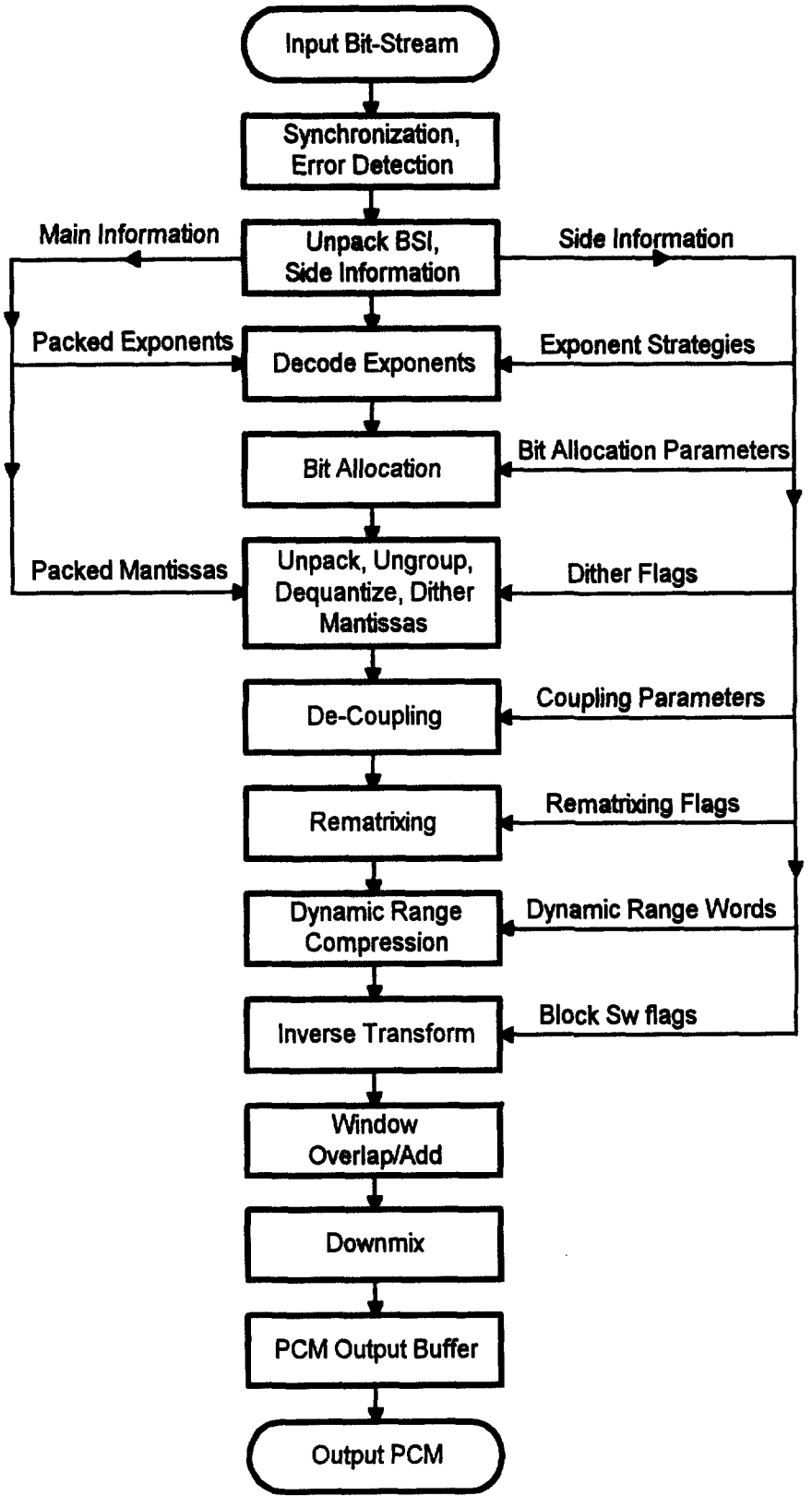

Figure 6.1 Flow diagram of the decoding process.

6.2.2 Synchronization and Error Detection

The AC-3 bit-steam format allows rapid synchronization. The 16-bit sync word has a low probability of false detection. With no input stream alignment the probability of false detection of the sync word is 0.0015% per input stream bit position. For a bit-rate of 384 kb/s, the probability of false sync word detection is 19% per frame. Byte alignment of the input stream drops this probability to 2.5%, and word alignment drops it to 1.2%.

When a sync pattern is detected the decoder may be estimated to be in sync and one of the CRC words (crc1 or crc2) may be checked. Since crc1 comes first and covers the first 5/8 of the frame, the result of a crc1 check may be available after only 5/8 of the frame has been received. Or, the entire frame size can be received and crc2 checked. If either CRC checks, the decoder may safely be presumed to be in sync and decoding and reproduction of audio may proceed. The chance of false sync in this case would be the concatenation of the probabilities of a false sync word detection and a CRC misdetection of error. The CRC check is reliable to 0.0015%. This probability, concatenated with the probability of a false sync detection in a byte aligned input bit stream, yield a probability of false synchronization of 0.000035% (or about once in 3 million synchronization attempts).

If this small probability of false sync is too large for an application, there are several methods which may reduce it. The decoder may only presume correct sync in the case that both CRC words check properly. The decoder may require multiple sync words to be received with the proper alignment. If the data transmission or storage system is aware that data is in error, this information may be made known to the decoder.

Additional details on methods of bit stream synchronization are not provided in this standard. Details on the CRC calculation are provided in Section 7.10 on page 91.

6.2.3 Unpack BSI, Side Information

Inherent to the decoding process is the unpacking (de-multiplexing) of the various types of information included in the bit stream. Some of these items may be copied from the input buffer to dedicated registers, some may be copied to specific working memory location, and some of the items may simply be located in the input buffer with pointers to them saved to another location for use when the information is required. The information which must be unpacked is specified in detail in Section 5.3. Further details on the unpacking of BSI and side information are not provided in this standard.

