音響 OFDM 技術仕様 Ver. 1.00

目次

1. はじめに

NTT ドコモでは, オーディオ信号に OFDM(Orthogonal Frequency Division Multiplexing)を利用しデータを多重する音響 OFDM（サービス仮称：音のバーコード）の技術を開発した．本稿では，音響 OFDM の技術仕様を記述する．

2. 適用範囲

この規格は,「音響 OFDM」として知られるオーディオ信号体系の技術的要件を規定する．ここでは,「音響 OFDM」の体系の特徴，データのオーディオ信号による伝送形式，エンコード及びデコードの手法を規定する．

3. 定義

この規格で用いる主な用語の定義は，次による．

a) データ(Data)

伝送する情報．通常，文字情報が伝送される．URL やメールアドレス，ID なども含まれる．

b) オーディオ信号(Audio Signal)

スピーカで再生し，マイクロホンで受信できる形式の信号．**可聴周波数帯域**（20Hz～20kHz）を含む音楽や音声などの信号が望ましい．本稿では，ディジタル上は主に 16bit, サンプリング周波数 44.1kHz の **PCM**（Pulse Code Modulation）形式の信号.

c) オリジナル音源(Sound Source)

エンコーダに入力するオーディオ信号．本稿では，データが重畳されていないオーディオ信号.

d) OFDM(Orthogonal Frequency Division Multiplexing)

直交周波数分割多重方式．本規格ではベースバンドで多重する．

e) サブキャリア(Sub-Carrier)

OFDM のキャリア．本規格では，伝送する周波数帯域のサブキャリアに振幅をもたせ，**D-BPSK**（Differential Binary Phase-Shift Keying：差動 2 相位相偏移変調）する．

f) サンプリング周波数(Sampling Frequency)

オーディオ信号を，サンプリングする周波数．通常，本規格では CD 等でも利用されている 44.1kHz を用いる．

g) 変調方式(Modulation Type)

データを波形に載せる方式．通常，本規格では各サブキャリアを D-BPSK で変調する．

h) エンコーダ(Encoder)

音響 OFDM でデータをオーディオ信号に重畳し，重畳されたオーディオ信号を出力するソフトウェアまたは機器.

i)　デコーダ(Decoder)

音響 OFDM でデータを重畳されたオーディオ信号から，データを再構成するソフトウェアまたは機器.

j)　再生器（スピーカ：Loudspeaker）

オーディオ信号を音波に変換し出力する機器．本稿では，通常，**DA**（Digital to Analog Converter）とアンプ，電気音響変換器（通称：スピーカ）を含む.

k)　録音器（マイクロホン：Microphone）

音波をデジタル信号に変換する機器．本稿では，通常，音響電気変換器（通称：マイクロホン），アンプ，**AD**（Analog to Digital Converter）を含む.

l)　OFDM 伝送周波数帯域（OFDM Frequency Band）

データを重畳するサブキャリアを配置する周波数帯域. 本稿では, 通常, 6.4kHz～8.0kHz の周波数帯域.

m）帯域除去オーディオ信号（Band Elliminated Signal）

OFDM 伝送周波数帯域の振幅が低減された音響 OFDM のオーディオ信号．本稿では，通常，OFDM 伝送周波数帯域を除去したオーディオ信号をいう.

n）OFDM 信号（OFDM Signal）

位相変調したサブキャリアからなる OFDM 伝送周波数帯域の音響 OFDM のオーディオ信号．本規格では，通常，オリジナル音源由来の成分を含まない.

o)　合成オーディオ信号（Output Audio Signal）

バンドカットオーディオ信号と，OFDM 信号，OFDM フレーム同期信号を含むオーディオ信号．通常，エンコーダから出力されるオーディオ信号.

p)　OFDM 送信シンボル（OFDM Transmission Symbol）

OFDM シンボルと GI，OFDM フレーム境界を含む区間．本規格では 44.1kHz サンプリングで 2032 サンプルを設定.

q)　OFDM データシンボル(OFDM Data Symbol)

OFDM で FFT する区間．　GI や OFDM フレーム境界は含まない．本規格では 44.1kHz サンプリングで 1024 サンプルを設定.

r)　GI(Guard Interval)

