0$

$index_0 = 0$

indexTable and *StepSize* are defined as follows:

$$\begin{array}{ll} indexTable[16] = & [\\ & -1 \quad -1 \quad -1 \quad -1 \quad 2 \quad 4 \quad 6 \quad 8 \\ & -1 \quad -1 \quad -1 \quad -1 \quad 2 \quad 4 \quad 6 \quad 8 \\ &] \end{array}$$

$stepSize[89] =$

[

7	8	9	10	11	12	13	14	16	17
19	21	23	25	28	31	34	37	41	45
50	55	60	66	73	80	88	97	107	118
130	143	157	173	190	209	230	253	279	307
337	371	408	449	494	544	598	658	724	796
876	963	1060	1166	1282	1411	1552	1707	1878	2066
2272	2499	2749	3024	3327	3660	4026	4428	4871	5358
5894	6484	7132	7845	8630	9493	10442	11487	12635	13899
15289	16818	18500	20350	22385	24623	27086	29794	32767	

]

Saturation is also incorporated into the algorithm so that:

x_i cannot exceed the range $-2^{15} \le x_i \le 2^{15} - 1$

$index_i$ cannot exceed the range $0 \le index_i \le 88$.

Encoding ADPCM

For each value x_i, where $0 \le i < N$, the corresponding ADPCM output, y_i, is given by:

$$difference_i = x_i - DecodeADPCM\left(y_{i-1}\right)$$

$$step_i = stepSize\left[index_i\right]$$

$$delta_i = \frac{\left|difference_i\right| \times 4}{step_i} \quad \text{(integer division)}$$

$$delta_i = \begin{cases} delta_i + 8 & difference_i < 0 \\ delta_i & otherwise \end{cases}$$

$$index_{i+1} = index_i + indexTable\left[delta_i\right]$$

$$y_i = delta_i$$

where:

$y_{-1} = 0$

$index_0 = 0$

indexTable and *stepSize* are defined as given in *Decoding ADPCM* on page 2-3.

Saturation is also incorporated into the algorithm so that:

$delta_i$ when initially calculated, before the addition of 8 for negative values, cannot exceed the range $0 \le delta_i \le 7$

$index_i$ cannot exceed the range $0 \le index_i \le 88$.

2.1.2 Files

The files in Table 2-1 are provided in the implementation.

Table 2-1 ADPCM files

Filename	Archive name	Code type	Functionality
`adpcms.s`	`arm_adpm`	ARM assembly language	ADPCM coding and decoding
`adpcms.h`	`arm_adpm`	C header	ADPCM function prototypes
`adpstruc.h`	`arm_adpm`	C header	ADPCM structure definition

2.2 ADPCMState data structure

This structure is used to maintain step-size and prediction values internally during the ADPCM operations and between routine calls.

The ADPCM encode and decode routines are passed a pointer to this structure.

2.2.1 Definition

```
typedef struct    ADPCMState    ADPCMState ;
typedef           ADPCMState    *ADPCMStatePtr ;

struct ADPCMState {
    int stepIndex ;
    int prediction ;
} ;
```

2.2.2 Description

The ADPCM routines operate over a set of data values where the current output value is directly related to the previous output value. Because of this, each call to one of the routines requires some history of the previous values. This history is defined by the step-size and the prediction for the next input value, and is maintained in the `ADPCMState` structure between routine calls.

2.2.3 Usage

When starting a new coding or decoding sequence, before you pass the structure to the ADPCM routines, you must initialize the `stepIndex` and `prediction` entries to zero. Between the routine calls for the sequence of input values, you must maintain the structure with the values that are updated during the operation of the routine.

2.3 Functions

This section describes the ADPCM routines. The functions are:

- Encode 16-bit *pulse code modulation* (PCM) to 4-bit ADPCM (*adpcm_encode*)
- Decode 4-bit ADPCM to 16-bit PCM (*adpcm_decode* on page 2-9).

2.3.1 adpcm_encode

You can call this function repeatedly to encode a set of 16-bit PCM inputs to a set of corresponding 4-bit compressed ADPCM samples.

Syntax

```
int adpcm_encode(int pcmSample, ADPCMStatePtr encodeStatePtr)
```

where:

pcmSample is the current 16-bit PCM data value to be encoded. This value must be a 16-bit quantity, even though it is passed in a 32-bit integer word.

encodeStatePtr is a pointer to the `ADPCMState` structure required during the encoding operation.

When this function returns, *encodeStatePtr* points to an updated `ADPCMState` structure that contains the values required during the next call to `adpcm_encode()`. The structure must be maintained as-is between calls.

Return Value

The 4-bit compressed ADPCM encoding of the given 16-bit PCM input data value.

Usage

For each set of PCM data values to be encoded, the entries in the `ADPCMState` structure must be zero-initialized before the first call to `adpcm_encode()`. The `ADPCMState` structure must then be passed to each subsequent function call with the values returned from the previous function call.

2.3.2 adpcm_decode

You can call this function repeatedly to decode a set of 4-bit compressed ADPCM inputs to a set of corresponding uncompressed 16-bit PCM samples.

Syntax

```
int adpcm_decode(int adpcmSample, ADPCMStatePtr decodeStatePtr)
```

where:

adpcmSample is the current 4-bit compressed ADPCM data value to be decoded. This value must be a 4-bit quantity, even though it is passed in a 32-bit integer word.

decodeStatePtr is a pointer to the ADPCMState structure required during the decoding operation.

When this function returns, *decodeStatePtr* points to an updated ADPCMState structure that contains the values required during the next call to adpcm_decode(). The structure must be maintained as-is between calls.

Return Value

The 16-bit uncompressed PCM decoding of the given 4-bit ADPCM input data value.

Usage

Because the ADPCM implementation is specific, the values in *adpcmSample* must be values that have been returned from adpcm_encode(). Data from another ADPCM encoder may not be in the correct format.

For each set of ADPCM data values to be decoded, the entries in the ADPCMState structure must be zero-initialized before the first call to adpcm_decode(). The ADPCMState structure must then be passed to each subsequent function call with the values returned from the previous function call.

Chapter 3
G.711—A-law, µ-law, PCM Conversions

This chapter describes an implementation of the *International Telecommunications Union* (ITU) G.711 standard for A-law and µ-law conversion of *pulse code modulation* (PCM) signals. It contains the following sections:

- *Overview* on page 3-2
- *Functions* on page 3-3.

3.1 Overview

This section provides general information on G.711.

3.1.1 Implementation

The implementation is based on the G.711 standard from the ITU. The G.711 standard is defined in *General Aspects of Digital Transmission Systems; Terminal Equipments, Volume III, Recommendations G.700-G.795*.

3.1.2 Files

The files in Table 3-1 are provided with the implementation. The G.711 standard is implemented as a set of ARM assembly language macros in `g711m.h`.

Table 3-1 G711 files

File name	Archive name	Code type	Functionality
`g711m.h`	`arm_g711`	ARM assembly language	PCM, A-law, and μ-law conversions
`g711uats.s`	`arm_g711`	ARM assembly language	A-law to and from μ-law conversion lookup tables, `G711_u2a_lookup` and `G711_a2u_lookup`
`g711s.s`	`arm_g711`	ARM assembly language	G.711 macro initializations with C-based code wrapping
`g711s.h`	`arm_g711`	C header	G.711 function prototypes

3.2 Functions

This section describes the G.711 macros. The macros are:

- 16-bit linear PCM to 8-bit A-law conversion (*G711_linear2alaw_macro*)
- 8-bit A-law to 16-bit linear PCM conversion (*G711_alaw2linear_macro* on page 3-4)
- 16-bit linear PCM to 8-bit μ-law conversion (*G711_linear2ulaw_macro* on page 3-5)
- 8-bit μ-law to 16-bit linear PCM conversion (*G711_ulaw2linear_macro* on page 3-6)
- 8-bit A-law to 8-bit μ-law conversion (*G711_alaw2ulaw_macro* on page 3-7)
- 8-bit μ-law to 8-bit A-law conversion (*G711_ulaw2alaw_macro* on page 3-9).

3.2.1 G711_linear2alaw_macro

This macro converts a 16-bit linear PCM value to an 8-bit compressed A-law value.

Syntax

```
MACRO G711_linear2alaw_macro $pcm, $alaw, $t1, $t2, $msk
```

where:

$pcm is a register that holds the 16-bit linear PCM value to be converted. This value must be no more than a 16-bit quantity with the least significant bit first, even though it is in a 32-bit register. The value can be less than 16 bits, but each sample must occupy two bytes and must be shifted up, leaving zeros in the bottom unused bits, so that dynamic range is not lost.

$alaw is a register that holds the 8-bit compressed A-law result.

$t1, *$t2*, *$msk* are temporary registers required during the conversion. On output, any value is undefined.

Register differentiation

$pcm, *$t1*, *$t2* and *$msk* must be distinct registers.

$alaw, *$t1* and *$t2* must be distinct registers.

$pcm and *$alaw* need not be distinct registers.

$alaw and *$msk* need not be distinct registers.

3.2.2 G711_alaw2linear_macro

This macro converts an 8-bit compressed A-law value to a 16-bit linear PCM value.

Syntax

```
MACRO G711_alaw2linear_macro $alaw, $pcm, $t1, $t2
```

where:

$alaw is a register that holds the 8-bit compressed A-law value to be converted. This value must be an 8-bit quantity with the least significant bit first, even though it is in a 32-bit register.

$pcm is a register that holds the 16-bit linear PCM result.

$t1, *$t2* are temporary registers required during the conversion. On output, any value is undefined.

Register differentiation

$alaw, `$t1` and *$t2* must be distinct registers.

$pcm, *$t1* and *$t2* must be distinct registers.

$alaw and *$pcm* need not be distinct registers.

3.2.3 G711_linear2ulaw_macro

This macro converts a 16-bit linear PCM value to an 8-bit compressed μ-law value.

Syntax

```
MACRO G711_linear2ulaw_macro $pcm, $ulaw, $msk, $t1, $t2
```

where:

$pcm is a register that holds the 16-bit linear PCM value to be converted. This value must be no more than a 16-bit quantity with the least significant bit first, even though it is in a 32-bit register. The value can be less than 16 bits, but each sample must occupy two bytes and must be shifted up, leaving zeros in the bottom unused bits, so that dynamic range is not lost.

$ulaw is a register that holds the 8-bit compressed μ-law result.

$msk, *$t1*, *$t2* are temporary registers required during the conversion. On output, any value is undefined.

Register differentiation

$pcm, *$t1*, *$t2* and *$msk* must be distinct registers.

$ulaw, *$t1* and *$t2* must be distinct registers.

$pcm and *$ulaw* need not be distinct registers.

$ulaw and *$msk* need not be distinct registers.

3.2.4 G711_ulaw2linear_macro

This macro converts an 8-bit compressed µ-law value to a 16-bit linear PCM value.

Syntax

```
MACRO G711_ulaw2linear_macro $ulaw, $pcm, $seg
```

where:

`$ulaw` is a register that holds the 8-bit compressed µ-law value to be converted. This value must be an 8-bit quantity with the least significant bit first, even though it is in a 32-bit register.

`$pcm` is a register that holds the 16-bit linear PCM result.

`$seg` is a temporary register required during the conversion. On output, any value is undefined.

Register differentiation

`$ulaw` and `$seg` must be distinct registers.

`$pcm` and `$seg` must be distinct registers.

`$ulaw` and `$pcm` need not be distinct registers.

3.2.5 G711_alaw2ulaw_macro

This macro converts an 8-bit compressed A-law value to an 8-bit compressed µ-law value.

Syntax

```
MACRO G711_alaw2ulaw_macro $alaw, $ulaw, $tmp, $table, $hastable
```

where:

$alaw is a register that holds the 8-bit compressed A-law value to be converted. This value must be an 8-bit quantity with the least significant bit first, even though it is in a 32-bit register.

$ulaw is a register that holds the 8-bit compressed µ-law result.

$tmp is a temporary register required during the conversion. On output, any value is undefined.

$table is a register that holds the address of the `G711_a2u_lookup` table, either supplied on input or initialized by the macro. If the macro is to be used repeatedly, *$table* can be initialized the first time the macro is used and passed as a parameter with each subsequent usage.

$hastable is an optional parameter that can contain any value. If *$hastable* is present, *$table* must contain the address of the `G711_a2u_lookup` table. If *$hastable* is not present, *$table* is initialized by the macro.

Usage

The macro requires the address of the `G711_a2u_lookup` table (defined in `g711uats.s`) for the conversion. This address is held in the register identified by *$table*, and can either be:

- supplied to the macro, in which case *$hastable* must be supplied
- initialized by the macro, in which case *$hastable* must not be supplied.

Register differentiation

`$alaw`, `$tmp` and `$table` must be distinct registers.

`$ulaw` and `$tmp` must be distinct registers.

`$ulaw` and `$table` need not be distinct registers. However, the address of the lookup table is overwritten if the same register is used for both parameters.

`$alaw` and `$ulaw` need not be distinct registers.

3.2.6 G711_ulaw2alaw_macro

This macro converts an 8-bit compressed µ-law value to an 8-bit compressed A-law value.

Syntax

`MACRO G711_ulaw2alaw_macro $ulaw, $alaw, $tmp, $table, $hastable`

where:

$ulaw is a register that holds the 8-bit compressed µ-law value to be converted. This value must be an 8-bit quantity with the least significant bit first, even though it is given in a 32-bit register.

$alaw is a register that holds the 8-bit compressed A-law result.

$tmp is a temporary register required during the conversion. On output, any value is undefined.

$table is a register that holds the address of the `G711_u2a_lookup` table, either supplied on input or initialized by the macro. If the macro is to be used repeatedly, *$table* can be initialized the first time the macro is used and passed as a parameter with each subsequent usage.

$hastable is an optional parameter that can contain any value. If *$hastable* is present, *$table* must contain the address of the `G711_u2a_lookup` table. If *$hastable* is not present, *$table* is initialized by the macro.

Usage

The macro requires the address of the `G711_u2a_lookup` table (defined in `g711uats.s`) for the conversion. This address is held in the register identified by *$table*, and can either be:

- supplied to the macro, in which case *$hastable* must be supplied
- initialized by the macro, in which case *$hastable* must not be supplied.

Register differentiation

`$ulaw`, `$tmp` and `$table` must be distinct registers.

`$alaw` and `$tmp` must be distinct registers.

`$alaw` and `$table` need not be distinct registers. However, the address of the lookup table is overwritten if the same register is used for both parameters.

`$ulaw` and `$alaw` need not be distinct registers.

Chapter 4
Fast Fourier Transform and Windowing

This chapter describes an implementation of the *fast Fourier transform* (FFT) and implementations of Hamming and Hanning windows that can be used with the FFT. It contains the following sections:

4.1 Overview

This section provides general information on:

- the implementation of forward FFT and inverse FFT
- the flags relating to FFT optimization and conditional assembly
- Hamming and Hanning windows
- the files provided with the FFT implementation.

4.1.1 Implementation

This section describes the formulas used in the implementation of the forward and inverse FFTs, and the settings that can be adjusted for the implementation. A Radix2 algorithm is used. It also describes the formulas for the windowing techniques.

Forward FFT

The forward FFT, given by:

$$\underline{X} = FFT(\underline{x})$$

performs the calculation specified by the formula:

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n]\left(\cos(nk\omega) - i\sin(nk\omega)\right)$$

where ω = 2π / N and the 1 / *N* scaling that multiplies the summation prevents overflow within the algorithm.

Using the complex identity $e^{i\theta} = \cos\theta + i \sin\theta$, the transform can be written as:

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-ink\omega}$$

Evaluating the sum directly requires of the order of N^2 multiplications. However, this can be reduced to the order of *N* log *N* by rearranging the terms in the following way:

Separate out the even-numbered and odd-numbered elements:

$$X[k] = \frac{1}{N}\left(\sum_{n=0}^{N/2-1} x[2n]e^{-2ink\omega} + \sum_{n=0}^{N/2-1} x[2n+1]e^{-i(2n+1)k\omega}\right)$$

Take out a factor of $e^{-ik\omega}$:

$$X[k] = \frac{1}{N}\left(\sum_{n=0}^{N/2-1} x[2n]e^{-2ink\omega} + e^{-ik\omega}\sum_{n=0}^{N/2-1} x[2n+1]e^{-2ink\omega}\right)$$

Inspection shows that if:

$$Y = FFT(x[0], x[2], x[4], \ldots, x[N-2])$$

and:

$$Z = FFT(x[1], x[3], x[5], \ldots, x[N-1])$$

then:

$$X[k] = \frac{1}{2}\left(Y[k] + e^{-ik\omega}Z[k]\right) \qquad 0 \le k < N/2$$
$$X[k] = \frac{1}{2}\left(Y[k-N/2] - e^{-ik\omega}Z[k-N/2]\right) \qquad N/2 \le k < N$$

The problem has been reduced to calculating two FFTs of size $N / 2$ and performing N complex multiplications.

Note

The bottom bit of k determines whether $x[k]$ is in the FFT calculation for Y or for Z.

By repeating this process for Y and Z, and recursing the FFT algorithm, the next subsection is derived. This process is known as *decimation in time*.

Inverse FFT

The inverse FFT, given by:

$$\underline{x} = IFFT(\underline{X})$$

performs the calculation specified by the formula:

$$x[k]=\frac{1}{N}\sum_{n=0}^{N-1}X[n](\cos(nk\omega)+i\sin(nk\omega))$$

where ω = 2π / N and the 1 / *N* scaling that multiplies the summation prevents overflow within the algorithm.

This can be simplified as given in the case of the forward FFT above.

Table lookup

The cosine and sine values used in the forward and inverse FFT algorithms only depend on the value of *N*. If the value of *N* is fixed, the cosine and sine values required are a set of constants and a lookup table can be created. In addition, the lookup table has symmetries so that only the values between 0 and π / 4 radians need calculating, and all other cosine and sine values can be determined from these values.

Therefore, the FFT is implemented using a cosine and sine lookup table that must be generated and included before assembling the FFT functions. The generation of the lookup table is handled by a separate set of files contained in the `arm_fft\arm_tgen` directory. When the archive files are compiled and executed, a table is generated in `ffttabls.h`. The directory location for the file and the maximum number of points in the FFT, *N*, are specified at runtime.

The directory path should locate the directory for `ffts.s` from that of the executable creating the table, so that the header file is included when assembling `ffts.s`.

For example, given the following structure:

```
apps_lib\arm_fft\fft.mcp
apps_lib\arm_fft\src\ffts.s
apps_lib\arm_fft\variants\[variations]\fft.axf
apps_lib\arm_fft\fft_tgen\fft_tgen.mcp
apps_lib\arm_fft\fft_tgen\variants\[variations]\fft_tgen.axf
```

the `ffttabls.h` table must be in the `apps_lib\fft\src\` directory, which must be identified when executing `fft_tgen.axf`. (The *`variations`* can be found in *Variants* on page 1-8.)

If `fft_tgen.axf` is executed in `apps_lib\fft\fft_tgen\variants\[`*`variations`*`]`, the directory location for the table is `..\..\..\src` because the creation process must change directory by going back through the `\fft_tgen\variants\[`*`variations`*`]` directories to `apps_lib\fft\` and then forward into the `src` directory.

However, if `..\fft_tgen\variants\[`*`variations`*`]\fft_tgen.axf` is executing in `apps_lib\fft\src`, the directory path for the table is empty because the directory structure has been incorporated into the execution of `fft_tgen.axf`.

The number of points in the FFT must be given as the maximum number of points that the FFT can perform with each call, and therefore, the maximum number of inputs to, and outputs from, the FFT. The number of points in the FFT must be a power of two and should be set with consideration to speed and memory.