6.2.4 Decode Exponents

The exponents are delivered in the bit stream in an encoded form. In order to unpack and decode the exponents two types of side information are required. First, the number of exponents must be known. For fbw channels this may be determined from either chbwcod[ch] (for uncoupled channels) or from cplbegf (for coupled channels). For the coupling channel, the number of exponents may be determined from cplbegf and cplendf. For the lfe channel (when on), there are always 7 exponents. Second, the exponent strategy in use (D15, etc.) by each channel must be known. The details on how to unpack and decode exponents are provided in Section 7.1 on page 44.

6.2.5 Bit Allocation

The bit allocation computation reveals how many bits are used for each mantissa. The inputs to the bit allocation computation are the decoded exponents, and the bit allocation side information. The outputs of the bit allocation computation are a set of bit allocation pointers (baps), one bap for each coded mantissa. The bap indicates the quantizer used for the mantissa, and how many bits in the bit stream were used for each mantissa. The bit allocation computation is described in detail in Section 7.2 on page 49.

6.2.6 Process Mantissas

The coarsely quantized mantissas make up the bulk of the AC-3 data stream. Each mantissa is quantized to a level of precision indicated by the corresponding bap. In order to pack the mantissa data more efficiently, some mantissas are grouped together into a single transmitted value. For instance, two 11-level quantized values are conveyed in a single 7-bit code (3.5 bits/value) in the bit stream.

The mantissa data is unpacked by peeling off groups of bits as indicated by the baps. Grouped mantissas must be ungrouped. The individual coded mantissa values are converted into a de-quantized value. Mantissas which are indicated as having zero bits may be reproduced as either zero, or by a random dither value (under control of the dither flag). The mantissa processing is described in full detail in Section 7.3 on page 64.

6.2.7 De-Coupling

When coupling is in use, the channels which are coupled must be decoupled. Decoupling involves reconstructing the high frequency Section (exponents and mantissas) of each coupled channel, from the common coupling channel and the coupling coordinates for the individual channel. Within each coupling band, the coupling channel coefficients (exponent and mantissa) are multiplied by the individual channel coupling coordinates. The coupling process is described in detail in Section 7.4 on page 68.

6.2.8 Rematrixing

In the 2/0 audio coding mode rematrixing may be employed, as indicated by the rematrix flags (rematflg[rbnd]). Where the flag indicates a band is rematrixed, the coefficients encoded in the bit stream are sum and difference values instead of left and right values. Rematrixing is described in detail in Section 7.5 on page 70.

6.2.9 Dynamic Range Compression

For each block of audio a dynamic range control value (dynrng) may be included in the bit stream. The decoder, by default, shall use this value to alter the magnitude of the coefficient (exponent and mantissa) as specified in Section 7.7.1 on page 75.

6.2.10 Inverse Transform

The decoding steps described above will result in a set of frequency coefficients for each encoded channel. The inverse transform converts the blocks of frequency coefficients into blocks of time samples. The inverse transform is detailed in Section 7.9 on page 85.

6.2.11 Window, Overlap/Add

The individual blocks of time samples must be windowed, and adjacent blocks must be overlapped and added together in order to reconstruct the final continuous time output PCM audio signal. The window and overlap/add steps are described along with the inverse transform in Section 7.9 on page 85.

6.2.12 Downmixing

If the number of channels required at the decoder output is smaller than the number of channels which are encoded in the bit stream, then downmixing is required. Downmixing in the time domain is shown in this example decoder. Since the inverse transform is a linear operation, it is also possible to downmix in the frequency domain prior to transformation. Section 7.8 on page 80 describes downmixing and specifies the downmix coefficients which decoders shall employ.

6.2.13 PCM Output Buffer

Typical decoders will provide PCM output samples at the PCM sampling rate. Since blocks of samples result from the decoding process, an output buffer is typically required. This standard does not specify or describe output buffering in any further detail.

6.2.14 Output PCM

The output PCM samples may be delivered in form suitable for interconnection to a digital to analog converter (DAC), or in any other form. This standard does not specify the output PCM format.

7. ALGORITHMIC DETAILS

The following sections describe various aspects of AC-3 coding in detail.

7.1 Exponent Coding

7.1.1 Overview

The actual audio information conveyed by the AC-3 bit stream consists of the quantized frequency coefficients. The coefficients are delivered in floating point form, with each coefficient consisting of an exponent and a mantissa. This section describes how the exponents are encoded and packed into the bit stream.

Exponents are 5-bit values which indicate the number of leading zeros in the binary representation of a frequency coefficient. The exponent acts as a scale factor for each mantissa, equal to 2^{-exp}. Exponent values are allowed to range from 0 (for the largest value coefficients with no leading zeroes) to 24. Exponents for coefficients which have more than 24 leading zeroes are fixed at 24, and the corresponding mantissas are allowed to have leading zeros. Exponents require 5 bits in order to represent all allowed values.

AC-3 bit streams contain coded exponents for all independent channels, all coupled channels, and for the coupling and low frequency effects channels (when they are enabled). Since audio information is not shared across frames, block 0 of every frame will include new exponents for every channel. Exponent information may be shared across blocks within a frame, so blocks 1 through 5 may reuse exponents from previous blocks.

AC-3 exponent transmission employs differential coding, in which the exponents for a channel are differentially coded across frequency. The first exponent of a fbw or lfe channel is always sent as a 4-bit absolute value, ranging from 0-15. The value indicates the number of leading zeros of the first (DC term) transform coefficient. Successive (going higher in frequency) exponents are sent as differential values which must be added to the prior exponent value in order to form the next absolute value.

The differential exponents are combined into groups in the audio block. The grouping is done by one of three methods, D15, D25, or D45, which are referred to as exponent strategies. The number of grouped differential exponents placed in the audio block for a particular channel depends on the exponent strategy and on the frequency bandwidth information for that channel. The number of exponents in each group depends only on the exponent strategy.