Cyclic Prefix の区間．OFDM シンボルの前段部分の複製を前段に配置する．本規格では 44.1kHz サンプリングで 800 サンプルを設定.

s)　OFDM シンボル同期信号(OFDM Symbol Synchronization Signal)

OFDM フレームの区間を判別するために挿入される信号．本規格では，低い周波数帯域に，M 系列による拡散信号を挿入する．OFDM 送信シンボルの先頭部分に配置される.

t)　データフレーム（Data Frame）

データを複数の固定長サイズに分割した際のひとつの分割データ．本規格では，1 データフレームは 45 バイトの固定長サイズで構成される.

u) 誤り訂正（Error Correction）

誤りが生じたかどうかを発見し、誤りがあれば元のデータを再現する処理．誤り訂正符号(Error Correction Code)ともいう．誤り訂正符号は，本来のデータとは別に付加される冗長なデータをいう場合もある．本規格では，通常，畳み込み符号と軟判定を用いる．

v) 巡回冗長検査（Cyclic Redundancy Check）

ある種の多項式の演算に基づいた巡回冗長符号をデータに付加して誤りの検出と訂正を行う方式．計算は，元の 2 進コードをある種の生成多項式で割り，その余りを巡回符号として付加する．

w) マスキング（Masking）

ある音の聞き取りが，別の音の存在によって妨害を受ける現象．その中で，通常，大きな音の周波数の近傍の周波数の音は知覚できない現象を周波数マスキング（Frequency Masking），大きな音の時間の近傍の時間の音は知覚できない現象を時間マスキング（Time Masking）という．

x) マスキング閾値(Masking Threshold)

妨害音の存在により聞き取りたい信号音の最小可聴値が上昇し，その上昇した値.

y) パディングビット（Padding Bit）

データビット列の終端パターンの後ろにある最終コード語の空の位置を充てんする目的で使用する．データを示さないビット.

4. 記法

a) 周波数（Frequency）

周波数は，Hz，kHz 単位で示す.

b) 時間（Time）

時間は，秒単位か，サンプリング周波数 44.1kHz におけるサンプル単位で示す.

c) バイト表記

バイト（Byte）の内容は，16 進数で示す.

d) 音圧レベル（Sound Pressure Level）

音圧レベルは，dBSPL 単位で示す．ある音の音圧の実効値の 2 乗と，基準音圧（20 μ Pa：マイクロ・パスカル）の 2 乗との比の常用対数.

e) 音量（Volume）

音量は，A 特性音圧レベルを利用し，dB(A) 単位で示す．人間の聴覚が周波数によって異なる性質を考慮し、補正を行った聴感特性補正値．一般に騒音レベルともいう．1kHz では，dB(A)単位と dBSPL 単位は同量．一般に，人の会話は 60dB(A)，ステレオ（正面 1m、夜間）が 70dB(A)程度．通常，10dB 上がれば，人には倍程度の大きさに聞こえる.

5. 技術仕様

5.1. エンコード及びデコード手順の概要

エンコード及びデコードの手順を図 1 に示す.