The more points in the FFT, the larger the lookup table, and hence the data size, and the slower the FFT functions perform due to cache misses. However, the more points in the FFT and the greater the number of possible input values that can be given to each call of the FFT, the greater the number of possible outputs that can be generated. The fewer the points in the FFT and the smaller the table and data size, the quicker the FFT performs, but with fewer inputs that can be processed with each FFT call.

The number of points in the FFT given here only determines the maximum number of possible inputs to the FFT functions, and not the number of inputs that must be passed to each call of the FFT.

If the directory path and/or the number of points are not given at runtime, their default values are used. The default directory path is the directory of the executable creating the table and the default maximum number of points in the FFT is 1024.

4.1.2 FFT optimization and conditional assembly

The following flags are defined in `ffts.s` and relate to FFT optimization and conditional assembly.

OPTIMISE

This flag determines which algorithm the FFT uses. There are two separate algorithms, one optimized for size and one optimized for speed.

If the smaller but slower algorithm is required, the flag must be set to 0 (false). If the larger but highly optimized algorithm is required, this flag must be set to 1 (true). On average, the unoptimized algorithm takes approximately 1.5 times as long as the optimized version for large FFTs and twice as long for smaller FFTs.

Given that the FFT is an N-point function, the code size for the optimized algorithm is (1592 + N) / 2 bytes, including the lookup table. The read/write data size is 64 bytes, not including the input and output buffers. The unoptimized algorithm has a smaller code size of (548 + N) / 2 bytes but the same read/write data size of 64 bytes.

FORWARD, INVERSE

The same function is used to perform a forward and inverse FFT. The direction of the FFT is determined by the `FORWARD` or `INVERSE` flags. To reduce code size, support for either the forward or inverse FFT can be enabled or disabled independently by setting the appropriate flags.

- To enable the forward FFT, set the `FORWARD` flag to 1. To disable the forward FFT, set it to 0.
- To enable the inverse FFT, set the `INVERSE` flag to 1. To disable the forward FFT, set it to 0.

If either or both flags are disabled and the FFT function is called with the direction set to an operation that is disabled, the result is undetermined.

INPLACE, OUTPLACE

The addresses for the input and output buffers are given as arguments to the FFT functions. The FFT is *in-place* if the input buffer and output buffer reference the same block of memory. Conversely, if the input and output buffers reference different blocks of memory, the FFT is *out-of-place*.

- To permit in-place buffers, set the `INPLACE` flag to 1. When `INPLACE` is set to 0, in-place buffers are not supported.
- To permit out-of-place buffers, set the `OUTPLACE` flag to 1. When `OUTPLACE` is set to 0, out-of-place buffers are not supported.

The results are undetermined if:

- the buffers are in-place when only out-of-place buffers are allowed
- the buffers are out-of-place and only in-place buffers are allowed
- neither in-place nor out-of-place buffers are allowed.

REALFFTS

The `ffts.s` file defines a `REALFFT()` function that can be used when the inputs to the FFT are real values.

- If this function is not required, set the `REALFFTS` flag to 0 (false), so that the function is not defined.
- Otherwise, set the flag to 1 (true) to define and export the function.

4.1.3 Hamming and Hanning windows

The Hamming and Hanning Windows are defined by the formula:

$$h[i] = in[i] \times \left((1-\alpha) - \alpha \cos\left(\frac{2\pi i}{N} \right) \right)$$

where:

$$\alpha = \begin{cases} 0.46 & \text{Hamming Window} \\ 0.50 & \text{Hanning Window} \end{cases}$$

4.1.4 Files

The files in Table 4-1 are provided with the implementation.

Table 4-1 FFT files

Filename	Archive name	Code type	Functionality
`ffts.s`	`arm_fft`	ARM assembly language	FFT real coding, and imaginary coding and decoding
`ffttabls.h`	`arm_fft`	ARM assembly language	FFT point lookup table for FFT calculation
`fftstruc.h`	`arm_fft`	C header	`Complex` structure definition
`ffts.h`	`arm_fft`	C header	FFT function prototypes
`windowsc.c`	`arm_fft`	C	`GenerateWindow()`, `Hamming()` and `Hanning()` windowing functions
`windowsc.h`	`arm_fft`	C header	Window function prototypes and constant definitions
`ffttgenc.c`	`arm_fft\arm_tgen`	C	Code for creation of the FFT point lookup table file `ffttabls.h`
`ffttgenc.h`	`arm_fft\arm_tgen`	C header	Constant definitions for creation of the FFT point lookup table

4.2 Complex data structure

This section describes the `Complex` data structure required by the FFT functions. This structure is used to pass or retrieve complex data values to or from the FFT routines.

4.2.1 Definition

```
typedef struct Complex  Complex ;

struct Complex {
    int r ;
    int i ;
} ;
```

4.2.2 Description

The FFT routines operate over complex data values. Each complex number consists of a 32-bit integer containing the real part, followed by a 32-bit integer containing the imaginary part. The `Complex` structure is used to maintain each data value with a real and imaginary part, and is eight bytes long.

4.3 Functions

This section describes the FFT and windowing routines. The functions are:

- Forward or inverse FFT on complex values (*FFT*)
- Forward FFT on real values (*REALFFT* on page 4-12)
- Generating coefficients for Hamming or Hanning window (*GenerateWindow* on page 4-14)
- Perform Hamming window (*HammingWindow* on page 4-15)
- Perform Hanning window (*HanningWindow* on page 4-16).

4.3.1 FFT

This function calculates the forward or inverse FFT for a given set of complex data values and outputs a set of complex transform coefficients to an output data buffer.

Syntax

```
int FFT(Complex *in, Complex *out, int logN, int direction)
```

where:

in is a pointer to the starting address of the input data consisting of *N* complex values. The buffer is 8*N* bytes long.

To prevent overflow within the algorithm, the real and imaginary parts of the complex input data values referenced by *in* must be sign extended 16-bit quantities and must be between –32768 and +32767.

out is an initialized pointer to the starting address of the output data buffer which must reference at least as many `Complex` data items as *in* references input data points. The *out* parameter must also be at least 8*N* bytes long.

When the function returns, *out* contains a pointer to the starting address of the buffer that holds the *N* outputs.

logN is an integer that defines the base-2 logarithm of the number of complex data input values, *N*, that are referenced by *in*. *N* defines the number of points in the FFT.

direction is a flag that indicates the direction of the FFT:

`1` the forward FFT is to be performed.

`0` the inverse FFT is to be performed (the data is to be inverted to its original state).

Return Value

`0` the FFT was successful and `out` points to the buffer containing valid FFT transform coefficients.

`1` the FFT was unsuccessful because the trigonometry table is not large enough for the given size of *N*. The values in the output data buffer pointed to by `out` are unspecified.

Usage

The number of data inputs that are referenced by `in` and the number of outputs that are returned by the FFT function can be determined from the value given by `logN`. The value defined by `logN` must be the power to which 2 is raised to give the number of inputs. That is, `logN` is the base-2 logarithmic value of N, and is equivalent to the highest bit that is set in *N*, such that *N* = (1 << `logN`).

N must be a power of 2. The largest value of *N* that can be passed to the FFT depends on the size of the trigonometry table that was used when the FFT was assembled. For the optimized version of the FFT, the smallest value of *N* that can be used is 16, such that `logN` is at least 4.

Notes

The output data is scaled by a factor of $1/N$ to reduce the possibility of overflow during the FFT algorithm.

The input and output buffers can be in-place or out-of-place. However, if memory is slow on the host system, the FFT calculations are slightly faster when performed in-place.

4.3.2 REALFFT

This function calculates the forward FFT for a given set of real data values.

Syntax

```
int REALFFT(int *in, Complex *out, int logN)
```

where:

in is a pointer to the starting address of the input data consisting of 2*N* real values. The buffer is 8*N* bytes long.

To prevent overflow within the algorithm, the real input data values referenced by *in* must be sign extended 16-bit quantities and must be between –32768 and +32767.

out is an initialized pointer to the starting address of the output data buffer, which must reference at least as many `Complex` data items as half the number of real input data points referenced by *in*. Therefore, *out* must be at least 8*N* bytes long.

When the function returns, *out* contains a pointer to the starting address of the buffer that holds the *N* outputs.

logN is an integer that defines the base-2 logarithm of the number of real input data values referenced by *in*, divided by two. Therefore, *logN* defines half the number of points in the FFT and the number of outputs that are returned by `REALFFT()`.

Return Value

`0` the FFT was successful and *out* points to the buffer containing valid FFT transform coefficients.

`1` the FFT was unsuccessful because the trigonometry table is not large enough for the given size of *N*. The values in the output data buffer pointed to by *out* are unspecified.

Usage

If the values to be fast Fourier transformed are real values (complex numbers with a zero-value imaginary part), `REALFFT()` can be used instead of `FFT()` to perform the forward FFT and almost double the speed of the transform.

The FFT of $2N$ real values is calculated using an N-point complex FFT. Therefore, the size of the input buffer referenced by `in` must be at least $2N$ and filled with the real parts of:

$$x[0], \ldots, x[2N-1]$$

After performing the real FFT, the output buffer referenced by `out` contains the first half of the FFT transform coefficients, N complex values given by:

$$X[0], \ldots, X[N-1]$$

The second half can be calculated by symmetry because:

$X[2N\text{-}k]$

is the complex conjugate of:

$X[k]$

in the real case.

The number of data inputs that are referenced by `in` and the number of outputs that are returned by the FFT function can be determined from the value given by `logN`. The value defined by `logN` must be the power to which 2 is raised to give the number of outputs and, therefore, half the number of inputs. In other words, `logN` is the base-2 logarithmic value of N and is equivalent to the highest bit that is set in N, such that N = (1 << `logN`) and the number of inputs is given as $2N$.

N must be a power of 2. The largest value of `N` that can be passed to the FFT depends on the size of the trigonometry table that was used when the FFT was assembled. In other words, for each N the trigonometry table must be at least of size $2N$. For the optimized version of the FFT, the smallest value of N that can be used is 16, such that `logN` is at least 4, and therefore the minimum number of points in the real FFT is 32.

Notes

The input and output buffers are both $8N$ bytes and can be in-place or out-of-place. However, if memory is slow on the host system, the FFT calculations are slightly faster when performed in-place.

4.3.3 GenerateWindow

This function generates an array of coefficients that describe either a Hamming or Hanning window to be performed on data that is to be passed to the FFT. The array can then be multiplied by the input data pre-FFT to perform the window.

Syntax

```
int *GenerateWindow(int windowCoefficient, int nDataPoints)
```

where:

windowCoefficient

is the fixed-point coefficient value, α, with the fixed-point between bits 13 and 14. This value determines whether the generated window coefficients are Hamming or Hanning window coefficients. The *windowCoefficient* parameter must be either 0x1d70 for a Hamming Window or 0x2000 for a Hanning Window.

nDataPoints

is the number of points in the Hamming or Hanning Window. This is usually the number of data points to be input to the FFT.

Return Value

An array of *nDataPoints* sign-extended, fixed-point windowing coefficients, with the fixed-point between bits 13 and 14, or NULL if an error occurred.

Usage

The `GenerateWindow()` function should be used if the required window is to be performed on several sets of input data. Otherwise, `HammingWindow()` or `HanningWindow()` should be used.

4.3.4 HammingWindow

This function multiplies the input data to be passed to the FFT by the Hamming Window coefficients.

Syntax

```
void HammingWindow(int outputs[], int inputs[], int nDataPoints)
```

where:

outputs is an initialized array that must reference at least *nDataPoints* entries.

When the function returns, *outputs* is an array that contains *nDataPoints* data values, which are the result of multiplying the input data values by the Hamming Window.

inputs is an array of *nDataPoints* data values to be multiplied by the Hamming Window. The values must be no more than 16-bit quantities. Otherwise, the multiplication by the Hamming Window overflows.

nDataPoints is the number of data points in *inputs* and the number of entries in *outputs*, after performing the Hamming Window.

Usage

The input data array and the output data array can be in-place. That is, the result of the multiplication by the Hamming Window can be returned in the same memory as the input data so that the input data is overwritten.

The `HammingWindow()` function should be used if the window is to be performed on the given set of input data only. If the window is to be applied to several sets of data values, each with the same number of data points, `GenerateWindow()` should be used instead to obtain the Hamming Window coefficients, and the multiplication should be carried out separately.

4.3.5 HanningWindow

This function multiplies the input data to be passed to the FFT by the Hanning Window coefficients.

Syntax

```
void HanningWindow(int outputs[], int inputs[], int nDataPoints)
```

where:

outputs is an initialized array that must reference at least *nDataPoints* entries.

When the function returns, *outputs* is an array that contains *nDataPoints* data values, which are the result of multiplying the input data values by the Hanning Window.

inputs is an array of *nDataPoints* data values to be multiplied by the Hanning Window. The values must be no more than 16-bit quantities. Otherwise, the multiplication by the Hanning Window overflows.

nDataPoints is the number of data points in *inputs* and the number of entries in *outputs*, after performing the Hanning Window.

Usage

The input data array and the output data array can be in-place. That is, the result of the multiplication by the Hanning Window can be returned in the same memory as the input data so that the input data is overwritten.

The `HanningWindow()` function should be used if the window is to be performed on the given set of input data only. If the window is to be applied to several sets of data values, each with the same number of data points, `GenerateWindow()` should be used instead to obtain the Hanning Window coefficients, and the multiplication should be carried out separately.

Chapter 5
Two-Dimensional Discrete Cosine Transform

This chapter describes an ARM implementation of a *two-dimensional* (2D) *discrete cosine transform* (DCT). This chapter contains the following sections:

5.1 Overview

This section provides general information on the 2D DCT.

5.1.1 ARM architecture requirements

The file `dcts.s` defines the `FASTMUL` flag. This flag determines whether the DCT algorithms use the ARM multiply instruction, or addition and shifting to achieve the multiplication.

On older processors, where addition and shifting is a faster process than the multiply instruction, `FASTMUL` should be set to 0 (false). On newer processors, where multiplication is fast, `FASTMUL` should be set to 1 (true).

Table 5-1 contains a list of ARM processors and the recommended settings of the `FASTMUL` flag.

Table 5-1 FASTMUL flag settings

Processor	FASTMUL
ARM7	0
ARM7TDMI	0 or 1
ARM9TDMI	1
SA-110	1

5.1.2 Implementation

The 2D DCT is implemented in a block form as a *one-dimensional* (1D) 8-element DCT horizontally on every row, followed by a 1D 8-element DCT vertically on every column.

1D 8-element forward DCT

The 1D 8-element forward DCT takes eight inputs and produces eight outputs. It can be described by the formula:

$$C(u)(f(0)c(u,0)+\ldots+f(7)c(u,7)) \qquad 0\le u\le 7$$

where:

$$C(u)=\begin{cases}\dfrac{1}{2\sqrt{2}} & u=0\\ \dfrac{1}{2} & 1\le u\le 7\end{cases}$$

and:

$$c(u,x) = \cos\left[\frac{\pi}{16}u(2x+1)\right]$$

The 1D DCT implementation devised by Arai, Agui and Nakajima is the most efficient method known to date. Their method can be split into five stages:

Stage 1:

$$g(0)=f(0)+f(7)$$
$$g(1)=f(1)+f(6)$$
$$g(2)=f(2)+f(5)$$
$$g(3)=f(3)+f(4)$$
$$g(4)=f(3)-f(4)$$
$$g(5)=f(2)-f(5)$$
$$g(6)=f(1)-f(6)$$
$$g(7)=f(0)-f(7)$$

Stage 2:

$$f(0) = g(0) + g(3)$$
$$f(3) = g(0) - g(3)$$

$$f(1) = g(1) + g(2)$$
$$f(2) = g(1) - g(2)$$

$$f(4) = g(4) + g(5)$$
$$f(5) = g(5) + g(6)$$
$$f(6) = g(6) + g(7)$$
$$f(7) = g(7)$$

Stage 3:

$$g(0) = f(0)$$
$$g(1) = f(1)$$

$$g(2) = (f(2) + f(3)) \times \frac{1}{\sqrt{2}}$$
$$g(3) = f(3)$$

$$temp = (f(4) - f(6)) \times \cos\left(\frac{3\pi}{8}\right)$$
$$g(4) = temp + f(4) \times \left(\cos\left(\frac{\pi}{8}\right) - \cos\left(\frac{3\pi}{8}\right)\right)$$
$$g(6) = temp + f(6) \times \left(\cos\left(\frac{\pi}{8}\right) + \cos\left(\frac{3\pi}{8}\right)\right)$$

$$temp = f(5) \times \frac{1}{\sqrt{2}}$$
$$g(5) = f(7) + temp$$
$$g(7) = f(7) - temp$$

Stage 4:

$$F(0) = g(0) + g(1)$$
$$F(4) = g(0) - g(1)$$

$$F(2) = g(3) + g(2)$$
$$F(6) = g(3) - g(2)$$

$$F(5) = g(7) + g(4)$$
$$F(3) = g(7) - g(4)$$

$$F(1) = g(5) + g(6)$$
$$F(7) = g(5) - g(6)$$

Stage 5:

$$T(0) = \frac{F(0)}{2}$$
$$T(u) = \frac{F(u)}{2\sqrt{2}\cos\left(\frac{\pi u}{16}\right)} \quad 1 \le u \le 7$$

1D 8-element reverse DCT

The reverse DCT algorithm is calculated by reversing each step of the above algorithm. For example:

$$F(1) = g(5) + g(6)$$
$$F(7) = g(5) - g(6)$$

from stage 4 are reversed as:

$$g(5) = \frac{F(1) + F(7)}{2}$$
$$g(6) = \frac{F(1) - F(7)}{2}$$

The result is a reverse DCT that is almost lossless while only using 16-bit arithmetic.

2D 8x8 element DCT

The 2D 8x8 element forward DCT takes as input a 2D 8x8 array *f*(*i*,*j*) and produces as output a new 8x8 array *T*(*u*,*v*) described by the formula:

$$T(u,v) = C(u)C(v)\sum_{i=0}^{7}\sum_{i=0}^{7} f(i,j)\,c(u,i)\,c(v,j)$$

where *C* and *c* are as defined in *1D 8-element forward DCT* on page 5-3.

The formula is separable. That is, the 2D transform can be calculated by performing a 1D transform on each of the eight rows and eight columns of the matrix. The order in which the rows and columns are transformed does not affect the final result. The algorithm implemented here calculates the 2D DCT in this way.

5.1.3 Files

The files in Table 5-2 are provided with the implementation.

Table 5-2 2D/DCT files

Filename	Archive name	Code type	Functionality
`dcts.s`	`arm_dct`	ARM assembly language	2D DCT coding and decoding
`dcts.h`	`arm_dct`	C macros, C header	DCT setup macros, function prototypes and coding/decoding tables
`dcttgenc.c`	`arm_dct\dct_tgen`	C	Coding and decoding table generation
`dcttgenc.h`	`arm_dct\dct_tgen`	C header	Coding and decoding table constant declarations

5.2 SCALETABLE data structure

The `SCALETABLE` data type is used to store scaling table and re-ordering information required during the DCT operations.

5.2.1 Definition

```
typedef unsigned int SCALETABLE[ 64 ] ;
```

5.2.2 Description

The following sections describe how arrays of type `SCALETABLE` are used by the 2D DCT code for forward and reverse transforms:

- *Forward 2D DCT*
- *Reverse 2D DCT* on page 5-8.