An AC-3 audio block contains two types of fields with exponent information. The first type defines the exponent coding strategy for each channel, and the second type contains the actual coded exponents for channels requiring new exponents. For independent channels, frequency bandwidth information is included along with the exponent strategy fields. For coupled channels, and the coupling channel, the frequency information is found in the coupling strategy fields.

7.1.2 Exponent Strategy

Exponent strategy information for every channel is included in every AC-3 audio block. Information is never shared across frames, so block 0 will always contain a strategy indication (D15, D25, or D45) for each channel. Blocks 1 through 5 may indicate reuse of the prior (within the same frame) exponents. The three exponent coding strategies provide a tradeoff between data rate required for exponents, and their frequency resolution. The D15 mode provides the finest frequency resolution, and the D45 mode requires the least amount of data. In all three modes, a number differential exponents are combined into 7-bit words when coded into an audio block. The main difference between the modes is how many differential exponents are combined together.

The absolute exponents found in the bit stream at the beginning of the differentially coded exponent sets are sent as 4-bit values which have been limited in either range or resolution in order to save one bit. For fbw and lfe channels, the initial 4-bit absolute exponent represents a value from 0 to 15. Exponent values larger than 15 are limited to a value of 15. For the coupled channel, the 5-bit absolute exponent is limited to even values, and the lsb is not transmitted. The resolution has been limited to valid values of 0,2,4...24. Each differential exponent can take on one of five values: -2, -1, 0, +1, +2. This allows deltas of up to ±2 (± 12 dB) between exponents. These five values are mapped into the values 0, 1, 2, 3, 4 before being grouped, as shown in Table 7.1.

Table 7.1 Mapping of differential exponent values, D15 mode.

diff exp	mapped value
+ 2	4
+ 1	3
0	2
- 1	1
- 2	0

Mapped Value = Diff Exp + 2 ;
Diff Exp = Mapped Value - 2 ;

In the D15 mode, the above mapping is applied to each individual differential exponent for coding into the bit stream. In the D25 mode, each *pair* of differential exponents is represented by a single mapped value in the bit stream. In this mode the second differential exponent of each pair is implied as a delta of 0 from the first element of the pair as indicated in Table 7.2.

Table 7.2 Mapping of differential exponent values, D25 mode.

diff exp n	diff exp n+1	mapped value
+2	0	4
+1	0	3
0	0	2
-1	0	1
-2	0	0

The D45 mode is similar to the D25 mode except that *quads* of differential exponents are represented by a single mapped value, as indicated by Table 7.3.

Table 7.3 Mapping of differential exponent values, D45 mode.

diff exp n	diff exp n+1	diff exp n+2	diff exp n+3	mapped value
+2	0	0	0	4
+1	0	0	0	3
0	0	0	0	2
-1	0	0	0	1
-2	0	0	0	0

Since a single exponent is effectively shared by 2 or 4 different mantissas, encoders must ensure that the exponent chosen for the pair or quad is the minimum absolute value (corresponding to the largest exponent) needed to represent all the mantissas.

For all modes, sets of three adjacent (in frequency) mapped values (M1,M2 and M3) are grouped together and coded as a 7 bit value according to the following formula:

Coded 7 bit Grouped Value = (25 × M1) + (5 × M2) + M3 ;

The exponent field for a given channel in an AC-3 audio block consists of a single absolute exponent followed by a number of these grouped values.

7.1.3 Exponent Decoding

The exponent strategy for each coupled and independent channel is included in a set of 2-bit fields designated chexpstr[ch]. When the coupling channel is present, a cplexpstr strategy code is also included. Table 7.4 shows the mapping from exponent strategy code into exponent strategy.

Table 7.4 Exponent strategy coding.

chexpstr[ch], cplexpstr	exponent strategy	exponents per group
00	reuse prior exponents	0
01	D15	3
10	D25	6
11	D45	12

When the low frequency effects channel is enabled the lfeexpstr field is present. It is decoded as shown in Table 7.5.

Table 7.5 LFE channel exponent strategy coding.

lfeexpstr	exponent strategy	exponents per group
0	reuse prior exponents	0
1	D15	3

Following the exponent strategy fields in the bit stream is a set of channel bandwidth codes, chbwcod[ch]. These are only present for independent channels (channels

not in coupling) that have new exponents in the current block. The channel bandwidth code defines the end mantissa bin number for that channel according to the following:

```
endmant[ch] = ((chbwcod[ch] + 12) * 3) + 37 ;   /* (ch is not coupled) */
```

For coupled channels the end mantissa bin number is defined by the starting bin number of the coupling channel:

```
endmant[ch] = cplstrtmant ;                     /* (ch is coupled) */
```

where cplstrtmant is as derived below. By definition the starting mantissa bin number for independent and coupled channels is 0.

```
strtmant[ch] = 0 ;
```

For the coupling channel, the frequency bandwidth information is derived from the fields cplbegf and cplendf found in the coupling strategy information. The coupling channel starting and ending mantissa bins are defined as:

```
cplstrtmant = (cplbegf * 12) + 37 ;
cplendmant = ((cplendf + 3) * 12) + 37 ;
```

The low frequency effects channel, when present, always starts in bin 0 and always has the same number of mantissas:

```
lfestrtmant = 0 ;
lfeendmant = 7 ;
```

The second set of fields contains coded exponents for all channels indicated to have new exponents in the current block. These fields are designated as exps[ch][grp] for independent and coupled channels, cplexps[grp] for the coupling channel, and lfeexps[grp] for the low frequency effects channel. The first element of the exps fields (exps[ch][0]) and the lfeexps field (lfeexps[0]) is always a 4-bit absolute number. For these channels the absolute exponent always contains the exponent value of the first transform coefficient (bin #0). These 4 bit values correspond to a 5-bit exponent which has been limited in range (0 to 15, instead of 0 to 24), i.e., the most significant bit is zero. The absolute exponent for the coupled channel, cplabsexp, is only used as a reference to begin decoding the differential exponents for the coupling channel (i.e. it does not represent an actual exponent). The cplabsexp is contained in the audio block as a 4-bit value, however it corresponds to a 5-bit value. The LSB of the coupled channel initial exponent is always 0, so the decoder must take the 4-bit value which was sent, and double it (left shift by 1) in order to obtain the 5-bit starting value.