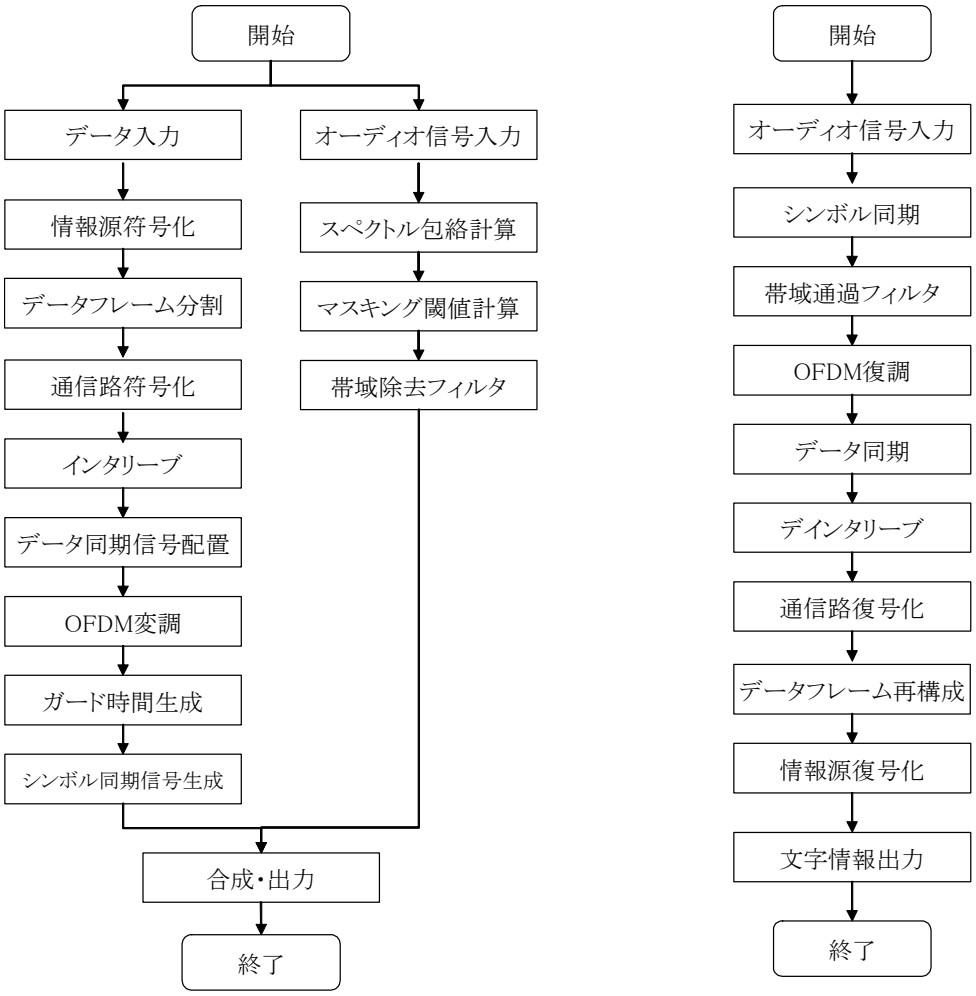

図 1. 処理手順

エンコーダにはオリジナル音源となるオーディオ信号と，それに載せたいテキストなどのデータを入力し，データを重畳したオーディオ信号が出力される．デコーダには，マイクから録音したオーディオ信号が入力され，重畳されているデータを抽出してデータを出力する．以降では，エンコーダ・デコーダの各ステップごとに，より詳細な処理手順を説明する．また表 1 には本規格の伝送パラメータを示す.

表 1.　伝送パラメータ

ファイルパラメータ	規定	任意
サンプリング周波数	44.1kHz	
量子化ビット数	16bit	
OFDM パラメータ		
サブキャリア変調方式	D-BPSK	
フレーム長	2032 サンプル	
シンボル長	1024 サンプル	
ガード時間	1008 サンプル	
サブキャリア数	36	
低域上側帯域端周波数	5512.5 Hz	
高域下側帯域端周波数	6416.9 Hz	
サブキャリア（最低）振幅		-70dBfs を推奨
シンボル同期信号拡散率	127	
シンボル同期信号チップレート	2756 Hz	
シンボル同期信号生成多項式	x^7+x+1	
シンボル同期信号振幅		周波数マスキング閾値の -3dB～+3dB の範囲を推奨
情報系列長	360	
誤り訂正符号　符号化率	1/3	
誤り訂正符号　拘束長	6	
誤り検出用 CRC 長	7	
誤り検出用 CRC 多項式	x^7+x+1	

5.2. 情報源符号化 / 復号化

情報源符号化では，伝送する文字情報を伝送ビット列に変換する。変換方式はアプリケーション依存とする。基本的には文字コードのビット列をそのまま伝送ビット列とする．文字情報を圧縮して伝送する場合は，JIS X0610(2004)に則り，変換モードを選択してビット列に変換することも可能である．

5.3. データフレーム分割 / 再構成

情報源符号化されたデータを固定長サイズのデータフレームに分割して伝送するための送信側における分割処理，受信側における再構成処理を規定する．また，分割データの消失を訂正するために，Reed-Solomon 符号による冗長フレームを生成する．図 2 に分割処理および冗長フレーム生成の概念を示す．