Forward 2D DCT

The first part of the forward 2D DCT code applies stages 1 to 4 of the 1D DCT algorithm (as described in *1D 8-element forward DCT* on page 5-3) to all rows and columns of the input data. In the application of stage 4, the data is not reordered, and so columns and rows remain permuted in the order 0, 4, 2, 6, 5, 3, 1, 7, as defined in stage 4.

The second part of the forward 2D DCT code scales the data as described in stage 5 of *1D 8-element forward DCT* on page 5-3 and carries out the permutation remaining from stage 4, to produce the final result. To do this, it uses the `FDCTScales` array. This array is of type `SCALETABLE`. It contains 64 entries, each entry consisting of a 32-bit word that contains the permutation reorder information in the top eight bits, and the scaling coefficients in the bottom 24 bits. The table is defined in the file `dcts.h`.

The `FDCTScales` table is derived from the `AANScales` matrix defined in `dcttgenc.h`. `AANScales` defines the scalings required by stage 5 of *1D 8-element forward DCT* on page 5-3 and is calculated by the formula:

$$AANScales(i, j) = t(i)t(j) \times 2^{14}$$

where:

$$t(i) = \begin{cases} 1 & i = 0 \\ \sqrt{2}\cos\left(\dfrac{\pi}{16}i\right) & 1 \le i \le 7 \end{cases}$$

t(j) is defined in the same way as *t(i)*.

the multiplication by 2^{14} shifts the decimal values produced by 14 bits to give integer values with precision.

The maximum shift possible to give the greatest precision in the resulting integer values is 14 bits. This value ensures that the multiplications do not overflow the 16 bits allocated for each value. For example:

$$AANScales(2,5) = 32768 \times \cos\left(\frac{\pi}{8}\right) \times \cos\left(\frac{5\pi}{16}\right) \approx 16819$$

The complete constant value `AANScales` matrix is given as:

$$AANScales = \begin{bmatrix} 16384 & 22725 & 21407 & 19266 & 16384 & 12873 & 8867 & 4520 \\ 22725 & 31521 & 29692 & 26722 & 22725 & 17855 & 12299 & 6270 \\ 21407 & 29692 & 27969 & 25172 & 21407 & 16819 & 11585 & 5906 \\ 19266 & 26722 & 25172 & 22654 & 19266 & 15137 & 10426 & 5315 \\ 16384 & 22725 & 21407 & 19266 & 16384 & 12873 & 8867 & 4520 \\ 12873 & 17855 & 16819 & 15137 & 12873 & 10114 & 6967 & 3552 \\ 8867 & 12299 & 11585 & 10426 & 8867 & 6967 & 4799 & 2446 \\ 4520 & 6270 & 5906 & 5315 & 4520 & 3552 & 2446 & 1247 \end{bmatrix}$$

Reverse 2D DCT

The reverse 2D DCT algorithm is split into the following two parts:

- Part one scales and permutes the data array using the `RDCTScales` table, of type `SCALETABLE`. This part is performed by the `PRERDCT` macro.
- Part two inverts stages 1 to 4 of *1D 8-element forward DCT* on page 5-3 on each row and column of the matrix. This part is performed by the `rdct_fast()` function.

The `RDCTScales` table is generated from the `AANScales` matrix.

Details on the calculation of `FDCTScales` and `RDCTScales` from `AANScales` is not included in this text. The source file `dcttgenc.c` generates the tables.

5.3 Functions

This section describes the DCT functions. Both functions require special variable initialization and custom data types to be used. A set of macros that can be used to perform the initialization are described in *Supplementary macros* on page 5-13.

The DCT functions are:

- Forward 2D DCT on 8x8 blocks (*fdct_fast*)
- Reverse 2D DCT on 8x8 blocks (*rdct_fast* on page 5-11).

5.3.1 fdct_fast

This function performs the forward DCT on pairs of interleaved blocks, with one block of a pair stored in the low 16 bits of the integers and the other block of the pair in the high 16 bits.

Syntax

```
void fdct_fast(int *dctBlock, int nBlocks, SCALETABLE *sTable[])
```

where:

dctBlock is a 256 byte-aligned pointer to the start of a sequence of 8x8 integer block pairs that hold the data that is to be transformed using the forward DCT. Adding multiples of 256 bytes to the starting address of *dctBlock* accesses additional pairs. The address of the pointer must have the low eight bits set to zero.

For each block of a block pair, the respective 16 bits of each integer must contain an unsigned 8-bit value in the range from 0-255 shifted up by one bit to provide fixed-point precision.

When this function returns, *dctBlock* holds the output data.

nBlocks is the number of blocks that are defined in *dctBlock*, both on input and output. This parameter must be the total number of blocks that contain data and not the number of block pairs.

sTable is an array of pointers to the SCALETABLE information. This information is used to perform the scaling and re-ordering of the calculated DCT values before returning from the function call, so that a map from an input value to its respective output value is possible.

For every block pair being transformed by the forward DCT, a pointer to a SCALETABLE is required that contains the information used to perform the scaling and re-ordering. There must be at least (*nBlocks* + 1) / 2 array entries, each a pointer to a SCALETABLE.

Usage

The *dctBlock* pointer must address enough memory to hold all block pairs plus enough memory for an additional block pair that is used as scratch space during calculations. Each block pair requires 256 bytes (64 integers of four bytes each).

This function can operate over an odd number of blocks, but during calculations *nBlocks* is rounded up to the nearest multiple of two. If an odd number of blocks is used, the last block must still be given in a block pair, with the top 16 bits of each integer in the block zeroed so that it does not contain any data.

For example, to perform six forward DCTs at a time, requiring three block pairs for the data and an extra block pair for scratch space (1KB of workspace), the *dctBlock* parameter can be viewed as shown in Figure 5-1.

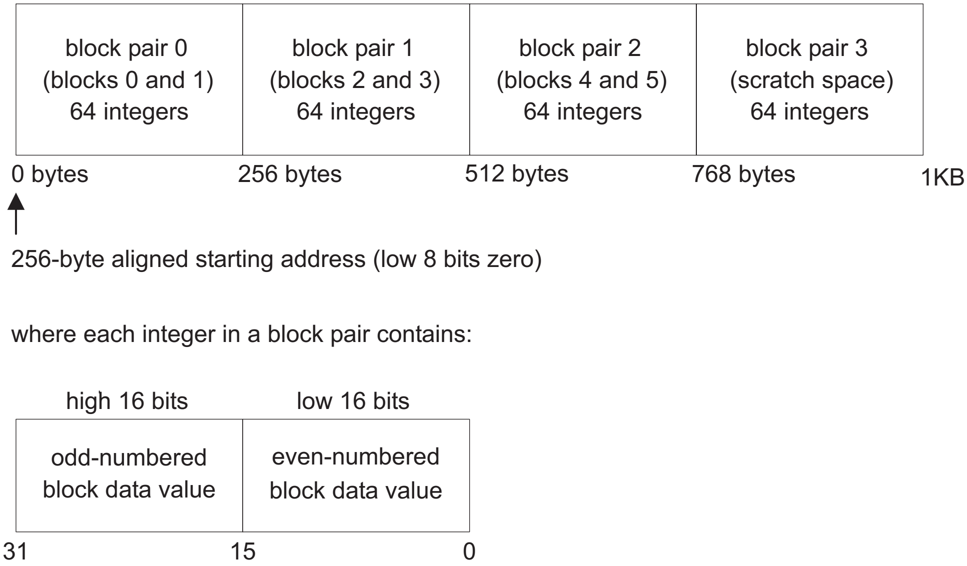

Figure 5-1 dctBlock parameter

See also

For information on setting up *sTable* see *CREATEFDCTSTABLEARRAY* on page 5-13.

For information on initializing *dctBlock* see *CREATEDCTBLOCK* on page 5-15.

For information on adding data to *dctBlock* see *PREFDCT* on page 5-16. For information on retrieving data from *dctblock* see *POSTFDCT* on page 5-17.

5.3.2 rdct_fast

This function performs the reverse DCT on pairs of interleaved blocks, with one block of a pair stored in the low 16 bits of the integers, and the other block of the pair in the high 16 bits. The data in the blocks is data that has been output from a forward DCT algorithm, such as `fdct_fast()`.

Syntax

```
void rdct_fast(int *dctBlock, int nBlocks)
```

where:

dctBlock is a 256 byte-aligned pointer to the start of a sequence of 8x8 integer block pairs that hold the data that is to be transformed using the reverse DCT. Adding multiples of 256 bytes to the starting address of *dctBlock* accesses additional pairs. The address of the pointer must have the low eight bits set to zero.

The data to be transformed using the reverse DCT must start in the block pair referenced by *dctBlock* + 256. The first block pair referenced by *dctBlock* must not contain data because it is used as scratch space during the reverse DCT calculations.

When this function returns, *dctBlock* holds the output data. This output data is the original data that was passed to a forward DCT and starts with the first block referenced by *dctBlock* and not *dctBlock* + 256.

nBlocks is the number of blocks that are defined in *dctBlock*, both on input and output. This parameter must be the total number of blocks that contain data and not the number of block pairs.

Usage

The *dctBlock* pointer must address enough memory to hold all block pairs plus enough memory for an additional block pair that is used as scratch space during calculations. Each block pair requires 256 bytes (64 integers of four bytes each).

This function can operate over an odd number of blocks, but during calculations *nBlocks* is rounded up to the nearest multiple of two. If an odd number of blocks is used, the last block must still be given in a block pair, with the top 16 bits of each integer in the block zeroed so that it does not contain any data.

For example, to perform six reverse DCTs at a time, requiring three block pairs for the data and an extra block pair for scratch space (1KB of workspace), the *dctBlock* parameter can be viewed as shown in Figure 5-1 on page 5-10.

Limitations on input values

Only data that was generated by a forward DCT, such as `fdct_fast()`, should be passed to the reverse DCT operation. However, output data from a forward DCT requires re-ordering and shifting before it can be placed into *`dctBlock`* and passed through to the reverse DCT procedure.

Each value must be shifted up by three bits to allow for a fixed-point precision. The re-ordering is the reverse operation to that which was performed in a call to `fdct_fast()`, which sorted the data into a form suitable for output. The reverse operation is performed externally to the reverse DCT call because the values produced by the forward DCT are normally entropy encoded and the operation of re-ordering required is normally absorbed into the entropy decoding before passing the data to the reverse DCT. If the re-ordering is not absorbed into an entropy decoding algorithm, it must be performed separately before the data is passed to the reverse DCT.

See also

For information on initializing *`dctBlock`*, see *CREATEDCTBLOCK* on page 5-15.

For information on adding data to *`dctBlock`* see *PRERDCT* on page 5-18. For retrieving data from *`dctblock`* see *POSTRDCT* on page 5-20.

5.4 Supplementary macros

This section describes macros that can be used to initialize variables and prepare data for the DCT function calls.

For information on the DCT functions, see *Functions* on page 5-9.

The supplementary macros are:

- Create an array of pointers to `SCALETABLE` data (*CREATEFDCTSTABLEARRAY*)
- Create a 256-byte aligned pointer to 8x8 block pairs (*CREATEDCTBLOCK* on page 5-15)
- Add data to an array for `fdct_fast()`(*PREFDCT* on page 5-16)
- Extract data processed by `fdct_fast()`(*POSTFDCT* on page 5-17)
- Add data to an array for `rdct_fast()`(*PRERDCT* on page 5-18)
- Extract data processed by `rdct_fast()`(*POSTRDCT* on page 5-20).

5.4.1 CREATEFDCTSTABLEARRAY

This macro creates an array of pointers to `SCALETABLE` data from the number of blocks to be passed through to each call of the forward DCT (not the number of block pairs). The array is the third parameter required by `fdct_fast()`.

Syntax

`CREATEFDCTSTABLEARRAY(`*`FDCTSTABLEARRAYPTRS`*`, `*`NUMBERBLOCKS`*`)`

where:

`FDCTSTABLEARRAYPTRS`

is an uninitialized pointer to a pointer to `SCALETABLE` data. This parameter should not reference memory because it is initialized by the macro to reference the memory that is allocated by the macro.

On output, *`FDCTSTABLEARRAYPTRS`* contains an array of (*`NUMBERBLOCKS`* + 1) / 2 pointers to *`SCALETABLE`* data. Each array entry contains a pointer to the `FDCTScales` table.

`NUMBERBLOCKS`

is the number of blocks to be passed through to each forward DCT call.

Usage

The macro allocates enough memory for (*NUMBERBLOCKS* + 1) / 2 entries in the array, that is, one entry for each block pair that is passed to the forward DCT. Each pointer in the array references the `FDCTScales` table. The table is required to scale and re-order the data in the blocks during the forward DCT call.

FDCTSTABLEARRAYPTRS can be passed as the third parameter to each call of `fdct_fast()`. The memory referenced by *FDCTSTABLEARRAYPTRS* should be freed when it is no longer required.

5.4.2 CREATEDCTBLOCK

This macro creates a 256-byte aligned pointer to a start of a sequence of 8x8 integer block pairs that can be passed to either the forward or reverse DCT. Enough space is allocated to hold the data for each block, in addition to the space required by the DCT operations.

Syntax

```
CREATEDCTBLOCK(DCTBLOCKPTR, DCTBLOCK, NUMBERBLOCKS)
```

where:

DCTBLOCKPTR is an unallocated pointer to an integer. This parameter should not reference memory because it is initialized by the macro to reference memory allocated by the macro.

On output, DCTBLOCKPTR contains a pointer to sufficient memory to hold (*NUMBERBLOCKS* + 1) / 2 + 2 block pairs. Each block pair is 256 bytes (64 integers of four bytes each).

Memory for two additional block pairs is allocated:

- one to be used as scratch space during the DCT calls
- one to allow for the adjustment of the address pointed to by *DCTBLOCKPTR* to create a 256-byte aligned pointer referenced by *DCTBLOCK*.

DCTBLOCKPTR references the original memory address returned by the call to initialize the memory, and can be used to free the memory when it is no longer required.

DCTBLOCK is an unallocated pointer to an integer. This parameter should not reference any memory because it is initialized by the macro to reference memory allocated by the macro.

On output, *DCTBLOCK* contains a 256-byte aligned pointer to enough memory to hold (*NUMBERBLOCKS* + 1) / 2 + 1 block pairs. Each block pair is 256 bytes (64 integers of four bytes each). This is the 256-byte aligned version of *DCTBLOCKPTR* with the low eight bits of the starting address for the pointer set to zero.

This pointer can be passed to either the forward or reverse DCT.

DCTBLOCK cannot be used to free the memory when it is no longer required, because it does not reference all the memory that was allocated during the macro execution.

NUMBERBLOCKS is the number of blocks of data required, to be referenced by *DCTBLOCK* when allocated.

5.4.3 PREFDCT

This macro adds the current integer value of the data to be forward DCT'd in the correct position and format to a data array to be passed to `fdct_fast()`.

Syntax

`PREFDCT(DCTBLOCK, INTVALUE, X, Y, BCOUNTER)`

where:

DCTBLOCK is a pointer to the start of a sequence of 8x8 integer block pairs.

On output, *DCTBLOCK* contains a pointer to the start of a sequence of 8x8 integer block pairs that hold the data to be passed to `fdct_fast()`, including *INTVALUE* stored in block *BCOUNTER* at position (*X*,*Y*).

INTVALUE is the current integer value to be added into *DCTBLOCK*. *INTVALUE* must be an unsigned character in the range 0-255.

X is the x (column) coordinate into the current block where *INTVALUE* is to be stored.

Y is the y (row) coordinate into the current block where *INTVALUE* is to be stored.

BCOUNTER is the current block number in the sequence of blocks in *DCTBLOCK* where *INTVALUE* is to be added.

Usage

BCOUNTER provides the offset into *DCTBLOCK* for the current block that data is being added. *X* and *Y* provide the location within that block where the current value must be stored.

Notes

This macro adds the current integer value into the low 16 bits of the integer in the current block, if the value of *BCOUNTER* is divisible by 2. If the value of *BCOUNTER* is not divisible by 2, the value is added into the high 16 bits.

Because blocks are paired, the data is stored in the block referenced by *BCOUNTER* / 2.

INTVALUE is shifted by one bit to allow for a fixed-point precision before being stored in the block in either the low or high 16 bits.

5.4.4 POSTFDCT

This macro extracts a value from a data array that was processed by `fdct_fast()`.

Syntax

`POSTFDCT(DCTBLOCK, INTVALUE, X, Y, BCOUNTER)`

where:

DCTBLOCK is a pointer to the start of a sequence of 8x8 integer block pairs that hold data generated by `fdct_fast()`.

INTVALUE is an integer to store the current value read from *DCTBLOCK*.

On output, *INTVALUE* contains the integer read from block *BCOUNTER* in *DCTBLOCK*, at position (*X*,*Y*).

X is the x (column) coordinate into the current block for the integer value to retrieve.

Y is the y (row) coordinate into the current block for the integer value to retrieve.

BCOUNTER is the current block number in the sequence of blocks in *DCTBLOCK* from which the integer value is to be retrieved.

Usage

BCOUNTER provides the offset into *DCTBLOCK* for the current block from which the data is being retrieved. *X* and *Y* provide the location within that block from which the current value must be retrieved.

Notes

This macro extracts the current integer value from the low 16 bits of the integer in the current block if the value of *BCOUNTER* is divisible by 2. If the value of *BCOUNTER* is not divisible by 2, the value is extracted from the high 16 bits.

Because blocks are paired, the data is retrieved from the block referenced by *BCOUNTER* / 2.

5.4.5 PRERDCT

This macro adds the current integer value of the data to be reversed DCT'd in the correct position and format to a data array to be passed to `rdct_fast()`.

Syntax

`PRERDCT(DCTBLOCK, INTVALUE, X, Y, BCOUNTER)`

where:

DCTBLOCK is a pointer to the start of a sequence of 8x8 integer block pairs.

On output, *DCTBLOCK* contains a pointer to the start of a sequence of 8x8 integer block pairs that hold the data to be passed to `rdct_fast()`, including *INTVALUE* stored in block *BCOUNTER* at position (*X*,*Y*).

INTVALUE is the current integer value to be added into *DCTBLOCK*. The value must be data that has been output from a call to `fdct_fast()`.

X is the x (column) coordinate into the current block where *INTVALUE* is to be stored.

Y is the y (row) coordinate into the current block where *INTVALUE* is to be stored.

BCOUNTER is the current block number in the sequence of blocks in *DCTBLOCK* where *INTVALUE* is to be added.

Usage

BCOUNTER provides the offset into *DCTBLOCK* for the current block to which the data is being added. *X* and *Y* provide the location within that block where the current value must be stored.

Notes

This macro adds the current integer value into the low 16 bits of the integer in the current block, if the value of *BCOUNTER* is divisible by 2. If the value of *BCOUNTER* is not divisible by 2, the value is added into the high 16 bits.

Because blocks are paired, the data is stored in the block referenced by *BCOUNTER* / 2.

INTVALUE is re-ordered and shifted by three bits to allow for a fixed-point precision rather than being stored in the block in either the low or high 16 bits. The re-ordering is the reverse operation to that performed in a call to `fdct_fast()`, which sorts the data into a form suitable for output. The reverse operation is performed externally to the reverse

DCT call because the values produced by a forward DCT are normally entropy encoded and then the operation of re-ordering performed here is normally absorbed into the entropy decoding before passing the data to the reverse DCT.

5.4.6 POSTRDCT

This macro extracts a value from a data array that was processed by `rdct_fast()`. This is the original data before the call to a forward DCT, such as `fdct_fast()`.

Syntax

`POSTRDCT(DCTBLOCK, INTVALUE, X, Y, BCOUNTER)`

where:

DCTBLOCK is a pointer to the start of a sequence of 8x8 integer block pairs that hold data generated by `rdct_fast()`.