For each coded exponent set the number of grouped exponents (not including the first absolute exponent) to decode from the bit stream is derived as follows:

For independent and coupled channels:

```
nchgrps[grp]    = truncate ((endmant[grp] - 1) / 3) ;       /* for D15 mode */
                = truncate ((endmant[grp] - 1 + 3) / 6) ;   /* for D25 mode */
                = truncate ((endmant[grp] - 1 + 9) / 12) ;  /* for D45 mode */
```

For the coupling channel:

```
ncplgrps      = (cplendmant - cplstrtmant) / 3     /* for D15 mode */
              = (cplendmant - cplstrtmant) / 6     /* for D25 mode */
              = (cplendmant - cplstrtmant) / 12    /* for D45 mode */
```

For the low frequency effects channel:

```
nlfegrps      = 2 ;
```

Decoding a set of coded grouped exponents will create a set of 5-bit absolute exponents. The exponents are decoded as follows:

1. Each 7 bit grouping of mapped values (gexp) is decoded using the inverse of the encoding procedure:

```
M1 = truncate (gexp / 25) ;
M2 = truncate ((gexp % 25) / 5) ;
M3 = (gexp % 25) % 5 ;
```

4. Each mapped value is converted to a differential exponent (dexp) by subtracting the mapping offset:

```
dexp = M - 2 ;
```

3. The set of differential exponents if converted to absolute exponents by adding each differential exponent to the absolute exponent of the previous frequency bin:

```
exp[n] = exp[n-1] + dexp[n] ;
```

4. For the D25 and D45 modes each absolute exponent is copied to the remaining members of the pair or quad.

The above procedure can be summarized as follows:

Pseudo code

```
/* unpack the mapped values */
for (grp = 0; grp < ngrps; grp++)
{
    expacc = gexp[grp] ;
    dexp[grp * 3] = truncate (expacc / 25) ;
    expacc = expacc - ( 25 * dexp[grp * 3]) ;
    dexp[(grp * 3) + 1] = truncate ( expacc / 5) ;
    expacc = expacc - (5 * dexp[(grp * 3) + 1]) ;
    dexp[(grp * 3) + 2] = expacc ;
}

/* unbiased mapped values */
for (grp = 0; grp < (ngrps * 3); grp++)
{
    dexp[grp] = dexp[grp] - 2 ;
}

/* convert from differentials to absolutes */
prevexp = absexp ;
for (i = 0; i < (ngrps * 3); i++)
{
    aexp[i] = prevexp + dexp[i] ;
    prevexp = aexp[i] ;
```

Pseudo code

```
}

/* expand to full absolute exponent array, using grpsize */
exp[0] = absexp ;
for (i = 0; i < (ngrps * 3); i++)
{
    for (j = 0; j < grpsize; j++)
    {
        exp[(i * grpsize) + j +1] = aexp[i] ;
    }
}
```

where,

ngrps = number of grouped exponents (nchgrps[ch], ncplgrps, or nlfegrps)
grpsize = 1 for D15
= 2 for D25
= 4 for D45
absexp = absolute exponent (exps[ch][0], (cplabsexp<<1), or lfeexps[0])

For the coupling channel the above output array, exp[n], should be offset to correspond to the coupling start mantissa bin:

cplexp[n + cplstrtmant] = exp[n + 1] ;

For the remaining channels exp[n] will correspond directly to the absolute exponent array for that channel.

7.2 Bit Allocation

7.2.1 Overview

The bit allocation routine analyzes the spectral envelope of the audio signal being coded with respect to masking effects to determine the number of bits to assign to each transform coefficient mantissa. In the encoder, the bit allocation is performed globally on the ensemble of channels as an entity, from a common bit pool. There are no preassigned exponent or mantissa bits, allowing the routine to flexibly allocate bits across channels, frequencies, and audio blocks in accordance with signal demand.

The bit allocation contains a parametric model of human hearing for estimating a noise level threshold, expressed as a function of frequency, which separates audible from inaudible spectral components. Various parameters of the hearing model can be adjusted by the encoder depending upon signal characteristics. For example, a prototype masking curve is defined in terms of two piecewise continuous line segments, each with its own slope and y-axis intercept. One of several possible slopes and intercepts is selected by the encoder for each line segment. The encoder may iterate on one or more such parameters until an optimal result is obtained. When all parameters used to estimate the noise level threshold have been selected by the encoder, the final bit allocation is computed. The model parameters are conveyed to the decoder with other side information. The decoder executes the routine in a single pass.

The estimated noise level threshold is computed over 50 bands of nonuniform bandwidth (an approximate 1/6 octave scale). The banding structure, defined by tables in the next section, is independent of sampling frequency. The required bit allocation for each mantissa is established by performing a table lookup based upon the difference between the input signal power spectral density (PSD) evaluated on a fine-grain uniform frequency scale, and the estimated noise level threshold evaluated on the coarse-grain (banded) frequency scale. Therefore, the bit allocation result for a particular channel has spectral granularity corresponding to the exponent strategy employed. More specifically, a separate bit allocation will be computed for each mantissa within a D15 exponent set, each pair of mantissas within a D25 exponent set, and each quadruple of mantissas within a D45 exponent set.

The bit allocation must be computed in the decoder whenever the exponent strategy (chexpstr, cplexpstr, lfeexpstr) for one or more channels does not indicate reuse, or whenever baie, snroffste, or deltbaie = 1. Accordingly, the bit allocation can be updated at a rate ranging from once per audio block to once per 6 audio blocks, including the integral steps in between. A complete set of new bit allocation information is always transmitted in audio block 0.