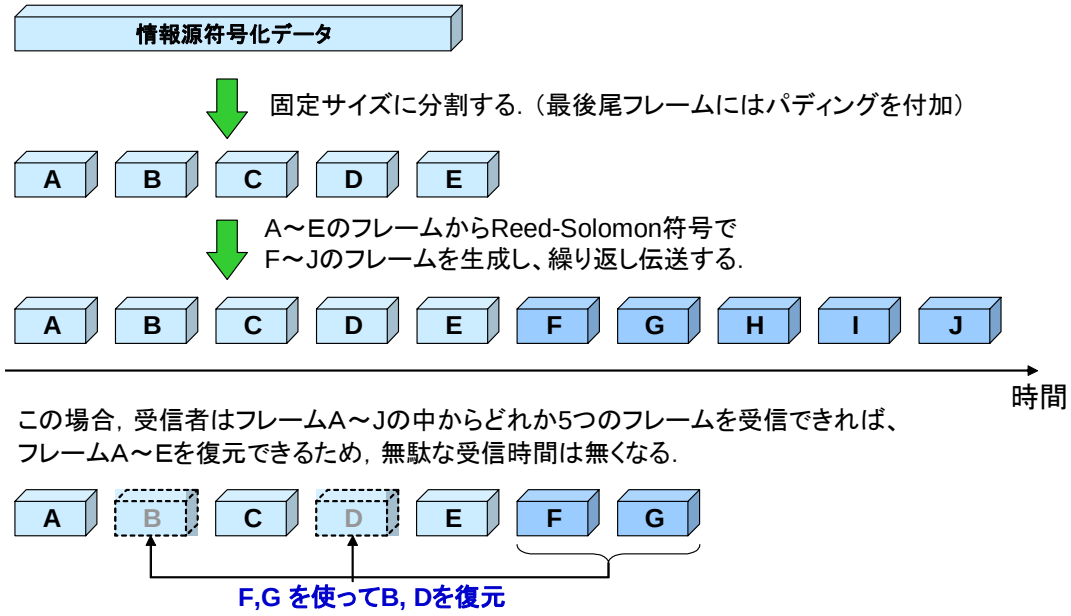

図 2．データフレーム分割・冗長フレーム生成

まず，データを固定長サイズに分割する．余ったデータもパディング情報を付加して固定サイズに揃えてフレームＡ～Ｅを生成する．次にデータフレームＡ～Ｅから Reed-Solomon 符号で冗長フレームＦ～Ｊを生成する．Reed-Solomon 符号のアルゴリズムについては，付属書 14 章に示す．Ａ～Ｊのフレームの各々に 1 バイトのデータフレームヘッダを付加し、再構成するための情報を格納する．データフレームヘッダについては図 3 にて後述する．受信側では，A～J のフレームの中から分割フレーム数に相当する数（この例では 5 個）の任意のフレームを受信すれば A～E のデータフレームを復元できる．この

例では，A，C，E，F，G のフレームから B，D のフレームを復元して A～E のフレームを再構成する．

図 3 にデータフレームヘッダの構成を示す．データフレームヘッダは 1 バイトで構成されフレームの先頭に付加される．1 バイトの領域は，予約ビット（R:1 ビット），FEC フラグ（F:1 ビット），フレーム数（3 ビット），シーケンス番号（3 ビット）で構成される．予約ビットは将来の拡張のために確保し，通常は常に 0 を格納する．FEC フラグは，データフレームが分割フレームか冗長フレームかを判別するためのビットであり，分割フレームの場合は 0，冗長フレームの場合は 1 を格納する．フレーム数（3 ビット）には，分割フレーム数を挿入する．最大 7 分割まで可能になる．シーケンス番号（3 ビット）には，分割フレームの場合は先頭から何番目のフレームかを示し，冗長フレームの場合には Reed-Solomon 符号の符号化に利用する生成行列（付属書 12 章に記載）の何行目の計算で生成されたかを示す．