INTVALUE is an integer to store the current value read from *DCTBLOCK*.

On output, *INTVALUE* contains the integer read from block *BCOUNTER* in *DCTBLOCK*, at position (*X*,*Y*). The integer is shifted down by seven bits to remove the fixed-point precision. The macro adds 128 to the value because each output value from the reverse DCT is in the range from –128 to 127.

X is the x (column) coordinate into the current block for the integer value to retrieve.

Y is the y (row) coordinate into the current block for the integer value to retrieve.

BCOUNTER is the current block number in the sequence of blocks in *DCTBLOCK* from which the integer value is to be retrieved.

Usage

BCOUNTER provides the offset into *DCTBLOCK* for the current block from which the data is being retrieved. *X* and *Y* provide the location within that block from which the current value must be retrieved.

Notes

This macro extracts the current integer value from the low 16 bits of the integer in the current block if the value of *BCOUNTER* is divisible by 2. If the value of *BCOUNTER* is not divisible by 2, the value is extracted from the high 16 bits.

Because blocks are paired, the data is retrieved from the block referenced by `BCOUNTER / 2`.

Chapter 6
Huffman and Bit Coding/Decoding

This chapter describes an implementation of a Huffman *coder/decoder* (codec) that uses a general bit codec based on lookup tables. It contains the following sections:

6.1 Overview

This section provides general information on the Huffman and bit coding/decoding.

6.1.1 Implementation

Huffman coding compresses data using the probability of a symbol appearing in the data to be coded. A symbol with a high probability of occurrence is coded using fewer bits than a symbol with a low probability of occurrence. Given the probabilities, the frequency of occurrence for each length of codeword can be determined.

Variable length codewords, with a unique prefix attribute, can be generated from the symbols of the data to be coded. These codewords are sorted into increasing order of frequency of occurrence, together with the frequency of occurrence for each length of codeword. This information is all that is required to perform Huffman coding and decoding, and must be stored with a Huffman encoded stream to enable the stream to be uniquely decoded.

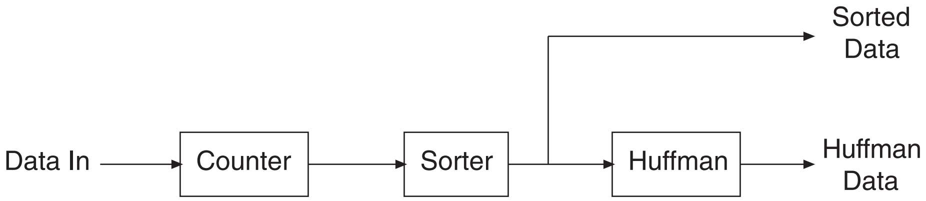

Figure 6-1 Huffman coding

The functionality of the blocks in Figure 6-1 is as follows:

Counter counts the frequency of occurrence for each symbol in the data and determines the maximum symbol value in the data.

Sorter sorts the symbols and frequency of occurrence for each symbol into increasing order of frequency.

Huffman generates the frequency of occurrence for each length of codeword.

Encoder

Encoding is performed as shown in Figure 6-2.

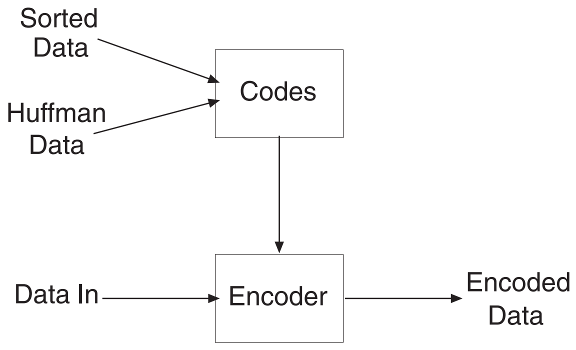

Figure 6-2 Encoding

The functionality of the blocks is described as follows:

Codes generates the symbol-to-codeword lookup table.

Encoder encodes the fixed size input data to variable size codewords.

Decoder

Decoding is performed as shown in Figure 6-3.

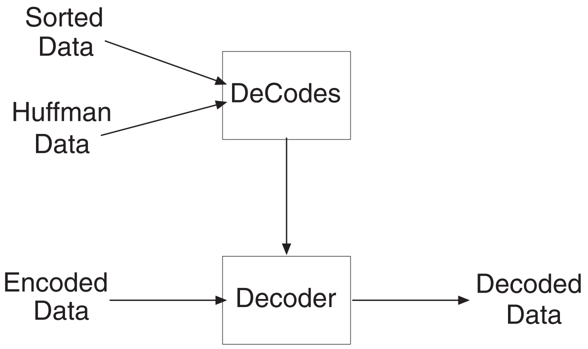

Figure 6-3 Decoding

The functionality of the blocks is described as follows:

DeCodes generates the codeword-to-symbol lookup tables.

Decoder decodes the variable size codewords to fixed size output data.

Table lookup

The encoder and decoder use lookup tables to convert symbols to and from codewords. The maximum length of a codeword is 27 bits. Restricting codewords to this length enables a codeword of 27 bits to be packed into a 32-bit integer word, with the length of the codeword packed in the remaining five bits (the minimum number of bits required for a value of 27). It is not possible to have a 28-bit codeword because 33 bits would be required.

In practice, the maximum codeword length of 27 bits is not a limitation, because Huffman codewords are shorter for symbols with a higher probability of occurrence and a codeword of length 27 has a very small probability of occurrence. An increase of one bit in the maximum codeword length results in a negligible increase in compression, at the expense of increasing the overhead of computation because a codeword and its length cannot be packed into a 32-bit integer word.

For example, if the data to be coded has a Fibonacci sequence for the frequency of occurrence of the symbols, this gives the highest probability of occurrence for each length of codeword. The probability of a 27-bit codeword occurring is:

$$1/1346268 \approx 7.4 \times 10^{-7}$$

The probability of a 28-bit code word occurring is:

$$1/2178308 \approx 4.6 \times 10^{-7}$$

6.1.2 Files

The files in Table 6-1 are provided in the implementation.

Table 6-1 Huffman and codec files

Filename	Archive name	Code type	Functionality
huffmanc.c	arm_huff	C	Generation of an array of frequency of occurrence for each length of codeword
huffmanc.h	arm_huff	C header	Huffman function prototype and constant definitions
makctm.h	arm_huff	C macro	Generic function to generate the symbol-to-codeword lookup table for bit coding
makct8c.c, makct16c.c, makct32c.c	arm_huff	C	Generation of symbol-to-codeword lookup table for bit coding 8-bit, 16-bit, and 32-bit data, respectively
makctc.h	arm_huff	C header	Generic symbol-to-codeword table generation function prototype
makct8c.h, makct16c.h, makct32c.h	arm_huff	C header	Symbol-to-codeword table generation function prototypes for 8-bit, 16-bit, and 32-bit data, respectively
mkdctm.h	arm_huff	C macro	Generic function to generate the codeword-to-symbol lookup tables for bit decoding
mkdct8c.c, mkdct16c.c, mkdct32c.c	arm_huff	C	Generation of codeword-to-symbol lookup tables for bit decoding to 8-bit, 16-bit, and 32-bit data, respectively
mkdctc.h	arm_huff	C header	Generic codeword-to-symbol tables generation function prototype
mkdct8c.h, mkdct16c.h, mkdct32c.h	arm_huff	C header	Codeword-to-symbol tables generation function prototypes for 8-bit, 16-bit, or 32-bit data, respectively
bitcodes.s	arm_huff	ARM assembly language	Bit code 8-bit, 16-bit, or 32-bit data to variable length codewords
bitcodes.h	arm_huff	C header	Bit coding function prototypes
bitdcods.s	arm_huff	ARM assembly language	Bit decode variable length codewords to 8-bit, 16-bit, or 32-bit data
bitdcods.h	arm_huff	C header	Bit decoding function prototypes

6.2 BitStreamState data structure

This structure is used to maintain the bit-stream and the bit-position reached within the bit stream, between calls to the bit codec functions.

6.2.1 Definition

```
typedef struct   BitStreamState   BitStreamState ;
typedef          BitStreamState   *BitStreamStatePtr ;

struct BitStreamState {
    unsigned char  *bitstreamend ;
    unsigned int   freebits ;
    /* additional elements if required */
} ;
```

where:

`bitstreamend`

is the pointer to the bit-stream. It is the last word-aligned memory address before the last bit-position reached in the stream. Therefore, the address in `bitstreamend` is always within 32 bits of the last bit-position referenced by `freebits`.

`freebits` is the number of bits between `bitstreamend` and the next word-aligned memory address (`bitstreamend` + 4 bytes) that either does not contain coded data or has not been decoded. The memory address of the last bit-position reached in the stream is defined as (the memory address in `bitstreamend` + 32 – `freebits`).

6.2.2 Description

The bit stream of codewords, created by the bit coder and decoded by the bit decoder, is stored, eight bits at a time, as an array of **unsigned char**s. If necessary, codewords can continue over the boundary between two (8-bit) characters.

A pointer to the bit-stream is maintained in the BitStreamState structure, together with a reference to the final bit-position reached within the bit-stream. The bit stream is defined as all the bits from the starting address of the data array to the last bit-position in the last byte in the array that contains the coded data.

The bit stream structure simplifies calling the bit codec functions multiple times. Multiple calls are necessary if there is insufficient memory available to hold all the data for only one call to the codec routine, or if the entire data is not available with each call, for example if reading data from a file. Because the last bit position reached in the stream is saved in the BitStreamState structure between calls, the bit codec functions can continue referencing the bit-stream from the last position accessed by a previous call to a function.

The information in the BitStreamState structure is insufficient to determine the starting address of the bit-stream (for example when initializing the stream address for decoding) because bitstreamend references the last word-aligned memory address for the stream that was reached during a bit codec function call. Therefore, the starting address of the bit-stream must be maintained separately.

The first two elements in the BitStreamState structure must be bitstreamend and freebits, and they must be defined within the structure in that order. However, the structure can contain more than the two elements required by the bit codec functions. For example, the starting address of the stream as a third element and any other elements required by a particular implementation can be maintained in the structure after the definition of freebits, as indicated in the structure in *Definition* on page 6-6.

6.2.3 Usage

Before the first call to the bit codec functions, `BitStreamState` must be initialized as follows:

- `bitstreamend` must reference the word-aligned starting address of the unsigned character array bit stream
- `freebits` must be set to 32, which is the number of bits before the next word-aligned memory address.

If the bit codec functions are to be called using the same bit-stream, `BitStreamState` must be maintained as-is between calls to the bit codec functions.

For example, if the bit coder function is called and coding is limited to 64 bits of coded data, the bit-stream and the values in the `BitStreamState` structure for `bitstreamend` and `freebits` before the bit coder function call might be set as shown in Figure 6-4.

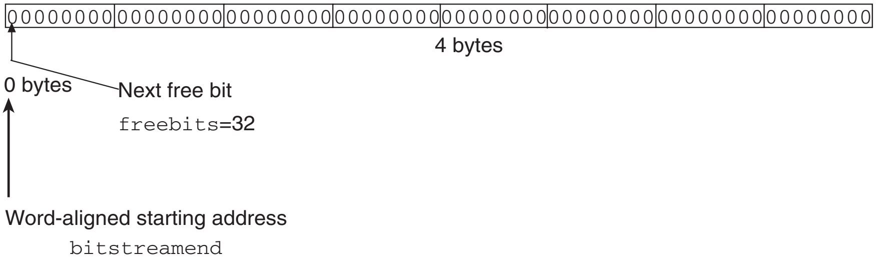

Figure 6-4 Before bit coding

After the bit coder function, if 54 bits of coded data were generated, the coded bit-stream and the values returned in the `BitStreamState` structure for `bitstreamend` and `freebits` might be set as shown in Figure 6-5.

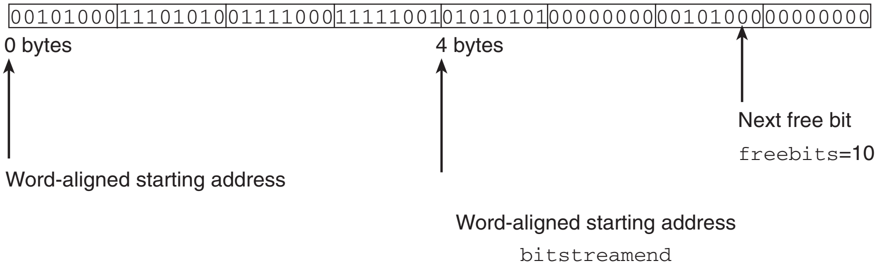

Figure 6-5 After bit coding, 54 bits of coded data

In this example, the bit-stream has been initialized to zero for clarity. However, this is not necessary because both ends of the coded stream are uniquely defined by the starting address of the stream and the values returned in `bitstreamend` and `freebits` after a function call.

6.3 Functions

This section describes the Huffman routines. The functions are:

- Create the array of frequency of occurrence for each length of codeword (*Huffman*)
- Create symbol-to-codeword lookup tables (*MakeHuffCodeTablenn* on page 6-12)
- Create codeword-to-symbol lookup tables (*MakeHuffDecodeTablenn* on page 6-14)
- Encode fixed-size data to variable-size codewords (*BitCodeByteSymbols, BitCodeHalfWordSymbols, and BitCodeWordSymbols* on page 6-17)
- Decode variable-size codewords to fixed-size data (*BitDecodeByteSymbols, BitDecodeHalfWordSymbols, and BitDecodeWordSymbols* on page 6-19).

6.3.1 Huffman

This function generates an array of the frequency of occurrence for each length of codeword in the data to be coded.

Syntax

```
unsigned int *Huffman(unsigned int freq[ ], unsigned int nfreqs,  unsigned int
                      maxcodewlen)
```

where:

freq is an array of frequency of occurrence for each symbol in the data to be coded, sorted into increasing order of frequency.

If the array is not sorted in order of increasing frequency, the returned array of frequency of occurrence for each length of codeword is incorrect.

nfreqs is the number of frequencies referenced by *freq*.

maxcodewlen is the maximum length of codeword that can be generated.

The value can be from two to the maximum length possible for a codeword (27 bits). The minimum value is two because if the maximum length for a codeword were given as one (or less), there is at most one symbol for coding, and Huffman is beyond the requirements of the data.

The number of distinct symbols in the data to be coded must be less than or equal to $2^{maxcodewlen}$ to guarantee that each symbol can be assigned a codeword. In most cases, the value should be given as 27 because this allows for the best compression ratio to be achieved for the given data.

Return value

An array of frequency of occurrence for each length of codeword.

Notes

The `freq` parameter must be sorted into increasing order of frequency such that:

$freq\,[i] \leq freq\,[i+1]\ \forall\ i$

The `nfreqs` parameter is the number of frequencies that are referenced by `freq`. The maximum index into the array is given as:

$(nfreqs - 1)$

and:

$freq\,[i] \leq freq[nfreqs - 1],\ 0 \leq i \leq (nfreqs - 2)$

This means that the last element in the array, $freq\,[nfreqs - 1]$, is the frequency of occurrence for the symbol that occurs most frequently in the data that is to be coded. The frequency of occurrence for the symbol that occurs most infrequently in the data to be coded is given by `freq`[0].

The number of elements that are created in the array of frequency of occurrence for each length of codeword is given as `maxcodewlen` + 1. The value of `maxcodewlen` defines the maximum codeword length that can be assigned to a symbol, so that all codewords that are created are between one and `maxcodewlen` bits in length. A zero length codeword cannot occur, so the first entry in the returned array is always zero.

The value of `maxcodewlen` ensures that, after creating pure Huffman codewords (which can be greater than the maximum length), codewords are recalculated as necessary to fit within the bit limit on their length. However, the value of `maxcodewlen` should be set as large as possible since any recalculation of codewords reduces the compression ratio that is achieved by using pure Huffman codewords.

6.3.2 MakeHuffCodeTable*nn*

These functions create a symbol-to-codeword lookup table from byte symbols, halfword symbols, or word symbols, where *nn* in the function name is 8 (byte), 16 (halfword), or 32 (word), depending on the symbol size. The lookup table can be used to code a stream of fixed size data that consists of occurrences of the given symbols to a stream of variable size codewords.

Syntax

```
unsigned int *MakeHuffCodeTable8(unsigned char symbols[ ], unsigned int
                                 nsymbols,
                                 unsigned int freqcodelen[ ], unsigned int
                                 maxcodewlen)

unsigned int *MakeHuffCodeTable16(unsigned short symbols[ ], unsigned int
                                  nsymbols,
                                  unsigned int freqcodelen[ ],  unsigned int
                                  maxcodewlen)

unsigned int *MakeHuffCodeTable32(unsigned int symbols[ ], unsigned int
                                  nsymbols,
                                  unsigned int freqcodelen[ ], unsigned int
                                  maxcodewlen)
```

where:

symbols is an array of byte symbols, halfword symbols, or word symbols, sorted into increasing order of frequency of occurrence for each symbol in the data to be coded.

nsymbols is the number of symbols in the array.

freqcodelen

is an array of frequency of occurrence for each length of codeword.

maxcodewlen

is the maximum length of codeword referenced by the array *freqcodelen*. This value also gives the number of elements in the array as *maxcodewlen* + 1. The maximum length must be a value greater than zero, because a zero length codeword can never occur. This value cannot be greater than the maximum length of codeword possible, which is 27.

Return value

A pointer to a symbol-to-codeword lookup table.

Usage

The initial presorted symbol array, `symbols`, must be such that for each index i, i ≤ 0 < `nsymbols`, `symbols`[i]=i. The array of symbols must be sorted into increasing order of frequency of occurrence for each symbol.

For symbols that have identical frequency of occurrence, the ordering is not significant, because although it affects the codeword that is assigned to each symbol, it does not change the length of the assigned codeword. Therefore, in the sorted array of symbols, the frequency of occurrence of `symbols` [i] ≤ the frequency of occurrence of `symbols`[i+1] ∀ i. The symbol that occurs most infrequently in the data to be coded is `symbols`[0].

The maximum symbol value that occurs in the data to be coded must be `nsymbols` – 1. The maximum index into the array is then `nsymbols` – 1, and the frequency of occurrence of `symbols` [i] ≤ the frequency of occurrence of `symbols` [`nsymbols` – 1], 0 ≤ i ≤ (`nsymbols` – 2). This means that the last element in the array, `symbols`[`nsymbols` – 1], is the symbol that occurs most frequently in the data that is to be coded.

The `freqcodelen` parameter must be such that, for each index i, 1 ≤ i ≤ `maxcodewlen`, `freqcodelen` [i] is the frequency of occurrence of codewords of length i. A codeword of length zero can never occur and `freqcodelen` [0] must always be zero. The value of:

$$\sum_{i=1}^{maxcodewlen} freqcodelen[i]$$

must also be the number of symbols in `symbols` that have a non-zero frequency of occurrence.

The symbol-to-codeword lookup table that is returned is unique for the data that is given. However, if `symbols` has symbols that occur with equal frequency, the ordering for these symbols directly affects the codewords that are assigned to these symbols. That is, the symbol-to-codeword lookup table is dependent on the order of the symbols that are given.

6.3.3 MakeHuffDecodeTable*nn*

These functions create codeword-to-symbol lookup tables for byte symbols, halfword symbols, or word symbols, where *nn* in the function name is 8 (byte), 16 (halfword), or 32 (word) depending on the symbol size. The lookup tables can be used to decode a stream of variable-size codewords to a stream of fixed-size data that consists of occurrences of the given symbols.