Since the parametric bit allocation routine must generate identical results in all encoder and decoder implementations, each step is defined exactly in terms of fixed-point integer operations and table lookups. Throughout the discussion below, signed two's complement arithmetic is employed. All additions are performed with an accumulator of 14 or more bits. All intermediate results and stored values are 8-bit values.

7.2.2 Parametric Bit Allocation

This section describes the seven-step procedure for computing the output of the parametric bit allocation routine in the decoder. The approach outlined here starts with a single uncoupled or coupled exponent set and processes all the input data for each step prior to continuing to the next one. This technique, called vertical execution, is conceptually straightforward to describe and implement. Alternatively, the seven steps can be executed horizontally, in which case multiple passes through all seven steps are made for separate subsets of the input exponent set.

The choice of vertical vs. horizontal execution depends upon the relative importance of execution time vs. memory usage in the final implementation. Vertical execution of the algorithm is usually faster due to reduced looping and context save overhead. However, horizontal execution requires less RAM to store the temporary arrays generated in each step. Hybrid horizontal/vertical implementation approaches are also possible which combine the benefits of both techniques.

7.2.2.1 Initialization

Compute start/end frequencies for the channel being decoded. These are computed from parameters in the bit stream as follows:

Pseudo code

```
/* for fbw channels */
for(ch=0; ch<nfchans; ch++)
{
    strtmant[ch] = 0;
    if(chincpl[ch]) endmant[ch] = 37 + (12 × cplbegf) ; /* channel is coupled */
    else endmant[ch] = 37 + (3 × (chbwcod + 12)) ; /* channel is not coupled */
}

/* for coupling channel */
cplstrtmant = 37 + (12 × cplbegf) ;
cplendmant = 37 + (12 × (cplendf + 3)) ;

/* for lfe channel */
lfestartmant = 0 ;
lfeendmant = 7 ;
```

Special case processing step:

Before continuing with the initialization procedure, all SNR offset parameters from the bit stream should be evaluated. These include csnroffst, fsnroffst[ch], cplfsnroffst, and lfefsnroffst. If they are all found to be equal to zero, then all elements of the bit allocation pointer array bap[] should be set to zero, and no other bit allocation processing is required for the current audio block.

Perform table lookups to determine the values of sdecay, fdecay, sgain, dbknee, and floor from parameters in the bit stream as follows:

Pseudo code

```
sdecay = slowdec[sdcycod] ;      /* Table 7.6 */
fdecay = fastdec[fdcycod] ;      /* Table 7.7 */
sgain = slowgain[sgaincod] ;     /* Table 7.8 */
dbknee = dbpbtab[dbpbcod] ;      /* Table 7.9 */
floor = floortab[floorcod] ;     /* Table 7.10 */
```

Initialize as follows for uncoupled portion of fbw channel:

Pseudo code

```
start = strtmant[ch] ;
end = endmant[ch] ;
lowcomp = 0 ;
fgain = fastgain[fgaincod[ch]] ;  /* Table 7.11 */
snroffset[ch] = ((csnroffst – 15) << 4 + fsnroffst[ch]) << 2 ;
```

Initialize as follows for coupling channel:

Pseudo code

```
start = cplstrtmant ;
end = cplendmant ;
fgain = fastgain[cplfgaincod] ;    /* Table 7.11 */
snroffset = ((csnroffst – 15) << 4 + cplfsnroffst) << 2 ;
if (cplleake)
{
    fastleak = (cplfleak << 8) + 768 ;
    slowleak = (cplsleak << 8) + 768 ;
}
```

Initialize as follows for lfe channel:

Pseudo code

```
start = lfestrtmant ;
end = lfeendmant ;
lowcomp = 0 ;
fgain = fastgain[lfefgaincod] ;
snroffset = ((csnroffst – 15) << 4 + lfefsnroffst) << 2 ;
```

7.2.2.2 Exponent Mapping into PSD

This step maps decoded exponents into a 13-bit signed log power-spectral density function.

Pseudo code

```
for (bin=start; bin<end; bin++)
{
    psd[bin] = (3072 – (exp[bin] << 7)) ;
}
```

Since exp[k] assumes integral values ranging from 0 to 24, the dynamic range of the psd[] values is from 0 (for the lowest-level signal) to 3072 for the highest-level signal. The resulting function is represented on a fine-grain, linear frequency scale.

7.2.2.3 PSD Integration

This step of the algorithm integrates fine-grain PSD values within each of a multiplicity of 1/6th octave bands. Table 7.12 contains the 50 array values for bndtab[] and bndsz. The bndtab[] array gives the first mantissa number in each band. The bndsz[] array provides the width of each band in number of included mantissas. Table 7.13 contains the 256 array values for masktab[], showing the mapping from mantissa number into the associated 1/6 octave band number. These two tables contain duplicate information, all of which need not be available in an actual implementation. They are shown here for simplicity of presentation only.

The integration of PSD values in each band is performed with log-addition. The log-addition is implemented by computing the difference between the two operands and

using the absolute difference divided by 2 as an address into a length 256 lookup table, latab[], shown in Table 7.14.

Pseudo code

```
j = start ;
k = masktab[start] ;
do
{
    bndpsd[k] = psd[j] ;
    j++ ;
    for (i = j; i < min(bndtab[k+1], end); i++)
    {
        bndpsd[k] = logadd(bndpsd[k], psd[j]) ;
        j++ ;
    }
    k++ ;
}
while (end > bndtab[k++]) ;
logadd(a, b)
{
    c = a – b ;
    address = min((abs(c) >> 1), 255) ;
    if (c >= 0)
    {
        return(a + latab(address)) ;
    }
    else
    {
        return(b + latab(address)) ;
    }
}
```

7.2.2.4 Compute Excitation Function

The excitation function is computed by applying the prototype masking curve selected by the encoder (and transmitted to the decoder) to the integrated PSD spectrum (bndpsd[]). The result of this computation is then offset downward in amplitude by the fgain and sgain parameters, which are also obtained from the bit stream.