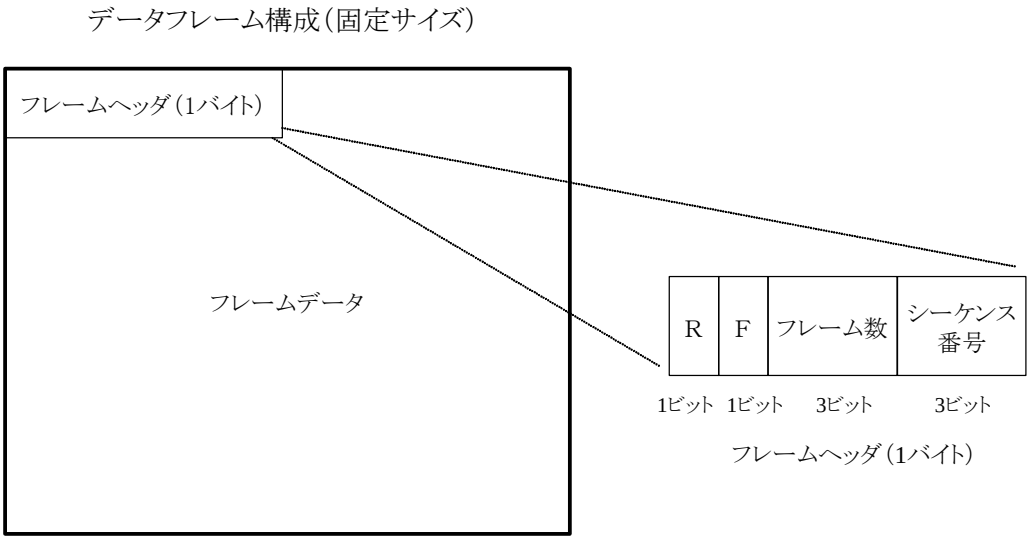

図 3．データフレームヘッダ

受信側における再構成では，受信したフレームを保存しておき，分割フレームおよび冗長フレームの合計数が，フレームヘッダのフレーム数に格納されている数に達した時点で，Reed-Solomon 復号をし，分割フレームを復元する．復元した分割フレームを再構成し，情報元符号化データを生成する．

Reed-Solomon 符号化には下記に示す Vandermonde 行列を用いる．この行列の n 行目は，$(1^n, 2^n, 3^n, 4^n, \ldots)$ となっている．各分割フレームを先頭から 1 バイトずつ取り出し，冗長フレームも 1 バイトずつ生成していく．下記に 4 つの分割フレームから 4 つの冗長フレームを生成する例を示す．4 つの分割フレームから 1 バイトずつのデータを取り出し，それぞ

れを(x_1, x_2, x_3, x_4)とする．このベクトルを Vandermonde 行列に掛けて算出されたベクトルは(f_1, f_2, f_3, f_4)となり，それぞれの項が各冗長フレームの 1 バイトのデータとなる．先頭のバイトデータから最終のバイトデータまで 1 バイトずつ同様の処理で符号化をする．

Encoder　　※計算は全てGF(256)のガロア体で演算する.

$$\begin{bmatrix} 1^1 & 2^1 & 3^1 & 4^1 \\ 1^2 & 2^2 & 3^2 & 4^2 \\ 1^3 & 2^3 & 3^3 & 4^3 \\ 1^4 & 2^4 & 3^4 & 4^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}$$

$x_1, x_2, x_3, x_4, f_1, f_2, f_3, f_4$ のどの4つのシンボルからでも x_1, x_2, x_3, x_4 を復元できる

Decoder　　例えば f_1, f_2, x_2, f_4 から復元

$$\begin{bmatrix} 1^1 & 2^1 & 3^1 & 4^1 \\ 1^2 & 2^2 & 3^2 & 4^2 \\ 0 & 1 & 0 & 0 \\ 1^4 & 2^4 & 3^4 & 4^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ x_2 \\ f_4 \end{bmatrix}$$

左側の行列の逆行列を計算し、
右側のベクトルに掛けると、
x_1, x_2, x_3, x_4 が求まる.