Syntax

```
unsigned char *MakeHuffCodeTable8(unsigned char symbols[ ], unsigned int
                                  nsymbols,
                                  unsigned int freqcodelen[ ], unsigned int
                                  maxcodewlen, unsigned int tablebits, unsigned
                                  char **lentable)

unsigned short *MakeHuffCodeTable16(unsigned short symbols[ ], unsigned int
                                    nsymbols, unsigned int freqcodelen[ ],
                                    unsigned int maxcodewlen, unsigned int
                                    tablebits, unsigned char **lentable)

unsigned int *MakeHuffCodeTable32(unsigned int symbols[ ], unsigned int
                                  nsymbols,
                                  unsigned int freqcodelen[ ], unsigned int
                                  maxcodewlen, unsigned int tablebits, unsigned
                                  char **lentable)
```

where:

symbols is an array of byte symbols, halfword symbols, or word symbols sorted into increasing order of frequency of occurrence for each symbol in the data that has been coded.

nsymbols is the number of symbols in the array.

freqcodelen is an array of frequency of occurrence for each length of codeword.

maxcodewlen is the maximum length of codeword referenced by the array *freqcodelen*. This value also gives the number of elements in the array as *maxcodewlen* + 1.

tablebits is the maximum length, in bits, of a codeword that can be decoded directly using a direct lookup table.

lentable is a pointer to an unallocated array of unsigned characters to hold the codeword length lookup table.

When this function returns, `lentable` contains a pointer to an array of unsigned characters that defines the codeword length lookup table.

Return value

A pointer to the codeword-to-symbol lookup table.

Usage

The parameters, *`symbols`*, *`nsymbols`*, *`freqcodelen`*, and *`maxcodewlen`* are exactly the same as the parameters to the `MakeHuffCodeTablenn()` functions. See *MakeHuffCodeTablenn* on page 6-12.

The codeword-to-symbol lookup table created by the function is split across two tables:

- *`lentable`*, which contains the length of the codewords that are being decoded
- the codeword-to-symbol lookup table returned by the function, which contains the symbols to which the codewords are decoded.

These tables are synchronized, which means that for a given index there is a value in the symbol table and a corresponding length value in *`lentable`* at the same index.

The tables that are returned are unique, and are only valid for reversing the symbol-to-codeword lookup table returned by a call to `MakeHuffCodeTablenn()`.

Depending on the value in *`tablebits`* and the data to be decoded, the function either uses direct lookup tables, or a combination of direct lookup tables and decoding tree tables.

direct lookup tables

In direct lookup tables, for each codeword in the stream that is to be decoded, the lookup tables have entries that give the symbol that the codeword is decoded to (symbol table) and the length of the codeword that is decoded (*`lentable`*). The index into the tables is the codeword.

decoding tree tables

In decoding tree tables, for each codeword that cannot be decoded directly using a lookup table, the codeword is decoded one bit at a time by traversing a binary tree until a leaf node is reached. The leaf nodes contain the symbols to decode to, and the lengths of codewords that are decoded in the symbol and length trees, respectively.

All codewords of a length up to and including *`tablebits`* have decoding entries defined in the direct lookup tables. All codewords longer than the value in *`tablebits`* have decoding entries defined in the decoding trees. The size of the direct lookup tables is determined, therefore, by the value of *`tablebits`* and the number of entries required for each direct lookup table is defined as $2^{tablebits}$. That is, with each bit increase in the length of codeword that can be decoded using the direct lookup tables, the size of the table doubles.

When specifying the value in *tablebits*, a trade-off between resources and speed is required. Larger values of *tablebits* means that more codewords can be decoded directly (and more quickly), but the direct lookup tables are larger and more memory is required. However, the greater the length of the codeword, the smaller the probability of occurrence.

For example, if the data to be decoded has a Fibonacci sequence for the frequency of occurrence of the original symbols, this gives the highest probability of occurrence for each length of codeword. In this case, a codeword of 10 bits has a probability of occurring of $1/376 \approx 2.7 \times 10^{-3}$ times, and if 10 bits of codeword can be decoded directly, the tables have 1024 entries.

If the length of the codeword that can be decoded directly is increased to 11 bits, a codeword of length 11 has a probability of occurring of $1/609 \approx 1.6 \times 10^{-3}$ times and the number of entries for each table has doubled to 2048. However, a 7-bit codeword, that has a probability of occurring of $1/88 \approx 0.011$, requires 16 entries in the direct decoding tables as opposed to eight entries when the tables only contained 10 bits of direct codeword lookup.

Limitations on input values

The values of *symbols*, *nsymbols*, *freqcodelen*, and *maxcodewlen* must be the same values passed to `MakeHuffCodeTable`*nn*`()`. If the values are not the same, the codeword-to-symbol lookup tables created by this function cannot correctly decode a stream of data that was coded using the symbol-to-codeword lookup table created by the functions `MakeHuffCodeTable`*nn*`()`.

6.3.4 BitCodeByteSymbols, BitCodeHalfWordSymbols, and BitCodeWordSymbols

These functions encode a stream of fixed-size byte input data, halfword input data, or word input data values to a bit-stream of variable-size codewords, using a codeword-length lookup table.

Syntax

```
BitStreamStatePtr BitCodeByteSymbols(unsigned char source[ ], unsigned int n,
BitStreamStatePtr bitstreamstateptr, unsigned int codetable[ ])
```

```
BitStreamStatePtr BitCodeHalfWordSymbols(
unsigned short source[ ],
unsigned int n,
BitStreamStatePtr bitstreamstateptr, unsigned int codetable[ ])
```

```
BitStreamStatePtr BitCodeWordSymbols(unsigned int source[ ],
unsigned int n,
BitStreamStatePtr bitstreamstateptr, unsigned int codetable[ ])
```

where:

source is the array of byte data, halfword data, or word data to be coded.

n is the number of coded values from the input array to be coded. This value can be any number from zero (no coding is performed), to the total number of entries referenced by the array.

bitstreamstateptr

is a pointer to a `BitStreamState` structure that references a bit-stream, and a bit-position within that bit stream.

codetable

is an array of codewords and lengths for encoding the symbols of the input data. The array must contain an entry for every possible symbol that occurs in the input data. The index into the table is the symbol value itself.

Each table entry must be constructed with the bottom *Z* bits containing the codeword that is used to encode the symbol, with the top (32 – *Z*) bits identifying the length of the codeword. The value of *Z* is implementation defined and is specified to the functions by the `CODEWBITS` flag in `bitcodes.s`. It must be given as the value that was used to construct the codeword-length table that is passed in the parameter *`codetable`*.

Return value

A pointer to the `BitStreamState` structure that references the encoded bit-stream and the last bit-position reached within the bit stream.

Usage

When the function is called, the `BitStreamState` structure must reference the first bit-position in an array of unsigned characters to which the codewords can be added. The `BitStreamState` structure that is returned references the first bit-position in the array after the last bit of coded data.

The encoded bit-stream is defined as all the bits from the starting address of the encoded data array to the last bit-position in the last byte reached in the array. The array must reference enough bytes to hold all the coded data. In the worst case, the maximum number of bits of coded data that can be produced is given as the maximum length (in bits) of all codewords, multiplied by *`n`*.

6.3.5 BitDecodeByteSymbols, BitDecodeHalfWordSymbols, and BitDecodeWordSymbols

These functions decode an encoded variable-size codeword bit-stream to a stream of fixed-size byte symbols, halfword symbols, or word symbols, given synchronized length and symbol lookup tables.

Syntax

```
BitStreamStatePtr BitDecodeByteSymbols(
                              BitStreamStatePtr bitstreamstateptr,
                              unsigned int n,
                              unsigned char dest[ ],
                              unsigned char lentable[ ],
                              unsigned char symtable[ ])

BitStreamStatePtr BitDecodeHalfWordSymbols(
                              BitStreamStatePtr bitstreamstateptr,
                              unsigned int n,
                              unsigned short dest[ ],
                              unsigned char lentable[ ],
                              unsigned short symtable[ ])

BitStreamStatePtr BitDecodeWordSymbols(
                              BitStreamStatePtr bitstreamstateptr,
                              unsigned int n,
                              unsigned int dest[ ],
                              unsigned char lentable[ ],
                              unsigned int symtable[ ])
```

where:

bitstreamstateptr
is a pointer to a `BitStreamState` structure that references a bit stream of coded data and bit-position within that bit stream.

n is the number of data items to be decoded.

dest is an array to hold the decoded byte data, halfword data, or word data.

lentable is a table of codeword lengths that can be decoded by the corresponding *symtable*.

symtable is a table of symbols for decoding the codewords from the input bit-stream.

Return value

A pointer to the `BitStreamState` structure that references the encoded bit-stream and the last bit-position reached within the stream.

Usage

When the function is called, the `BitStreamState` structure must reference the first bit-position in an array of unsigned characters that codewords are decoded from. The `BitStreamState` structure that is returned references the first bit-position in the array after the last bit of data that has been decoded.

No explicit value is given for the number of coded bits in the bit-stream being decoded because each codeword in the bit-stream represents a unique symbol. Therefore, by knowing the number of symbols that were coded into the stream, knowledge of the exact number of coded bits in the stream is not required. The number of values to decode into the *`dest`* array is defined by *`n`*, which can be any number from zero (although no decoding is performed) to the maximum of either:

- the total number of entries in *`dest`* to which decoded data can be added
- the number of coded symbols in the bit-stream being decoded.

The two lookup tables, *`lentable`* and *`symtable`*, must be synchronized arrays in which the codewords from the bit-stream are the index into the tables to find the related decoded symbol. The tables can be direct lookup tables or direct lookup tables and decoding trees.

Direct lookup table

The direct lookup table is defined for a maximum length of all codewords, X, in bits, such that for all lengths of codewords from one bit up to and including X bits, the codeword is the index into the lookup table. Therefore, the direct lookup table contains 2^X entries, and the indices into the table are X bits in length.

For codewords whose length, Y, is less than X, the lookup table must contain 2^{X-Y} entries for that codeword. The X bits of the indices for these entries consist of the codeword in the top Y bits, with the bottom $(X - Y)$ bits containing all possible combinations of these bits set. That is, the bottom $(X - Y)$ bits of the codeword take all values from zero to $2^{X-Y} - 1$. The entries in the tables for these codeword indices must be the length of the codeword that is being decoded in *`lentable`*, and the symbol to decode the codeword to in *`symtable`*. In this way each codeword up to X bits in length can be directly looked up in the tables and uniquely decoded because each codeword has the unique prefix property.

Decoding tree table

The decoding tree is defined for all codewords whose length is ($X + 1$) bits or more, up to the maximum possible length of all codewords (27 bits). For each of these codewords, the first X bits of the codeword are all set by the unique prefix property and this value, $2^x - 1$, is used as an index into the direct lookup tables. The entry for this index in `lentable` must be 255. The index into `symtable` gives the root node of the tree and is defined as: (`symtable` + (2 × *the size of symbols in bytes*)).

Each entry in the symbol table is now defined as a tree node that is either:

- an offset from the root of the tree to the next node pair in the tree, such that the next tree index is 2 x offset
- the decoded symbol value.

Each offset to a node pair in the tree gives the left child of the parent at the offset with the right child of the parent given by the left child's index plus the size of symbols in bytes.

Taking the bits from the codeword one bit at a time, starting with $X + 1$, while the codeword has not been decoded, if the bit is set, the right child of the parent node is selected. Otherwise, the left child of the parent node is selected. If the node in the tree is an internal node, such that the value in the symbol tree is for a node pair offset, the value for the node in the length tree must be 255. If the node in the tree is for a decoded symbol value, the value for the node in the length tree must be 255 minus the length of the codeword that is being decoded. In this way, the tree can be traversed, without the length of the codeword being known, and the search terminated when the length entry is no longer given as 255.

The value of X, which determines the size of the direct lookup table and the number of bits of codeword that can be decoded directly, is implementation defined and is specified to the functions by the `TABLEBITS` flag in the file `bitdcods.s`. It must be set to the value that was used to construct the symbol and length tables that are passed in `symtable` and `lentable`.

Chapter 7
Filters

This chapter describes implementations of *finite impulse response* (FIR), *infinite impulse response* (IIR), and *least mean square* (LMS) filters. It contains the following sections:

7.1 Files

The files in Table 7-1 are provided in the implementation.

Table 7-1 Filter files

Filename	Archive name	Code type	Functionality
`firs.s`	`arm_fil\arm_fir`	ARM assembly language	FIR filter
`firs.h`	`arm_fil\arm_fir`	C header	FIR filter function prototype
`iirm.h`	`arm_fil\arm_iir`	ARM assembly language macros	IIR filter macros
`iirs.s`	`arm_fil\arm_iir`	ARM assembly language	IIR filter macro initializations with C-based code wrapping
`iirs.h`	`arm_fil\arm_iir`	C header	IIR filter function prototypes
`lmsm.h`	`arm_fil\arm_lms`	ARM assembly language macros	LMS filter macros
`lmss.s`	`arm_fil\arm_lms`	ARM assembly language	LMS filter macro initializations with C-based code wrapping
`lmss.h`	`arm_fil\arm_lms`	C header	LMS filter function prototypes

7.2 Finite impulse response

The finite impulse response filter is implemented as a non-recursive n-point real filter, with each output characterized by the following difference equation:

$$y_k = \sum_{i=0}^{N-1} x_{k+\ i} c_i$$

where:

x_{k+i} is the *(k + i)*th input sample.

y_k is the *k*th output sample.

c_i is the *i*th coefficient value.

There is one function described in this section:

- Perform a real FIR filter (*s_blk_fir_rhs* on page 7-4).

7.2.1 s_blk_fir_rhs

This function performs FIR filtering on an array of input values, based on a set of coefficients (weights) that determine the behavior of the filter. The result is an array of filtered values.

Syntax

```
void s_blk_fir_rhs(int outputs[], int inputs[], int coeffs[], int nOutputs, int
                    nCoeffs, int nInputs)
```

where:

outputs is an initialized array to hold the output values. The size of the array is defined by *nOutputs*. The number of output values can be any number required and can be greater than the number of inputs, although the extra outputs are returned with zero value.

When the function returns, *outputs* contains an array of output values.

inputs is an array of input sample values. The length of the array is defined by *nInputs*. Any number of input samples can be filtered.

coeffs is an array of coefficient values. The appropriate coefficients must be selected to perform the required filtering. The length of the array is defined by *nCoeffs*. There is no limit on the number of coefficients that can be used. This depends on the required filter specification.

nOutputs is the number of outputs to obtain from the FIR filter, and the size of the output buffer, referenced by *outputs*.

nCoeffs is the number of data points in the coefficient array, referenced by *coeffs*.

nInputs is the number of data points in the input array, referenced by *inputs*.

Limitations on input values

The inputs and the coefficients are 32-bit values, but must complement each other so that multiplying the inputs by the coefficients and summing over the range of the coefficient values does not overflow a 32-bit value. Therefore, for larger input values, the coefficient values must be smaller, and for smaller input values, larger coefficient values are possible.

7.3 Infinite impulse response

The infinite impulse response filter implementation is a cascade realization of second-order canonic filter sections that are characterized by the following equations.

For each input element, the IIR filter of the element over *N* biquads is given as:

$$
\begin{aligned}
v_{0,k} &= x_k \\
w_{i,k} &= v_{i,k} + c_{i,2} w_{i,k-1} + c_{i,3} w_{i,k-2} \\
v_{i,k} &= w_{i,k} + c_{i,0} w_{i,k-1} + c_{i,1} w_{i,k-2} \\
y_k &= v_{N,k}
\end{aligned}
$$

where:

$$
\begin{aligned}
& i \le 0 < N \\
& w_{i,-1} = 0 \\
& w_{i,-2} = 0
\end{aligned}
$$

and:

x_k is the k^{th} input sample.

y_k is the k^{th} output sample.

The macros described in this section are:

- Perform a real IIR filter (*IIR_MACRO* on page 7-6)
- Initialize an array of coefficients and states (*IIR_PowerUp_MACRO* on page 7-8).

7.3.1 IIR_MACRO

This macro performs IIR filtering on an array of input values, based on a set of coefficients (weights) that determine the behavior of the filter. The result of the filtering is an array of data values.

Syntax

```
MACRO IIR_MACRO $outPtr, $inPtr, $nInputs, $nBqs, $coeffsPtr,
                $in, $c0, $c1, $c2, $c3, $s0, $s1, $coefPtrCop
```

where:

$outPtr is a register containing the address of an array to hold the output values. The output array must reference at least as many entries as the input sample array (it must be at least the size defined in *$nInputs*).

On output, *$outPtr* contains a pointer to the array of output values.

$inPtr is a register containing the address of an array of input values. The size of the array is defined in *$nInputs*. The number of input samples that can be filtered is not limited by the macro. The samples must be 16-bit fixed-point values, with the decimal point between bit 13 and bit 14.

$nInputs is a register containing the number of values in the input buffer and the number of values returned in the output array. It is used as a loop counter, and does not contain the original value on output.

$nBqs contains the number of biquads to perform. This number is determined by dividing the number of coefficients by four. The *$nBqs* register is used as a loop counter, and does not contain the original value on output.

$coeffsPtr

is a register containing the address of an array of coefficients interleaved with states, which are required during the IIR filtering. There is no limit on the number of coefficients that can be used, however the number of coefficients must be divisible by four. Each coefficient should be selected to perform the required filtering. More coefficients are required to get the filter specification closer to the ideal filter.

$in, *$c0*, *$c1*, *$c2*, *$c3*, *$s0*, *$s1*, *$coefPtrCop*

are temporary registers required during the calculations. On output, any value is undefined.

Register differentiation

All registers must be distinct.

The register allocation must obey the following ordering:

`$c0` < *`$c1`* < *`$c2`* < *`$c3`* < *`$s0`* < *`$s1`* and *`$in`* < *`$s0`*

See also

For information on setting up *`$coeffsPtr`*, see *IIR_PowerUp_MACRO* on page 7-8.

7.3.2 IIR_PowerUp_MACRO

Given an array of selected coefficients (weights) for IIR filtering, this macro initializes an array of coefficients interleaved with states, which is required by `IIR_MACRO`.

Syntax

```
MACRO IIR_PowerUp_MACRO $outPtr, $inPtr, $taps, $c0, $c1, $c2,
                        $c3, $s0, $s1
```

where:

$outPtr is a register containing the address of an array to hold the coefficients interleaved with the states. The array must be initialized to reference enough memory to hold at least 1.5 × the number of coefficients in the array referenced by *$inPtr*. That is, for every four coefficients in the output coefficient/state array, there are two corresponding state entries that are initialized to zero.

On output, `$outPtr` contains a pointer to the array of coefficients interleaved with the states.

$inPtr is a register containing the address of the array of input coefficient values. The number of entries in the array must be divisible by four. The coefficients must be 16-bit fixed-point values, with the decimal point between bit 13 and bit 14.

$taps contains the number of biquads. This number is determined by dividing the number of coefficient values (size of array referenced by *$inPtr*) by four. The *$taps* register is used as a loop counter and does not contain the original value on output.

$c0, *$c1*, *$c2*, *$c3*, *$s0*, *$s1*

are temporary registers required during the array initialization. On output, any value is undefined.

Register differentiation

All registers must be distinct.

The register allocation must obey the following ordering:

$c0 < *$c1* < *$c2* < *$c3* < *$s0* < *$s1*

7.4 Least mean square

The least mean square filter calculates a FIR filter output, compares this with a predetermined required value, and calculates the error between the values. The adaptation rate, the input value, and the error are multiplied together, and the coefficients for the filter are updated based on this value. The coefficients are initially set to an arbitrary fixed value.