Pseudo code

```
bndstrt = masktab[start] ;
bndend = masktab[end – 1] + 1 ;
if (bndstrt == 0) /* For fbw and lfe channels */
{ /* Note: Do not call calc_lowcomp() for the last band of the lfe channel, (i = 6) */
    lowcomp = calc_lowcomp(lowcomp, bndpsd[0], bndpsd[1], 0) ;
    excite[0] = bndpsd[0] – fastgain – lowcomp ;
    lowcomp = calc_lowcomp(lowcomp, bndpsd[1], bndpsd[2], 1) ;
    excite[1] = bndpsd[1] – fastgain – lowcomp ;
    begin = 7 ;
    for (bin = 2; bin < 7; bin++)
    {
        lowcomp = calc_lowcomp(lowcomp, bndpsd[bin], bndpsd[bin+1], bin) ;
```

Pseudo code

```
            fastleak = bndpsd[bin] – fastgain ;
            slowleak = bndpsd[bin] – slowgain ;
            excite[bin] = max(fastleak – lowcomp, slowleak) ;
            if (bndpsd[bin] <= bndpsd[bin+1])
            {
                begin = bin + 1 ;
                break ;
            }
        }
        for (bin = begin; bin < min(bndend, 22); bin++)
        {
            lowcomp = calc_lowcomp(lowcomp, bndpsd[bin], bndpsd[bin+1], bin) ;
            fastleak –= fastdec ;
            fastleak = max(fastleak, bndpsd[bin] - fastgain) ;
            slowleak –= slowdec ;
            slowleak = max(slowleak, bndpsd[bin] - slowgain) ;
            excite[bin] = max(fastleak – lowcomp, slowleak) ;
        }
        begin = 22 ;
    }
    else /* For coupling channel */
    {
        begin = bndstrt ;
    }
    for (bin = begin; bin < bndend; bin++)
    {
        fastleak –= fastdec ;
        fastleak = max(fastleak, bndpsd[bin] - fastgain) ;
        slowleak –= slowdec ;
        slowleak = max(slowleak, bndpsd[bin] – slowgain) ;
        excite[bin] = max(fastleak, slowleak) ;
    }
    calc_lowcomp(a, b0, b1, bin)
    {
        if (bin < 7)
        {
            if (b0 + 256 == b1) ;
            {
                a = 384 ;
            }
            else if (b0 > b1)
            {
                a = max(0, a – 64) ;
            }
        }
        else if (bin < 20)
        {
            if (b0 + 256 == b1)
            {
                a = 320 ;
            }
            else if (b0 > b1)
            {
                a = max(0, a – 64) ;
```

Pseudo code

```
        }
    }
    else
    {
        a = max(0, a - 128) ;
    }
    return(a) ;
}
```

7.2.2.5 Compute Masking Curve

This step computes the masking (noise level threshold) curve from the excitation function, as shown below. The hearing threshold hth[][] is shown in Table 7.15. The fscod and dbpbcod variables are received by the decoder in the bit stream.

Pseudo code

```
for (bin = bndstrt; bin < bndend; bin++)
{
    if (bndpsd[bin] < dbknee)
    {
        excite[bin] += ((dbknee - bndpsd[bin]) >> 2) ;
    }
    mask[bin] = max(excite[bin], hth[fscod][bin]) ;
}
```

7.2.2.6 Apply Delta Bit Allocation

The optional delta bit allocation information in the bit stream provides a means for the encoder to transmit side information to the decoder which directly increases or decreases the masking curve obtained by the parametric routine. Delta bit allocation can be enabled by the encoder for audio blocks which derive an improvement in audio quality when the default bit allocation is appropriately modified. The delta bit allocation option is available for each fbw channel and the coupling channel.

In the event that delta bit allocation is not being used, and no dba information is included in the bit stream, the decoder must not modify the default allocation. One way to insure this is to initialize the cpldeltnseg and deltnseg[ch] delta bit allocation variables to 0 at the beginning of each frame. This makes the dba processing (shown below) to immediately terminate, unless dba information (including cpldeltnseg and deltnseg[ch]) is included in the bit stream.

The dba information which modifies the decoder bit allocation are transmitted as side information. The allocation modifications occur in the form of adjustments to the default masking curve computed in the decoder. Adjustments can be made in multiples of ±6 dB. On the average, a masking curve adjustment of −6 dB corresponds to an increase of 1 bit of resolution for all the mantissas in the affected 1/6th octave band. The following code indicates, for a single channel, how the modification is performed. The modification

calculation is performed on the coupling channel (where deltnseg below equals cpldeltnseg) and on each fbw channel (where deltnseg equals deltnseg[ch]).

Pseudo code

```
if ((deltbae == 0) || (deltbae == 1))
{
    band = 0 ;
    for (seg = 0; seg < deltnseg+1; seg++)
    {
        band += deltoffst[seg] ;
        if (deltba[seg] >= 4)
        {
            delta = (deltba[seg] – 3) << 7 ;
        }
        else
        {
            delta = (deltba[seg] – 4) << 7 ;
        }
        for (k = 0; k < deltlen[seg]; k++)
        {
            mask[band] += delta ;
            band++ ;
        }
    }
}
```

7.2.2.7 Compute Bit Allocation

The bit allocation pointer array (bap[]) is computed in this step. The masking curve, adjusted by snroffset in an earlier step and then truncated, is subtracted from the fine-grain psd[] array. The difference is right-shifted by 5 bits, thresholded, and then used as an address into baptab[] to obtain the final allocation. The baptab[] array is shown in Table 7.16.