Each output sample, *yk*, *k*=0,1,2,..., is given by:

$$y_k = \sum_{i=0}^{N-1} x_{k-i} c_i$$

where:

x_{k-i} is the $(k\text{-}i)^{\text{th}}$ input sample.

$c_{k,i}$ is the i^{th} coefficient value for sample *k*.

The error estimate, *ek*, for each output sample is given by:

$$e_k = d_k - y_k$$

where:

dk is the k^{th} desired output.

The coefficients are updated after each output sample is calculated by:

$$c_{k+1,i} = c_{k,i} + \mu e_k x_{k-i} \qquad 0 \le i \le N$$

where:

μ is the adaptation rate.

The macros described in this section are:

- Perform an LMS filter (*LMS_MACRO* on page 7-10)
- Initialize an array of coefficients/states (*LMS_PowerUp_MACRO* on page 7-12)
- Retrieve modified coefficients from the coefficient and state array (*LMS_PowerDown_MACRO* on page 7-13).

7.4.1 LMS_MACRO

This macro performs an LMS filtering on an array of 16-bit fixed-point input values, based on a set of coefficients (weights) that determine the behavior of the filter, and a set of desired values to which the filter adapts. The result of the filtering is an array of data values and an array of modified coefficients interleaved with states.

Syntax

```
MACRO LMS_MACRO $outPtr, $inPtr, $nInputs, $desPtr, $out, $c0,
                $in, $tmp, $s0, $c1, $s1, $err, $nCoeffs,
                $coeffsPtr
```

where:

$outPtr is a register containing the address of an array to hold the output values. The output array must reference at least as many entries as the input sample array (it must be at least the size defined in *$nInputs*).

On output, *$outPtr* contains the pointer to the array of output values.

$inPtr is a register containing the address of an array of sample input values. The size of the array is defined by *$nInputs*. The number of samples that can be filtered is not limited by the macro. The samples must be 16-bit fixed-point values, with the decimal point between bit 13 and bit 14.

$nInputs is a register containing the number of values in the input buffer and the number of values returned in the output array. It is used as a loop counter, and does not contain the original value on output.

$desPtr is a register containing the address of the array of required values to which the filter adapts. There must be at least as many entries in this array as in the input array. In other words, for each of the inputs, the ideal FIR filter output is given by the respective desired value. The difference between the required and calculated values is used to update the coefficients of the LMS filter. Each required value should be selected to perform the required filtering and to ensure that the filter is stable.

$s0 is a register containing the adaptation multiplier that determines the step size used by the filter, and gives the rate at which the filter *learns*. A small step size makes the filter slow to adapt, but provides more accurate results with a good convergence rate and greater stability. A large step size makes the adaptation faster, but with less accuracy, a poorer convergence rate, and less stability. Therefore, a trade-off must be accepted between the step size and the rate of adaptation, the error in the values, the rate of convergence, and the stability of the filter.

$out, *$c0*, *$in*, *$tmp*, *$c1*, *$s1*, *$err*

are temporary registers required during the calculations. On output, any value is undefined.

$nCoeffs is a register containing the number of coefficients in the array referenced by *$coeffsPtr*.

$coeffsPtr

is a register containing the address of an array of coefficients interleaved with states, which are required during the LMS filtering. There is no limit on the number of coefficients that can be used, however the number of coefficients must be divisible by two, with each loop of the filter performing two operations. More coefficients are required to get the filter specification closer to the ideal filter.

On output, *coeffsPtr* contains the address of the array of modified coefficients interleaved with states.

Register differentiation

All registers must be distinct.

The register allocation must obey the following ordering:

$c0 < *$in* < *$tmp* < *$s0* < *$c1* < *$s1* and *$s0* < *$nCoeffs*

See also

For information on setting up *$coeffsPtr*, see *LMS_PowerUp_MACRO* on page 7-12.

For information on retrieving the modified coefficients post-LMS filtering, see *LMS_PowerDown_MACRO* on page 7-13.

7.4.2 LMS_PowerUp_MACRO

Given an array of selected coefficients (weights) for LMS filtering, this macro initializes the array of coefficients interleaved with states, which is required by `LMS_MACRO`.

Syntax

```
MACRO LMS_PowerUp_MACRO $outPtr, $inPtr, $nInputs, $c0, $s0,
                        $c1, $s1
```

where:

$outPtr is a register containing the address of an array to hold the coefficients interleaved with the states. The array must be initialized to reference enough memory for at least twice the number of coefficients in the array referenced by *$inPtr*. This means that for every two coefficients in the output coefficient/state array, there are two corresponding state entries that are initialized to zero.

On output, *$outPtr* contains a pointer to the array of coefficients interleaved with the states.

$inPtr is a register containing the address of the array of input coefficient values. The number of entries in the array must be divisible by two. The coefficients must be 16-bit fixed-point values, with the decimal point between bit 13 and bit 14.

$nInputs contains the number of coefficient values, which is the size of the array referenced by *$inPtr*. The *$nInputs* register is used as a loop counter and does not contain the original value on output.

$c0, *$s0*, *$c1*, *$s1*

are temporary registers required during the array initialization. On output, any value is undefined.

Register differentiation

All registers must be distinct.

The register allocation must obey the following ordering:

$c0 < *$s0* < *$c1* < *$s1*

7.4.3 LMS_PowerDown_MACRO

This macro retrieves the modified coefficients and states that have been modified by `LMS_MACRO`, and copies them into the output array.

Syntax

```
MACRO LMS_PowerDown_MACRO $outPtr, $inPtr, $nInputs, $c0, $s0,
                          $c1, $s1
```

where:

`$outPtr` is a register containing the address of an array, of size *`$nInputs`*, to hold the coefficients modified by `LMS_MACRO`. The array must be initialized to reference enough memory to hold at least the number of coefficients in the array referenced by *`$inPtr`* and does not require space for the states which are discarded. The number of coefficients must be divisible by two.

On output, *`$outPtr`* contains a pointer to the array of modified coefficients.

`$inPtr` is a register containing the address of the array of interleaved coefficient and state values. The input values must be `LMS_MACRO` filtered values.

`$nInputs` contains the number of coefficient values in the array referenced by *`$inPtr`* and the size of array referenced by *`$outPtr`*. The *`$nInputs`* register is used as a loop counter and does not contain the original value on output.

`$c0`, *`$s0`*, *`$c1`*, *`$s1`*

are temporary registers required during the array set-up. On output, any value is undefined.

Register differentiation

All registers must be distinct.

The register allocation must obey the following ordering:

`$c0` < *`$s0`* < *`$c1`* < *`$s1`*

Chapter 8
IS-54 Convolutional Encoder

This chapter describes an implementation of the convolutional encoder from the IS-54 standard for digital mobile telephones in the United States of America. It contains the following sections:

8.1 Overview

This section provides general information on the IS-54 convolution encoder.

8.1.1 Implementation

The implementation is based on the Cellular System Dual-Mode Mobile Station-Base Station Compatibility Standard (IS-54-B) from the *Electronic Industries Association* (EIA) and *Telecommunications Industry Association* (TIA).

The input data to the convolutional encoder is made up of 89 bits:

- 77 bits from the speech encoder
- seven bits of *cyclic redundancy check* (CRC) for the frame
- five bits for the tail.

The ARM implementation differs from the IS-54-B standard in that 16 bits of data are processed at a time rather than performing the encoding one bit at a time.

8.1.2 Files

The files in Table 8-1 are provided with the implementation.

Table 8-1 IS-54 files

Filename	Archive name	Code type	Functionality
`is54cem.h`	`arm_is54`	ARM assembly language macro	IS-54 convolutional encoding on a pair of 16 input bits
`is54ces.s`	`arm_is54`	ARM assembly language	Full IS-54 convolutional encoding of 89 bits
`is54ces.h`	`arm_is54`	C header	IS-54 function prototype

8.2 Macro and function

This section describes the IS-54 Convolutional Encoder macro and function. They are:

- Encode two 16-bit values to a 32-bit IS-54 value (*ConvolutionalEncoderKernelMacro*)
- Perform IS-54 encoding of 89 bits (*ConvolutionalEncoder* on page 8-5).

8.2.1 ConvolutionalEncoderKernelMacro

This macro encodes two 16 bits of both speech and CRC into a 32-bit IS-54 value.

Syntax

```
MACRO ConvolutionalEncoderKernelMacro $in, $G0, $G1, $msk1,
                                      $msk2, $msk3, $out
```

where:

$in is a register containing the two 16 bits of both speech and CRC to be encoded, packed into a 32-bit word:

- the high 16 bits must be the current 16 bits
- the low 16 bits must be the previous 16 bits in the sequence.

To encode the last nine bits of both speech and CRC, as in the case at the end of the 89 bits, the high 23 bits should be packed with zeros.

$G0, *$G1* are temporary registers required during the encoding. On output, any value is undefined.

$msk1, *$msk2*, *$msk3*

are registers that must contain the binary masks 0x0F000F00, 0x30303030 and 0x44444444, respectively. If these registers do not contain the required values, the encoding output is undetermined.

$out is a register to hold the 32-bit IS-54 encoding of the two 16 bits of input. When encoding the last nine bits of both speech and CRC, as in the case at the end of the 89 bits, the high 14 bits of the 32-bit output should be ignored because they are all zero and are produced as a result of the packing.

Register differentiation

`$in`, `$G0`, and `$G1` must be distinct registers.

`$G0`, `$G1`, `$msk1`, `$msk2`, and `$msk3` must be distinct registers.

`$in` need not be distinct from `$msk1`, `$msk2`, or `$msk3` (although if `$in` is equal to any of the three mask registers, the IS-54 encoding is applied to the respective mask).

`$out` need not be distinct from `$in`, `$G0`, `$G1`, `$msk1`, `$msk2`, or `$msk3`.

8.2.2 ConvolutionalEncoder

This function encodes 89 bits of both speech and CRC into a sequence of 178 IS-54 encoded output bits.

Syntax

```
void ConvolutionalEncoder(unsigned int outputs[],
                          unsigned short inputs[])
```

where:

outputs is an initialized array, referencing a minimum of six 32-bit entries that is used to hold the 178 output bits of the encoder.

When the function returns, *outputs* holds the 178 IS-54 encoder output bits and 14 pad bits, split over six 32-bit values.

inputs is an array that references seven 16-bit entries that must contain the 89 input bits and 23 bits of padding, split over the 16-bit values. The extra 23 bits of zeros are for padding because the encoding is performed on 16-bit quantities, and the final nine bits of both speech and CRC are padded by seven bits and encoded with a further 16 bits of zeros.

If the top seven bits of the sixth entry and the 16 bits of the seventh entry in the input array are not zero-initialized, the procedure still produces the required encoding. However, the top 14 bits of the final output may contain non-zero data and must not be regarded as encoded data.

Chapter 9
Multi-tone Multi-frequency Generation/Detection

This chapter describes an implementation of a *multi-tone multi-frequency* (MTMF) detector and generator. It contains the following sections:

9.1 Overview

This section provides general information on the MTMF generator and detector.

9.1.1 Implementation

The MTMF generator and detector are implemented using the Goertzel algorithm. The Goertzel algorithm is a second order resonant filter.

Each output value, w_i, $0 \le i < N$, is given by:

$$w_i = 2\cos\theta w_{i-1} - w_{i-2} + x_i$$

where:

$$\theta = \left(\frac{2\pi f}{f_s}\right)$$

f is the frequency of the tone.

f_s is the sample rate frequency.

$w_{-1} = 0$

$w_{-2} = 0$

x_i is the i^{th} input value.

w_N is the basis for the output energy with $w_0, \ldots, w_{N-1}$ being states internal to the algorithm.

The generator implementation is a special case of the Goertzel algorithm. The input is an impulse delta function, incorporated into the value for w_{-1}, such that there is no addition of an input value in calculating each output value, w_i. The intermediate values $w_0, \ldots, w_{N-1}$ are the generator outputs. Therefore, the generator implementation is:

$w_i = 2 \cos\theta w_{\ i-1} - w_{i-2}$

where:

$w_{-1} = \sin\theta$

θ, f, f_s, w_{-2} are defined as before.

9.1.2 Files

The files in Table 9-1 are provided with the implementation.

Table 9-1 MTMF files

Filename	Archive name	Code type	Functionality
`mtmfdets.s`	`arm_mtmf`	ARM assembly language	MTMF detection
`mtmfgens.s`	`arm_mtmf`	ARM assembly language	MTMF generation
`mtmfc.h`	`arm_mtmf`	C header	MTMF generation and detection function prototypes

9.2 ToneState data structure

This structure is used to maintain a tone, and a set of internal state values relative to this tone.

9.2.1 Definition

```
typedef struct   ToneState  ToneState ;
typedef          ToneState  *ToneStatePtr ;

struct ToneState {
    int  tone ;
    int  w0 ;
    int  w1 ;
} ;
```

9.2.2 Description

The MTMF generator and detector operations maintain internal state information. This state information is required between the MTMF generator setup routine and the generator, and between the MTMF detector and the function that obtains the results of the detection. The states and the frequency are maintained in the `ToneState` structure.

9.2.3 Usage

MTMF generation and detection require a `ToneState` structure for each of the tones to be generated or detected. Each structure instance must be initialized before passing through to the MTMF routines. For generation this is done using `ToneGenerateSetup()` and for detection this is done using `ToneDetectSetup()`.

9.3 Functions

This section describes the MTMF routines. The functions are:

- Initialize `ToneState` structures for detection (*ToneDetectSetup*)
- Detect a set of tones in a set of samples (*ToneDetect* on page 9-7)
- Determine energies detected for a set of tones (*ToneDetectResults* on page 9-8)
- Initialize a `ToneState` structure for generation (*ToneGenerateSetup* on page 9-9)
- Generate discrete samples for a waveform comprising of multiple tones (*ToneGenerate* on page 9-11).

9.3.1 ToneDetectSetup

This function initializes an MTMF state structure, for each tone coefficient, for a given set of 16-bit fixed-point tone coefficient values.

Syntax

```
void ToneDetectSetup(ToneState toneStates[ ], int tones[ ], unsigned int nTones)
```

where:

`toneStates`

is the array of MTMF structures initialized by the function.

`tones` is an array of tone coefficients for detection. Although the values are passed in 32-bit integer words, they must be 16-bit fixed-point values with the point between bits 14 and 15.

`nTones` is the number of tones, referenced by *`tones`*, for detection. This value also defines the number of MTMF state structures to be initialized, referenced by *`toneStates`*.

Usage

Each tone coefficient must be given as the value of:

$$\cos\left(\frac{2\pi f}{f_s}\right) \times 2^{15}$$

where:

f is the tone frequency.

f_s is the sample rate frequency.

multiplication by the constant 2^{15} defines the fixed-point precision.

9.3.2 ToneDetect

This function determines the energy for each of the tones in the given waveform, given:

- a stream of 16-bit fixed-point input samples that define a waveform sampled over time
- a set of tones to detect in the waveform, as defined in the MTMF state structures.

Syntax

```
short *ToneDetect(short *inputs, unsigned int nInputs,
                ToneState toneStates[ ], unsigned int nTones)
```

where:

inputs is a pointer to the discrete samples of the waveform in which the tones are to be detected. The input samples must be 16-bit fixed-point values with the point between bits 14 and 15.

nInputs is the number of input samples to analyze.

toneStates

is an array of initialized MTMF structures, with one structure for each tone to be detected.

When the function returns, *toneStates* contains the updated state values that reflect the detected energy for each tone.

nTones is the number of tones to be detected. This value defines the size of the array referenced by *toneStates*.

Return value

A pointer to the next location in the input data after the last input value read for detection. The pointer is given by *inputs* + *nInputs*.

Notes

The detection of each tone is performed by calculating its state values (given in the MTMF state structure for each tone) to reflect the energy of the tone that is detected in the given input waveform. However, the detector cannot fully differentiate between a tone to be detected and frequencies that are near to that tone. To select between frequencies with similar values to the tones, a large number of input samples may be required. The more input samples that are given, the greater the accuracy of the detected energies for the tones, but the slower the detection for the given set of tones.

9.3.3 ToneDetectResults

This function determines the detected energy for each of the tones in an array of `ToneState` structures updated by `ToneDetect()`.

Syntax

```
void ToneDetectResults(unsigned int outputs[ ],
                       unsigned int shift,
                       ToneState toneStates[ ],
                       unsigned int nTones)
```

where:

outputs is an array to hold the detected energy for each of the tones returned by the function.

shift is a scaling factor to ensure that the energies do not overflow 32 bits.

toneStates

is an array of MTMF structures that have been updated by the function `ToneDetect()`.

nTones is the number of tones, and, hence, MTMF state structures referenced by *toneStates*. This value also defines the number of outputs.

Usage

It is possible for any of the detected energies to be large enough that without any scaling, the energy overflows a 32-bit value. Therefore, the value given in *shift* should be half the number of bits to scale the detected energies, so that for each bit given to shift down by, the energy is actually shifted down by two bits.

9.3.4 ToneGenerateSetup

This function initializes an MTMF state structure for the tone coefficient values, given a correction factor for the generator, and the peak value of the waveform to be generated.

Syntax

```
void ToneGenerateSetup(ToneStatePtr toneStatePtr, int tone,
                       int correction, unsigned int level)
```

where:

toneStatePtr

is a pointer to the MTMF structure to be initialized by the function.

When the function returns, *toneStatePtr* contains a pointer to the initialized structure.

tone is the tone coefficient for which the waveform is generated. Although passed in a 32-bit integer word, the value must be 16-bit fixed-point, with the point between bits 14 and 15.

correction

is the MTMF correction factor for the Goertzel algorithm. Although passed in a 32-bit integer word, the value must be 16-bit fixed-point, with the point between bits 14 and 15.

level is the value of the waveform to be generated at its peak.

Usage

The tone coefficient must be given as the value of:

$$\cos\left(\frac{2\pi f}{f_s}\right) \times 2^{15}$$

where:

f is the tone frequency.

f_s is the sample rate frequency.

The correction factor must be given as the value of:

$$\sin\left(\frac{2\pi f}{f_s}\right) \times 2^{15}$$

For both the tone coefficient and the correction factor, the multiplication by the constant 2^{15} defines the fixed-point precision.

9.3.5 ToneGenerate

This function generates a discrete set of samples for the waveform, which represent the given set of tones and initialized states.

Syntax

```
short *ToneGenerate(short outputs[ ], unsigned int nOutputs,
                    ToneState toneStates[ ],
                    unsigned int nTones)
```

where:

outputs is an array to hold the discrete samples for the generated waveform. The samples generated are 16-bit fixed-point values with the point between bits 14 and 15.

nOutputs is the number of samples to be generated for the waveform.

toneStates

is an array of initialized MTMF structures, with one structure for each tone to be generated.

nTones is the number of tones to be generated. The value defines the size of the array referenced by *toneStates*.

Return Value

A pointer to the next location in the output stream after the last output value generated. The pointer is given by *outputs* + *nOutputs*.

Chapter 10
Bit Manipulation

This chapter describes macros to perform common bit manipulation routines. It contains the following sections:

- *Files* on page 10-2
- *Macros* on page 10-3.

10.1 Files

The files in Table 10-1 are provided in the implementation.