The sum of all channel mantissa allocations in one frame is constrained by the encoder to be less than or equal to the total number of mantissa bits available for that frame. The encoder accomplishes this by iterating on the values of csnroffst and fsnroffst (or cplfsnroffst or lfefsnroffst for coupled and low frequency effects channels) to obtain an appropriate result. The decoder is guaranteed to receive a mantissa allocation which meets the constraints of a fixed transmission bit-rate.

At the end of this step, the bap[] array contains a series of 4-bit pointers. The pointers indicate how many bits are assigned to each mantissa. The correspondence between bap pointer value and quantization accuracy is shown in Table 7.17.

Pseudo code

```
i = start ;
j = bndtab[start] ;
do
{
    mask[j] -= snroffset ;
    mask[j] -= floor ;
    if (mask[j] < 0)
    {
        mask[j] = 0 ;
    }
    mask[j] &= 0x1fe0 ;
    mask[j] += floor ;
    for (k = i; k < min(bndtab[j], end); k++)
    {
        address = (psd[i] - mask[j]) >> 5 ;
        address = min(63, max(0, address)) ;
        bap[i] = baptab[address] ;
        i++ ;
    }
}
while (end > bndtab[j++]) ;
```

7.2.3 Bit Allocation Tables

Table 7.6 Slow decay table, slowdec[].

address	slowdec[address]
0	0x0f
1	0x11
2	0x13
3	0x15

Table 7.7 Fast decay table, fastdec[].

address	fastdec[address]
0	0x3f
1	0x53
2	0x67
3	0x7b

Table 7.8 Slow gain table, slowgain[].

address	slowgain[address]
0	0x540
1	0x4d8
2	0x478
3	0x410

Table 7.9 dB/bit table, dbpbtab[].

address	dbpbtab[address]
0	0x000
1	0x700
2	0x900
3	0xb00

Table 7.10 Floor table, floortab[].

address	floortab[address]
0	0x2f0
1	0x2b0
2	0x270
3	0x230
4	0x1f0
5	0x170
6	0x0f0
7	0xf800

Table 7.11 Fast gain table, fastgain[].

address	fastgain[address]
0	0x080
1	0x100
2	0x180
3	0x200
4	0x280
5	0x300
6	0x380
7	0x400

Table 7.12 Banding structure tables, bndtab[], bndsz[].

band #	bndtab[band]	bndsz[band]
0	0	1
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	1
7	7	1
8	8	1
9	9	1
10	10	1
11	11	1
12	12	1
13	13	1
14	14	1
15	15	1
16	16	1
17	17	1
18	18	1
19	19	1
20	20	1
21	21	1
22	22	1
23	23	1
24	24	1

band #	bndtab[band]	bndsz[band]
25	25	1
26	26	1
27	27	1
28	28	3
29	31	3
30	34	3
31	37	3
32	40	3
33	43	3
34	46	3
35	49	6
36	55	6
37	61	6
38	67	6
39	73	6
40	79	6
41	85	12
42	97	12
43	109	12
44	121	12
45	133	24
46	157	24
47	181	24
48	205	24
49	229	24

Table 7.13 Bin number to band number table, masktab[bin]. bin = (10 × A) + B

	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9
A=0	0	1	2	3	4	5	6	7	8	9
A=1	10	11	12	13	14	15	16	17	18	19
A=2	20	21	22	23	24	25	26	27	28	28
A=3	28	29	29	29	30	30	30	31	31	31
A=4	32	32	32	33	33	33	34	34	34	35
A=5	35	35	35	35	35	36	36	36	36	36
A=6	36	37	37	37	37	37	37	38	38	38
A=7	38	38	38	39	39	39	39	39	39	40
A=8	40	40	40	40	40	41	41	41	41	41
A=9	41	41	41	41	41	41	41	42	42	42
A=10	42	42	42	42	42	42	42	42	42	43
A=11	43	43	43	43	43	43	43	43	43	43
A=12	43	44	44	44	44	44	44	44	44	44
A=13	44	44	44	45	45	45	45	45	45	45
A=14	45	45	45	45	45	45	45	45	45	45
A=15	45	45	45	45	45	45	45	46	46	46
A=16	46	46	46	46	46	46	46	46	46	46
A=17	46	46	46	46	46	46	46	46	46	46
A=18	46	47	47	47	47	47	47	47	47	47
A=19	47	47	47	47	47	47	47	47	47	47
A=20	47	47	47	47	47	48	48	48	48	48
A=21	48	48	48	48	48	48	48	48	48	48
A=22	48	48	48	48	48	48	48	48	48	49
A=23	49	49	49	49	49	49	49	49	49	49
A=24	49	49	49	49	49	49	49	49	49	49
A=25	49	49	49	0	0	0				

Table 7.14 Log-addition table, latab[val]. val = (10 × A) + B

	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9
A=0	0x0040	0x003f	0x003e	0x003d	0x003c	0x003b	0x003a	0x0039	0x0038	0x0037
A=1	0x0036	0x0035	0x0034	0x0034	0x0033	0x0032	0x0031	0x0030	0x002f	0x002f
A=2	0x002e	0x002d	0x002c	0x002c	0x002b	0x002a	0x0029	0x0029	0x0028	0x0027
A=3	0x0026	0x0026	0x0025	0x0024	0x0024	0x0023	0x0023	0x0022	0x0021	0x0021
A=4	0x0020	0x0020	0x001f	0x001e	0x001e	0x001d	0x001d	0x001c	0x001c	0x001b
A=5	0x001b	0x001a	0x001a	0x0019	0x0019	0x0018	0x0018	0x0017	0x0017	0x0016
A=6	0x0016	0x0015	0x0015	0x0015	0x0014	0x0014	0x0013	0x0013	0x0013	0x0012
A=7	0x0012	0x0012	0x0011	0x0011	0x0011	0x0010	0x0010	0x0010	0x000f	0x000f
A=8	0x000f	0x000e	0x000e	0x000e	0x000d	0x000d	0x000d	0x000d	0x000c	0x000c
A=9	0x000c	0x000c	0x000b	0x000b	0x000b	0x000b	0x000a	0x000a	0x000a	0x000a
A=10	0x000a	0x0009	0x0009	0x0009	0x0009	0x0009	0x0008	0x0008	0x0008	0x0008
A=11	0x0008	0x0008	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0006	0x0006
A=12	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0005	0x0005	0x0005	0x0005
A=13	0x0005	0x0005	0x0005	0x0005	0x0004	0x0004	0x0004	0x0004	0x0004	0x0004
A=14	0x0004	0x0004	0x0004	0x0004	0x0004	0x0003	0x0003	0x0003	0x0003	0x0003
A=15	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0002
A=16	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002
A=17	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0001	0x0001
A=18	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001
A=19	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001
A=20	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001
A=21	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
A=22	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
A=23	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
A=24	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
A=25	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000

Table 7.15 Hearing threshold table, hth[fscod][band].

band #	hth[0][band] (fs=48 kHz)	hth[1][band] (fs=44.1 kHz)	hth[2][band (fs=32 kHz)
0	0x04d0	0x04f0	0x0580
1	0x04d0	0x04f0	0x0580
2	0x0440	0x0460	0x04b0
3	0x0400	0x0410	0x0450
4	0x03e0	0x03e0	0x0420
5	0x03c0	0x03d0	0x03f0
6	0x03b0	0x03c0	0x03e0
7	0x03b0	0x03b0	0x03d0
8	0x03a0	0x03b0	0x03c0
9	0x03a0	0x03a0	0x03b0
10	0x03a0	0x03a0	0x03b0
11	0x03a0	0x03a0	0x03b0
12	0x03a0	0x03a0	0x03a0
13	0x0390	0x03a0	0x03a0
14	0x0390	0x0390	0x03a0
15	0x0390	0x0390	0x03a0
16	0x0380	0x0390	0x03a0
17	0x0380	0x0380	0x03a0
18	0x0370	0x0380	0x03a0
19	0x0370	0x0380	0x03a0
20	0x0360	0x0370	0x0390
21	0x0360	0x0370	0x0390
22	0x0350	0x0360	0x0390
23	0x0350	0x0360	0x0390
24	0x0340	0x0350	0x0380

band numbe	hth[0][band (fs=48 kHz)	hth[1][band] (fs=44.1 kHz)	hth[2][band] (fs=32 kHz)
25	0x0340	0x0350	0x0380
26	0x0330	0x0340	0x0380
27	0x0320	0x0340	0x0370
28	0x0310	0x0320	0x0360
29	0x0300	0x0310	0x0350
30	0x02f0	0x0300	0x0340
31	0x02f0	0x02f0	0x0330
32	0x02f0	0x02f0	0x0320
33	0x02f0	0x02f0	0x0310
34	0x0300	0x02f0	0x0300
35	0x0310	0x0300	0x02f0
36	0x0340	0x0320	0x02f0
37	0x0390	0x0350	0x02f0
38	0x03e0	0x0390	0x0300
39	0x0420	0x03e0	0x0310
40	0x0460	0x0420	0x0330
41	0x0490	0x0450	0x0350
42	0x04a0	0x04a0	0x03c0
43	0x0460	0x0490	0x0410
44	0x0440	0x0460	0x0470
45	0x0440	0x0440	0x04a0
46	0x0520	0x0480	0x0460
47	0x0800	0x0630	0x0440
48	0x0840	0x0840	0x0450
49	0x0840	0x0840	0x04e0

Table 7.16 Bit allocation pointer table, baptab[].

address	baptab[address]	address	baptab[address]
0	0	32	10
1	1	33	10
2	1	34	10
3	1	35	11
4	1	36	11
5	1	37	11
6	2	38	11
7	2	39	12
8	3	40	12
9	3	41	12
10	3	42	12
11	4	43	13
12	4	44	13
13	5	45	13
14	5	46	13
15	6	47	14
16	6	48	14
17	6	49	14
18	6	50	14
19	7	51	14
20	7	52	14
21	7	53	14
22	7	54	14
23	8	55	15
24	8	56	15
25	8	57	15
26	8	58	15
27	9	59	15
28	9	60	15
29	9	61	15
30	9	62	15
31	10	63	15

Table 7.17 Quantizer levels and mantissa bits vs bap.

bap	quantizer levels	mantissa bits (group bits / num in group)
0	0	0
1	3	1.67 (5/3)
2	5	2.33 (7/3)
3	7	3
4	11	3.5 (11/2)
5	15	4
6	32	5
7	64	6
8	128	7
9	256	8
10	512	9
11	1024	10
12	2048	11
13	4096	12
14	16,384	14
15	65,536	16

7.3 Quantization and Decoding of Mantissas

7.3.1 Overview

All mantissas are quantized to a fixed level of precision indicated by the corresponding bap. Mantissas quantized to 15 or fewer levels use symmetric quantization. Mantissas quantized to more than 15 levels use asymmetric quantization which is a conventional two's complement representation.

Some quantized mantissa values are grouped together and encoded into a common codeword. In the case of the 3-level quantizer, 3 quantized values are grouped together and represented by a 5-bit codeword in the data stream. In the case of the 5-level quantizer, 3 quantized values are grouped and represented by a 7-bit codeword. For the 11-level quantizer, 2 quantized values are grouped and represented by an 11-bit codeword.

In the encoder, each transform coefficient (which is always < 1.0) is left justified by shifting its binary representation left the number of times indicated by its exponent (0 to 24 left shifts). The amplified coefficient is then quantized to a number of levels indicated by the corresponding bap.

The following table indicates which quantizer to use for each bap. If a bap equals 0, no bits are sent for the mantissa. Grouping is used for baps of 1, 2 and 4 (3, 5, and 11 level quantizers.)