Table 10-1 Bit manipulation files

Filename	Archive name	Code type	Functionality
`bitmanm.h`	`arm_bitm`	ARM assembly language macros	Binary coded decimal addition Bit and byte reversal within a word Byte-wise maximum over two words Least and most significant bit set in a word Population count over a set of words
`bstables.s`	`arm_bitm`	ARM assembly language	A `BitSetTable` lookup table for least and most significant bit set in a word

10.2 Macros

This section describes the bit manipulation macros. All macros operate over 32-bit words unless specified otherwise. The macros are:

- Add two binary coded decimal numbers to produce a binary coded decimal result (*BCDADD*)
- Reverse the bits of a word (*BITREV, BITREVC* on page 10-5)
- Reverse the bytes of a word (*BYTEREV, BYTEREVC* on page 10-7)
- Calculate the byte-wise maximum of two words (*BYTEWISEMAX* on page 10-9)
- Find the least or most significant bit set of a word (*LSBSET, MSBSET* on page 10-10)
- Determine the number of bits set in up to seven words (*POPCOUNT, POPCOUNT7* on page 10-12).

10.2.1 BCDADD

This macro adds two binary coded decimal numbers and gives a binary coded decimal result.

Syntax

```
MACRO BCDADD $c, $a, $b, $t, $constant1, $constant2
```

where:

`$c` is a register to hold the binary coded decimal result of the addition of `$a` and `$b`.

`$a, $b` are registers that hold the two binary coded decimal numbers to be added.

`$t` is a temporary register required during the operation. On output, any value is undefined.

`$constant1, $constant2`
are registers that must contain the constant values 0x33333333 and 0x88888888, respectively.

Notes

In a binary coded decimal word each nibble (four bits) of the word can take a value in the range 0–9 only. Therefore, the value given in hexadecimal reads as a decimal value.

Limitations on input values

The output from the macro has no meaning if the two inputs are not binary coded decimal values.

Register differentiation

$constant1 and *$constant2* must be distinct from each other.

$t must be distinct from *$a*, *$b*, *$c*, and *$constant2*.

$c must be distinct from *$a* and *$b*.

$a need not be distinct from *$b*.

$constant1 need not be distinct from *$a*, *$b*, *$c*, or *$t*.

$constant2 need not be distinct from *$a*, *$b*, or *$c*.

10.2.2 BITREV, BITREVC

These macros reverse the bits of a given word (32-bit integer).

Syntax

```
MACRO BITREV $c, $a, $t, $constant

MACRO BITREVC $c, $a, $t, $constant1, $constant2, $constant3,
              $constant4
```

where:

$c is a register to hold the result of the bit-reversal of *$a*. For each bit x of *$a*, the bit is output as bit (31 – x) of *$c*.

$a is a register that holds the word to be bit-reversed. It is assumed that the value is unsigned.

$t is a temporary register required during the operation. On output, any value is undefined.

$constant is a temporary register required for the creation of the constant values during the operation (`BITREV` only).

$constant1, *$constant2*, *$constant3*, *$constant4*

are registers that must contain the constant values 0xffff00ff, 0x0f0f0f0f, 0x33333333, and 0x55555555, respectively (`BITREVC` only).

Cycle counts

Seventeen cycles with no register set-up (`BITREV`) or 12 cycles + five register set-up (`BITREVC`).

Usage

`BITREVC` requires the four constants, *`$constant1`*–*`$constant4`*, to be setup in registers prior to using the macro, but requires only 12 cycles to reverse the bits of the word.

In contrast, `BITREV` does not require the constants to be setup, but does require five more cycles than `BITREVC` to perform the bit-reversal operation.

Determining which macro to use depends on:

- the number of uses to reverse the bits of a word, which determines the total number of cycles after several macro uses
- the number of registers that are available for use.

Register differentiation

`$c` need not be distinct from *`$a`*.

`$a` and *`$c`* must be distinct from *`$t`*.

`$t` and *`$constant`* must be distinct registers (`BITREV` only).

`$a` and *`$c`* must be distinct from *`$constant`* (`BITREV` only).

`$c` and *`$constant2`*, *`$constant3`*, and *`$constant4`* must be distinct registers (`BITREVC` only).

`$t`, *`$constant1`*, *`$constant2`*, *`$constant3`*, and *`$constant4`* must all be distinct registers (`BITREVC` only).

`$c` need not be distinct from *`$constant1`* (`BITREVC` only).

`$a` need not be distinct from any of the *`$constant1`*, *`$constant2`*, *`$constant3`*, and *`$constant4`* (`BITREVC` only).

10.2.3 BYTEREV, BYTEREVC

These macros reverse the bytes of a given word (32-bit integer).

Syntax

`MACRO BYTEREV` *`$c`*`,` *`$a`*`,` *`$t`*

`MACRO BYTEREVC` *`$c`*`,` *`$a`*`,` *`$t`*`,` *`$constant`*

where:

`$c` is a register to hold the result, (d,c,b,a), of the byte-reversal of *`$a`*.

`$a` is a register that holds the word, (a,b,c,d), to be byte-reversed. The value is assumed to be unsigned.

`$t` is a temporary register required during the operation. On output, any value is undefined.

`$constant` is a register that must contain the constant value 0xffff00ff (`BYTEREVC` only).

Cycle counts

Four cycles with no register set-up (`BYTEREV`) or three cycles + register setup (`BYTEREVC`).

Usage

`BYTEREVC` requires the constant 0xffff00ff to be setup in a register prior to using the macro, but requires only three cycles to reverse the bytes of the word.

In contrast, `BYTEREV` does not require the constant setup, but does require one more cycle than `BYTEREVC` to perform the byte-reversal operation.

Determining which macro to use depends on:

- the number of uses to reverse the bytes of a word, which determines the total number of cycles after several macro uses
- the number of registers that are available for use.

Register differentiation

$a and *$t* must be distinct registers.

$c need not be distinct from *$a*.

$c and *$t* must be distinct registers (for BYTEREV only).

$constant and *$t* must be distinct registers (for BYTEREVC only).

$c need not be distinct from *$t* or *$constant* (for BYTEREVC only).

10.2.4 BYTEWISEMAX

This macro creates a word where each byte corresponds to the maximum value from the respective bytes of two given words.

Syntax

```
MACRO BYTEWISEMAX $c, $d, $a, $b, $t, $constant
```

where:

`$c` is a register to hold the byte-wise maximum word.

`$d` is a register to hold a 0 or 1 in each byte of the word, to indicate whether the byte in `$c` is from `$a` (0) or `$b` (1).

`$a, $b` are registers that hold the two words from which the byte-wise maximum word is created. The values are assumed to be unsigned.

`$t` is a temporary register required during the operation. On output, any value is undefined.

`$constant` is a register that must contain the constant value 0x01010101.

Example

Given two words:

`(a,b,c,d)` and `(e,f,g,h)`

this macro calculates:

```
(max(a,e),max(b,f),max(c,g,),max(d,h))
```

Register differentiation

$a, *$c*, *$d* and *$t* must be distinct registers.

$b and *$c* must be distinct registers.

$c and *$d* must be distinct from *$constant*.

$d or *$t* need not be distinct from *$b*.

$t need not be distinct from *$constant*.

$a, *$b*, and *$constant* need not be distinct.

10.2.5 LSBSET, MSBSET

These macros determine the position of the least significant bit set (`LSBSET`) or most significant bit set (`MSBSET`) in a given word.

Syntax

```
MACRO LSBSET $c, $a, $table, $hastable
```

```
MACRO MSBSET $c, $a, $table, $hastable
```

where:

$c is a register to hold the position of the least or most significant bit set in *$a*.

On output, if any bits are set in *$a*, *$c* contains a number between 0 and 31, inclusive, that defines the position of the least or most significant bit set. If no bits are set in the given word (value is 0), a value that is outside of the range 0 to 31, inclusive, is output.

$a is a register that holds the word for which the least significant or most significant bit set is to be determined.

$table is a register that holds the address of `BitSetTable` to determine the position of the least or most significant bit set. This address is either supplied on input or initialized by the macro. If the macro is to be used repeatedly, *$table* can be initialized the first time the macro is used and then passed as a parameter with each subsequent use.

$hastable is an optional parameter that can contain any value. If *$hastable* is present, the address of `BitSetTable` table is supplied to the macro in *$table*. If *$hastable* is not present, *$table* is initialized by the macro.

Usage

The macro requires the address of the `BitSetTable` for the conversion. This address is held in the register identified by *$table*, and can be either:

- supplied to the macro, in which case *$hastable* must be supplied
- initialized by the macro, in which case *$hastable* must not be supplied.

Register differentiation

`$a` and `$c` must be distinct from `$table`.

`$a` and `$c` need not be distinct registers.

10.2.6 POPCOUNT, POPCOUNT7

These macros determine population count, which is the number of bits set (the number of 1s), in the binary expressions of up to seven words.

Syntax

```
MACRO POPCOUNT $c, $a, $t, $constant1, $constant2
```

```
MACRO POPCOUNT7 $res, $a, $b, $c, $d, $e, $f, $g,
                $constant1, $constant2, $constant3
```

where:

$a is a register that holds the word for which the number of bits set is determined (POPCOUNT only).

$a, $b, $c, $d, $e, $f, $g
are registers that hold the seven words for which the number of bits set is determined (POPCOUNT7 only).

$c is a register to hold the result of the population count, which is the number of bits set in the binary expression of *$a* (POPCOUNT only).

$res is a register to hold the result of the population count over the seven words which are the number of bits set in the binary expressions of *$a*, *$b, $c*, *$d*, *$e*, *$f* and *$g* (POPCOUNT7 only).

$t is a temporary register required during the operation (POPCOUNT only). On output, any value is undefined.

$constant1, $constant2
are registers that must contain the constant values 0x49249249 and 0xc71c71c7, respectively (POPCOUNT only).

$constant1, $constant2, $constant3
are registers that must contain the constant values 0x55555555, 0x33333333 and 0x0f0f0f0f, respectively (POPCOUNT7 only).

Cycle counts

Ten cycles + two register constants (POPCOUNT) or 46 cycles + three register constants (POPCOUNT7).

Usage

The choice of which macro to use depends on the number of words on which the population count will be performed and the number of registers available. `POPCOUNT` requires at least four distinct registers, and `POPCOUNT7` requires at least 10 distinct registers.

`POPCOUNT` should be used for a population count of between one and four words. The macro is used repeatedly for each word up to the four given words with the result of each count accumulated into a free register. Each use of `POPCOUNT` takes 10 cycles. Therefore, assuming one cycle for each cumulative count, the total number of cycles for a population count over four words is 44 cycles.

`POPCOUNT7` should be used for a population count of between five and seven words, with a count over five or six words having the remaining words of the count as zero-initialized registers. The number of cycles for `POPCOUNT7` is 46. If two possible cycles for zero-initializing two registers is added, the total number of cycles is less than if `POPCOUNT` was used five or more times.

Register differentiation

$constant1, *$constant2*, *$c*, and *$t* must be distinct registers (`POPCOUNT` only).

$a and *$t* must be distinct registers (`POPCOUNT` only).

$a need not be distinct from *$constant1*, *$constant2*, or *$c* (`POPCOUNT` only).

$a, *$b*, *$c*, *$d*, *$e*, *$f*, *$g*, *$constant1*, *$constant2*, and *$constant3* must all be distinct registers (`POPCOUNT7` only).

$res need not be distinct from the other registers (`POPCOUNT7` only).

Chapter 11
Mathematics

This chapter describes macros that perform common mathematical functions. It contains the following sections:

11.1 Overview

This section provides general information on the mathematics component.

11.1.1 Files

The file in Table 11-1 is provided with the implementation.

Table 11-1 Mathematics files

Filename	Archive name	Code type	Functionality
`mathsm.h`	`arm_math`	ARM assembly language macros	Integer multiplication and division with inputs and outputs of varying precision Fixed-point divisions Fast square root and cube root Fixed-point cosine and sine c=a+abs(b) Signed-saturated addition

11.1.2 ARM architecture requirements

Some of the macro algorithms in this chapter are defined to use long multiply instructions under ARM architectures that support these instructions, with alternative definitions under ARM Architecture Version 3. The alternative definitions require the use of three extra distinct registers specified by `mul_temp_0`, `mul_temp_1` and `mul_temp_2`. The extra registers must be defined before the inclusion of the macro file, and they must be free for use. If ARM Architecture Version 3M, 4 or later is available, the extra registers are not required.

To include the `mathsm.h` file, refer to the following examples:

- ARM architectures that support long multiplies:

```
  INCLUDE mathsm.h
```

- ARM Architecture Version 3 (only):

```
mul_temp_0  RN  2
mul_temp_1  RN  3
mul_temp_2  RN  12
  INCLUDE mathsm.h
```

11.2 Integer multiplication

This section describes the mathematical macros that perform integer multiplication with varying precision. Table 11-2 lists these macros, and shows where they are documented within this section:

Table 11-2

Precision	Unsigned multiplication	Signed multiplication	Documentation
32-bit input, 64-bit output	`UMUL_32x32_64`	`SMUL_32x32_64`	page 11-3 and page 11-4
64-bit input, 64-bit output	`MUL_64x64_64`	`MUL_64x64_64`	page 11-5
64-bit input, 128-bit output	`UMUL_64x64_128`	`SMUL_64x64_128`	page 11-6 and page 11-7

11.2.1 UMUL_32x32_64

This macro multiplies a 32-bit unsigned integer by a 32-bit unsigned integer, producing a 64-bit unsigned integer result in two registers.

Syntax

`MACRO UMUL_32x32_64 $al, $ah, $b, $c`

where:

$al, *$ah* are registers to hold the 64-bit unsigned integer result of the multiplication. The low bits of the result are stored in *$a1*. The high bits of the result are stored in *$ah*.

$b, *$c* are registers that hold the two 32-bit unsigned integers to be multiplied together. If *$c* = *$b*, the result of the multiplication is the square of the value in *$b*.

Register differentiation

$al, *$ah*, and *$b* must be distinct registers.

$al, *$ah*, and *$c* must be distinct registers.

$b and *$c* need not be distinct registers.

11.2.2 SMUL_32x32_64

This macro multiplies a 32-bit signed integer by a 32-bit signed integer, producing a 64-bit signed integer result in two registers.

Syntax

```
MACRO SMUL_32x32_64 $al, $ah, $b, $c
```

where:

`$al, $ah` are registers to hold the 64-bit signed integer result of the multiplication. The low bits of the result are stored in `$a1`. The high bits of the result are stored in `$ah`.

`$b, $c` are registers that hold the 32-bit signed integers to be multiplied together. If `$c` = `$b`, the result of the multiplication is the square of the value in `$b`.

Register differentiation

`$al`, `$ah`, and `$b` must be distinct registers.

`$al`, `$ah`, and `$c` must be distinct registers.

`$b` and `$c` need not be distinct registers.

11.2.3 MUL_64x64_64

This macro multiplies a 64-bit integer by a 64-bit integer, producing a 64-bit integer result in two registers.

Syntax

```
MACRO MUL_64x64_64 $al, $ah, $bl, $bh, $cl, $ch
```

where:

$al, *$ah* are registers to hold the 64-bit integer result of the multiplication ((*$bh*, *$bl*) x (*$ch*,*$cl*)). The result is the low 64 bits of the actual 128-bit result. The low bits of the result are stored in *$a1*. The high bits of the result are stored in *$ah*.

$bl, *$bh* are registers that hold a 64-bit integer. The low bits of the integer are stored in *$b1*. The high bits of the integer are stored in *$bh*.

$cl, *$ch* are registers that hold a 64-bit integer. The low bits of the integer are stored in *$c1*. The high bits of the integer are stored in *$ch*.

Notes

The multiplication is the same, whether the input values are signed or unsigned. If the input values are signed, the result of the multiplication will be signed. Conversely, if the input values are unsigned, the result of the multiplication will be unsigned.

If *$ch* = *$bh* and *$cl* = *$bl*, the result of the multiplication is the square of the 64-bit value in registers *$bh* and *$bl*.

Register differentiation

$al, *$ah*, *$bl*, and *$bh* must be distinct.

$al, *$ah*, *$cl*, and *$ch* must be distinct.

Register pairs (*$bl*, *$bh*) and (*$cl*, *$ch*) need not be distinct.

11.2.4 UMUL_64x64_128

This macro multiplies a 64-bit unsigned integer by a 64-bit unsigned integer, producing a 128-bit unsigned integer result, which is split over four registers, from low to high bits.

Syntax

```
MACRO UMUL_64x64_128 $a0, $a1, $a2, $a3, $bl, $bh, $cl, $ch,
                     $tl, $th
```

where:

$a0, *$a1*, *$a2*, *$a3*

are registers to hold the 128-bit unsigned integer result of the multiplication ((*$bh*, *$bl*) x (*$ch*,*$cl*)). The result is split over the four registers from low 32 bits (*$a0*) to high 32 bits (*$a3*).

$bl, *$bh* are registers that hold a 64-bit unsigned integer. The low bits of the integer are stored in *$b1*. The high bits of the integer are stored in *$bh*.

$cl, *$ch* are registers that hold a 64-bit unsigned integer. The low bits of the integer are stored in *$c1*. The high bits of the integer are stored in *$ch*.

$tl, *$th* are temporary registers required during the multiplication. On output, any value is undefined.

Notes

If *$ch* = *$bh* and *$cl* = *$bl*, the result of the multiplication is the square of the 64-bit value in registers *$bh* and *$bl*.

Register differentiation

$a0, *$a1*, *$a2*, *$a3*, *$bl*, *$bh*, *$tl* and *$th* must be distinct registers.

$a0, *$a1*, *$a2*, *$a3*, *$cl*, *$ch*, *$tl* and *$th* must be distinct registers.

Register pairs (*$bl*, *$bh*) and (*$cl*, *$ch*) need not be distinct.

11.2.5 SMUL_64x64_128

This macro multiplies a 64-bit signed integer by a 64-bit signed integer, producing a 128-bit signed integer result, which is split over four registers, from low to high bits.

Syntax

```
MACRO SMUL_64x64_128 $a0, $a1, $a2, $a3, $bl, $bh, $cl, $ch,
                     $tl, $th
```

where:

$a0, *$a1*, *$a2*, *$a3*

are registers to hold the 128-bit signed integer result of the multiplication ((*$bh*, *$bl*) x (*$ch*,*$cl*)). The result is split over the 4 registers from low 32 bits (*$a0*) to high 32 bits (*$a3*).

$bl, *$bh* are registers that hold the 64-bit signed integer. The low bits of the integer are stored in *$b1*. The high bits of the integer are stored in *$bh*.

$cl, *$ch* are registers that hold the 64-bit signed integer. The low bits of the integer are stored in *$c1*. The high bits of the integer are stored in *$ch*.

$tl, *$th* are temporary registers required during the multiplication. On output, any value is undefined.

Notes

If *$ch* = *$bh* and *$cl* = *$bl*, the result of the multiplication is the square of the 64-bit value in registers *$bh* and *$bl*.

Register differentiation

$a0, *$a1*, *$a2*, *$a3*, *$bl*, *$bh*, *$tl*, and *$th* must be distinct registers.

$a0, *$a1*, *$a2*, *$a3*, *$cl*, *$ch*, *$tl*, and *$th* must be distinct registers.

Register pairs (*$bl*, *$bh*) and (*$cl*, *$ch*) need not be distinct.

11.3 Integer division

This section describes the mathematical macros that perform integer division with varying precision. Table 11-3 lists these macros, and shows where they are documented within this section:

Table 11-3

Precision {n,d}, {q,r}	Unsigned division	Signed division	Documentation
{32,16}-bit input, {16,16}-bit output	`UDIV_32d16_16r16`	SDIV_32d16_16r16	page 11-8 and page 11-10
{32,32}-bit input, {32,32}-bit output	`UDIV_32d32_32r32`	`SDIV_32d32_32r32`	page 11-12 and page 11-14
{64,32}-bit input, {32,32}-bit output	UDIV_64d32_32r32	`SDIV_64d32_32r32`	page 11-16 and page 11-18
{64,64}-bit input, {64-64}-bit output	`UDIV_64d64_64r64`	`SDIV_64d64_64r64`	page 11-20 and page 11-22

11.3.1 UDIV_32d16_16r16

This macro divides a 32-bit unsigned integer numerator by a 16-bit unsigned integer denominator, producing a 16-bit unsigned integer quotient and a 16-bit unsigned integer remainder.

Syntax

```
MACRO UDIV_32d16_16r16 $q, $r, $n, $d
```

where:

$q is a register to hold the 16-bit unsigned integer quotient of the integer.

$r is a register to hold the 16-bit unsigned integer remainder of the integer.

$n is a register that holds the 32-bit unsigned integer numerator.

$d is a register that holds the 16-bit unsigned integer denominator.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$$n = q \times d + r$$

where:

$$0 \le r < d$$

Limitations on input values

It is assumed $n < (d << 16)$. Otherwise, the value in q overflows. This means that division by zero ($d = 0$) is not possible because it violates the constraint.

Register differentiation

$n and *$d* must be distinct registers.

$q and *$r* must be distinct registers.

Register pairs (*$n*, *$d*) and (*$q*, *$r*) need not be distinct.

11.3.2 SDIV_32d16_16r16

This macro divides a 32-bit signed integer numerator by a 16-bit signed integer denominator to produce a 16-bit signed integer quotient and a 16-bit signed integer remainder.

Syntax

```
MACRO SDIV_32d16_16r16 $q, $r, $n, $d, $sign
```

where:

`$q` is a register to hold the 16-bit signed integer quotient of the result.

`$r` is a register to hold the 16-bit signed integer remainder of the result.

`$n` is a register that holds the 32-bit signed integer numerator.

`$d` is a register that holds the 16-bit signed integer denominator.

`$sign` is a temporary register required during the division. On output, any value is undefined.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$n = q \times d + r$

where:

$$0 \le |r| < |d|$$

Limitations on input values

It is assumed $n < (d << 16)$. Otherwise, the value in q overflows. This means that division by zero ($d = 0$) is not possible because it violates the constraint.

Register differentiation

$n, *$d* and *$sign* must be distinct registers.

$q, *$r* and *$sign* must be distinct registers.

Register pairs (*$n*, *$d*) and (*$q*, *$r*) need not be distinct.

11.3.3 UDIV_32d32_32r32

This macro divides a 32-bit unsigned integer numerator by a 32-bit unsigned integer denominator, producing a 32-bit unsigned integer quotient and a 32-bit unsigned integer remainder.

Syntax

```
MACRO UDIV_32d32_32r32 $q, $r, $n, $d
```

where:

$q is a register to hold the 32-bit unsigned integer quotient of the result.

$r is a register to hold the 32-bit unsigned integer remainder of the result.

$n is a register that contains the 32-bit unsigned integer numerator. On output, this register is corrupt.

$d is a register that contains the 32-bit unsigned integer denominator. On output, this register is corrupt.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$$n = q \times d + r$$

where:

$$0 \le r < d$$

If $d = 0$, then $q = 0$ and $r = 0$.

Register differentiation

All registers must be distinct registers.

11.3.4 SDIV_32d32_32r32

This macro divides a 32-bit signed integer numerator by a 32-bit signed integer denominator, producing a 32-bit signed integer quotient and a 32-bit signed integer remainder.

Syntax

```
MACRO SDIV_32d32_32r32 $q, $r, $n, $d,
```

where:

$q is a register to hold the 32-bit signed integer quotient of the result.

$r is a register to hold the 32-bit signed integer remainder of the result.

$n is a register that contains the 32-bit signed integer numerator. On output, this register is corrupt.

$d is a register that contains the 32-bit signed integer denominator. On output, this register is corrupt.

$sign is a temporary register required during the division. On output, any value is undefined.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$n = q \times d + r$

where:

$$0 \le |r| < |d|$$

If $d = 0$, then $q = 0$ and $r = 0$.

In the above calculation, q is rounded towards zero, and r has the same sign as n, therefore:

$\frac{-3}{2} = -1$ remainder -1

$\frac{3}{-2} = -1$ remainder 1

$\frac{-3}{-2} = 1$ remainder -1

Register differentiation

All registers must be distinct registers.

11.3.5 UDIV_64d32_32r32

This macro divides a 64-bit unsigned integer numerator by a 32-bit unsigned integer denominator, producing a 32-bit unsigned integer quotient and a 32-bit unsigned integer remainder.

Syntax

```
MACRO UDIV_64d32_32r32 $q, $r, $nl, $nh, $d
```

where:

$q is a register to hold the 32-bit unsigned integer quotient of the result.

$r is a register to hold the 32-bit unsigned integer remainder of the result.

$nl, $nh are registers that hold the 64-bit unsigned integer numerator. The low bits of the integer are stored in *$nl*. The high bits of the integer are stored in *$nh*.

$d is a register that holds the 32-bit unsigned integer denominator.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$$n = q \times d + r$$

where:

$$0 \le r < d$$

Limitations on input values

The top bit of denominator *d* must be 0 so that $d < 2^{31}$.

It is assumed that $n < (d << 32)$ else the value in *q* overflows. This means that division by zero ($d = 0$) is not possible because it violates the constraint.

Register differentiation

`$nl` and `$nh` must be distinct registers.

`$q` and `$r` must be distinct registers.

`$nl` and `$q` need not be distinct registers.

`$nh` and `$r` need not be distinct registers.

11.3.6 SDIV_64d32_32r32

This macro divides a 64-bit signed integer numerator by a 32-bit signed integer denominator, producing a 32-bit signed integer quotient and a 32-bit signed integer remainder.

Syntax

```
MACRO SDIV_64d32_32r32 $q, $r, $nl, $nh, $d, $sign
```

where:

`$q` is a register to hold the 32-bit signed integer quotient of the result.

`$r` is a register to hold the 32-bit signed integer remainder of the result.

`$nl, $nh` are registers that hold the 64-bit signed integer numerator. The low bits of the integer are stored in *$n1*. The high bits of the integer are stored in *$nh*.

`$d` is a register that holds the 32-bit signed integer denominator.

`$sign` is a temporary register required during the division. On output, any value is undefined.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$$n = q \times d + r$$

where:

$$0 \leq |r| < |d|$$

Limitations on input values

It is assumed that $n < (d << 32)$ else the value in q overflows. This means that division by zero ($d = 0$) is not possible because it violates the constraint.

Register differentiation

$nl, *$nh*, and *$sign* must be distinct registers.

$q, *$r*, and *$sign* must be distinct registers.

$nl and *$q* need not be distinct registers.

$nh and `$r` need not be distinct registers.

11.3.7 UDIV_64d64_64r64

This macro divides a 64-bit unsigned integer numerator by a 64-bit unsigned integer denominator, producing a 64-bit unsigned integer quotient and a 64-bit unsigned integer remainder.

Syntax

```
MACRO UDIV_64d64_64r64 $ql, $qh, $rl, $rh, $nl, $nh, $dl, $dh,
                       $t
```

where:

$ql, *$qh* are registers to hold the 64-bit unsigned integer quotient of the result. The low bits of the result are stored in *$q1*. The high bits of the result are stored in *$qh*.

$rl, *$rh* are registers to hold the 64-bit unsigned integer remainder of the result. The low bits of the result are stored in *$r1*. The high bits of the result are stored in *$rh*.

$nl, *$nh* are registers that hold the 64-bit unsigned integer numerator. The low bits of the integer are stored in *$n1*. The high bits of the integer are stored in *$nh*. On output, these registers are corrupt.

$dl, *$dh* are registers that hold the 64-bit unsigned integer denominator. The low bits of the integer are stored in *$d1*. The high bits of the integer are stored in *$dh*. On output, these registers are corrupt.

$t is a temporary register required during the division. On output, any value is undefined.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$n = q \times d + r$

where:

$$0 \le r < d$$

If $d = 0$, then $q = 0$ and $r = 0$.

Register differentiation

All registers must be distinct registers from each other.

11.3.8 SDIV_64d64_64r64

This macro divides a 64-bit signed integer numerator by a 64-bit signed integer denominator, producing a 64-bit signed integer quotient and a 64-bit signed integer remainder.

Syntax

```
MACRO SDIV_64d64_64r64 $ql, $qh, $rl, $rh, $nl, $nh, $dl, $dh,
                       $t, $sign
```

where:

$ql, $qh are registers to hold the 64-bit signed integer quotient of the result. The low bits of the result are stored in *$q1*. The high bits of the result are stored in *$qh*.

$rl, $rh are registers to hold the 64-bit signed integer remainder of the result. The low bits of the result are stored in *$r1*. The high bits of the result are stored in *$rh*.

$nl, $nh are registers that hold the 64-bit signed integer numerator. The low bits of the integer are stored in *$n1*. The high bits of the integer are stored in *$nh*. On output, these registers are corrupt.

$dl, $dh are registers that hold the 64-bit signed integer denominator. The low bits of the integer are stored in *$d1*. The high bits of the integer are stored in *$dh*. On output, these registers are corrupt.

$t, $sign are temporary registers required during the division. On output, any value is undefined.

Notes

If:

n = numerator
d = denominator
q = quotient
r = remainder

then the macro is equivalent to:

$$\frac{n}{d} = q + \frac{r}{d}$$

or:

$$n = q \times d + r$$

where:

$$0 \le |r| < |d|$$

If $d = 0$, then $q = 0$ and $r = 0$.

In the above calculation, q is rounded towards zero, and r has the same sign as n, therefore:

$$\frac{-3}{2} = -1 \quad \text{remainder -1}$$

$$\frac{3}{-2} = -1 \quad \text{remainder 1}$$

$$\frac{-3}{-2} = 1 \quad \text{remainder -1}$$

Register differentiation

All registers must be distinct registers from each other.

11.4 Fixed-point division

This section describes the mathematical macros that perform fixed-point division. Table 11-4 lists these macros, and shows where they are documented within this section:

Table 11-4

Precision	Unsigned division	Signed division	Documentation
{32,32}-bit fixed-point input, 32-bit fixed-point output	`UDIVF_32d32_32`	`SDIVF_32d32_32`	page 11-24 and page 11-26

11.4.1 UDIVF_32d32_32

This macro divides a 32-bit unsigned fixed-point numerator by a 32-bit unsigned fixed-point denominator, producing a 32-bit unsigned fixed-point result.

Syntax

```
MACRO UDIVF_32d32_32 $q, $n, $d, $bit, $topbit, $overflow
```

where:

$q is a register to hold the 32-bit unsigned fixed-point result.

$n is a register that holds the 32-bit unsigned fixed-point numerator. On output, this register is corrupt.

$d is a register that holds the 32-bit unsigned fixed-point denominator. On output, this register is corrupt.

$bit is a register that holds a constant value that defines the position of the binary point in the two fixed-point input values and the fixed-point output result. The binary point is given as the position before the value in *$bit*.

$topbit ia a register that holds a constant value that defines the maximum number of bits in the result before the value overflows. In other words, the result must be no more than *$topbit* bits. Otherwise, the macro branches to the procedure in *$overflow*. This value is typically 32.

$overflow is an optional label of a procedure to branch to if there is an overflow in the result or if the denominator is zero.

Notes

The format of a number is:

```
             31        $topbit   $bit    0
   unsigned  000...000  xxx...xxx yyy...yyy
```

where x is the integer part and y the fractional part.

Limitations on input values

The two 32-bit fixed-point inputs must have the same number of bits after the binary point.

Register differentiation

$q, *$n*, and *$d* must be distinct registers.

11.4.2 SDIVF_32d32_32

This macro divides a 32-bit signed fixed-point numerator by a 32-bit signed fixed-point denominator, producing a 32-bit signed fixed-point result.

Syntax

```
MACRO SDIVF_32d32_32 $q, $n, $d, $bit, $topbit, $sign, $overflow
```

where:

$q is a register to hold the 32-bit signed fixed-point result.

$n is a register that holds the 32-bit signed fixed-point numerator. On output, this register is corrupt.

$d is a register that holds the 32-bit signed fixed-point denominator. On output, this register is corrupt.

$bit is a register that holds a constant value that defines the position of the binary point in the two fixed-point input values and the fixed-point output result. The binary point is given as the position before the value in *$bit*.

$topbit is a register that holds a constant value that defines the maximum number of bits in the result before the value overflows. In other words, the result must be no more than *$topbit* bits. Otherwise, the macro branches to the procedure in *$overflow*. This value is typically 32.

$sign is a temporary register required during the division. On output, any value is undefined.

$overflow is an optional label to a procedure to branch to if there is an overflow in the result or if the denominator is zero.

Notes

The format of a number is

```
          31       $topbit   $bit    0
signed    sss...sss sxx...xxx yyy...yyy
```

where s is the sign bit, x is the integer part of the number, and y is the fractional part of the number.

Limitations on input values

The two 32-bit fixed-point inputs must have the same number of bits after the binary point.

Register differentiation

`$q`, `$n`, and `$d` must be distinct registers.

11.5 Integer square and cube root

This section describes the macros that perform integer square root and integer cube root. The macros are:

- Square root (*SQR_32_16r17*)
- Cube root (*CBR_32_11* on page 11-30).

11.5.1 SQR_32_16r17

This macro calculates the integer square root of a 32-bit unsigned integer, producing a 16-bit unsigned integer square root and a 17-bit unsigned integer remainder.

Syntax

```
MACRO SQR_32_16r17 $q, $r, $n, $t
```

where:

$q is a register to hold the 16-bit unsigned integer square root result.

$r is a register to hold the 17-bit unsigned integer remainder of the square root result.

$n is a register that holds the 32-bit unsigned integer for which the square root is calculated.

$t is a temporary register required during the calculation. On output, any value is undefined.

Notes

If:

n = input number
q = square root
r = remainder

then the macro is equivalent to:

$n = q \times q + r$

where:

$r \le 2 \times q$ and, therefore, can be 17 bits.

Register differentiation

`$q`, `$r`, and `$t` must be distinct registers.

`$n` need not be distinct from `$q`, `$r`, or `$t`.

11.5.2 CBR_32_11

This macro calculates the integer cube root of a 32-bit signed integer, producing an 11-bit signed integer cube root result.

Syntax

```
MACRO CBR_32_11 $q, $n, $t0, $t1, $t2, $t3, $t4
```

where:

$q is a register to hold the 11-bit signed integer cube root of the value in *$n*.

$n is a register that holds the 32-bit signed integer for which the cube root is calculated.

$t0, $t1, $t2, $t3, $t4

are temporary registers required during the calculation. On output, any value is undefined.

Notes

If:

n = input number
q = cube root

then the macro is equivalent to:

$$q \times q \times q \le n < (q+1) \times (q+1) \times (q+1)$$

The remainder is not returned.

Limitations on input values

There is no limitation on input values.

Register differentiation

$n, *$t0*, *$t1*, *$t2*, *$t3*, and *$t4* must be distinct registers.

$q does not need to be distinct from *$n*, *$t0*, *$t1*, *$t2*, *$t3*, or *$t4*.

11.6 Trigonometric functions

This section describes the macros that perform fixed-point trigonometric functions. The macros are:

- Fixed-point cosine (*ARMCOS*)
- Fixed-point sine (*ARMSIN* on page 11-33).

11.6.1 ARMCOS

This macro calculates the cosine of a 32-bit signed fixed-point radian value to produce a 32-bit signed fixed-point result.

Syntax

```
MACRO ARMCOS $r, $n, $t0, $t1, $prec, $range
```

where:

`$r` is a register to hold the 32-bit signed fixed-point result of the cosine calculation.

`$n` is a register that holds the 32-bit signed fixed-point radian value for which the cosine is calculated. The value cannot be specified in degrees. On output, this register is corrupt.

`$t0`, *`$t1`* are temporary registers required during the calculation. On output, any value is undefined.

`$prec` is a register that holds a value defining the position of the binary point in the fixed-point input value and in the fixed-point output result. The binary point is given as the position before bit *`$prec`* and determines the precision of the calculation. The value in *`$prec`* must be an integer such that 0 ≤ *`$prec`* ≤ 14.

`$range` is a register that holds 0 if the fixed-point radian value in register *`$n`* is in the range $-\pi/2 <$ *`$n`* $< \pi/2$, shifted to the fixed-point precision given in *`$prec`*. Otherwise, *`$range`* is set to 1.

Limitations on input values

If *`$range`* is 0 and the radian value of *`$n`* is outside the range $-\pi/2 <$ *`$n`* $< \pi/2$, shifted to the same fixed-point precision as *`$n`*, the result is undetermined.

Register differentiation

All registers must be distinct.

11.6.2 ARMSIN

This macro calculates the sine of a 32-bit signed fixed-point radian value to produce a 32-bit signed fixed-point result.

Syntax

```
MACRO ARMSIN $r, $n, $t0, $t1, $prec, $range
```

where:

$r is a register to hold the 32-bit signed fixed-point result of the sine calculation.

$n is a register that holds the 32-bit signed fixed-point radian value for which the sine is calculated. The value cannot be specified in degrees. On output, this register is corrupt.

$t0, *$t1* are temporary registers required during the calculation. On output, any value is undefined.

$prec is a register that holds a value defining the position of the binary point in the fixed-point input value and in the fixed-point output result. The binary point is given as the position before bit *$prec* and determines the precision of the calculation. The value in *$prec* must be an integer such that 0 ≤ *$prec* ≤ 14.

$range is a register set to 0 if the fixed-point radian value in register *$n* is in the range –π/2 < *$n* < π/2, shifted to the fixed-point precision given in *$prec*. Otherwise, *$range* is set to 1.

Notes

If *$range* is 0 and the radian value of *$n* is outside the range –π/2 < *$n* < π/2, shifted to the same fixed-point precision as *$n*, the result is undetermined.

Register differentiation

All registers must be distinct.

11.7 General macros

This section describes the macros that perform general mathematical operations. The macros are:

- Addition of absolute value, c=a+abs(b) (*ADDABS*)
- Signed-saturated addition (*SIGNSAT* on page 11-35).

11.7.1 ADDABS

This macro adds the absolute value of a 32-bit signed integer to a 32-bit signed integer, producing a 32-bit signed integer result.

Syntax

```
MACRO ADDABS $c, $a, $b
```

where:

$c is a register to hold the 32-bit signed integer result of the addition (*$a* + abs(*$b*)).

$a is a register that holds the 32-bit signed integer to add to the absolute value of *$b*.

$b is a register that holds the 32-bit signed integer to add the absolute value of to *$a*.

Register differentiation

$a and *$c* must be distinct registers.

$b and *$c* must be distinct registers.

$a and *$b* need not be distinct registers.

11.7.2 SIGNSAT

This macro adds a 32-bit signed integer to a 32-bit signed integer, producing a saturated 32-bit signed integer result.

Syntax

```
MACRO SIGNSAT $c, $a, $b, $constant
```

where:

$c is a register to hold the saturated 32-bit signed integer result of the addition.

$a, *$b* are registers that hold the two 32-bit signed integers to be added together.

$constant is a register that must contain the constant value 0x8000000.

Notes

The saturated result is such that:

- if *$a* and *$b* are both positive and *$a* + *$b* is negative the result is 0x7fffffff
- if *$a* and *$b* are both negative and *$a* + *$b* is positive the result is 0x80000000
- otherwise, the result is the value of *$a* + *$b*.

Register differentiation

$c and *$constant* must be distinct registers.

$a, *$b*, and *$constant* need not be distinct registers.

$a, *$b*, and *$c* need not be distinct registers.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.