データフレームのヘッダには各々シーケンス番号が付加される．分割フレームの場合は，先頭から何番目のフレームかを示し，冗長フレームの場合は Vandermonde 行列の何行目で演算されて生成されたデータかを示す．上記の例の場合，f_1 のシーケンス番号は 1 に，f_2 のシーケンス番号は 2 になる．受信側における復号化処理では，上記の例に示すとおり，受信したフレームを並べて，それに相当する生成行列を作り，その生成行列の逆行列を計算することで(x_1, x_2, x_3, x_4)を復元できる．計算は全て GF(256)のガロア体で行う．

5.4. 通信路符号化 / 復号化

データフレームの伝送において，ビット誤りを検出および訂正する処理が必要になる．本規格では，誤り検出に CRC，誤り訂正符号に畳み込み符号，誤り訂正復号に軟判定ビタビ復号を採用している．

CRC ビットは付加する CRC ビット長に応じて表 2 で示す生成多項式を用いて生成する．本規格では 7 ビットの CRC を付加する．CRC の符号化では，情報源符号化された情報ビット列に対して生成多項式のビット列で除算した剰余ビット列を後方に付加する．誤り検出処理では，受信した情報ビット列を生成多項式のビット列で除算し，余りが存在したら情報ビット列に誤りがあると判定する．

表 2.　生成多項式（CRC）

CRC 長	生成多項式
6 ビット	$x^6 + x + 1$
7 ビット	$x^7 + x + 1$
8 ビット	$x^8 + x^4 + 1$
9 ビット	$x^9 + x^4 + 1$
10 ビット	$x^{10} + x^3 + 1$

誤り訂正の符号化手順としては，まず CRC ビットを付加した情報ビット系列に対して，畳込み符号の終端コードを付加する．畳込み符号の拘束長を K とすると，K-1 ビットの 0 を終端に付加することになる．この終端コードを付加したビット系列を畳込み符号器に入力し，出力されたビットを整列して符号化ビット系列を生成する．畳込み符号器の生成多項式には，拘束長，符号化率に応じて，表 3 に示す生成多項式を用いる．本規格では，拘束長 6，符号化率 1/3 の生成多項式を用いて符号化を行う．

表 3.　生成多項式（畳込み符号）

拘束長	符号化率	生成多項式
4	1/2	$1+D+D^3,\quad 1+D+D^2+D^3$
4	1/3	$1+D^2+D^3,\quad 1+D+D^3,\quad 1+D+D^2+D^3$
5	1/2	$1+D^3+D^4,\quad 1+D+D^2+D^4$
5	1/3	$1+D^2+D^4,\quad 1+D+D^3+D^4,\quad 1+D+D^2+D^3+D^4$
6	1/2	$1+D+D^3+D^5,\quad 1+D^2+D^3+D^4+D^5$
6	1/3	$1+D^3+D^4+D^5,\quad 1+D^2+D^4+D^5,\quad 1+D+D^2+D^3+D^5$

誤り訂正の復号化手順としては，受信した信号系列に対して，符号化で行われたインタリーブを元に戻すデインタリーブを行う．インタリーブ・デインタリーブについては，次節で記述する．このデインタリーブした信号系列に対して軟判定ビタビ復号を行う．まず，OFDM の各サブキャリアの D-BPSK 変調を復調する際に，軟判定ビタビ復号に用いるブランチメトリックを計算する．ブランチメトリックの計算には次式を用いる．

$$\lambda(1) = \mathrm{Re}[z_n \cdot z_{n-1}] \qquad \lambda(-1) = -\mathrm{Re}[z_n \cdot z_{n-1}]$$

ここで，z_n は，時刻 n におけるキャリアの振幅を $(x^2+y^2)^{0.5}$ で，絶対位相を $\tan^{-1}(x/y)$. で表せる複素数 $x+jy$ である．パスメトリックはブランチメトリックの合計となり，パスメトリックの最も大きいパスが軟判定ビタビ復号の出力ビット系列となる．

この出力ビット系列から畳込み符号の終端コードを削除したビット系列に対して，CRC の誤り検出を行い，誤りが無ければ正常な情報ビット系列として出力する．

5.5. インタリーブ / デインタリーブ

OFDM 変調では，符号化系列を時間方向と周波数方向に配置するが，時間方向もしくは周波数方向に順番に配置すると，ある時間もしくはある周波数においてフェージングが生じた際に，その部分にバースト的なエラーが生じてしまうため，以下に示す 2 つのインタリーブ（配置変換）を組み合わせてランダム性を高める。下記の例では縦軸が周波数軸で横軸が時間軸である．インタリーブ前のビット配置は，符号化ビット列を周波数軸方向に低周波数から高周波数方向に順番に並べ，最高周波数までビットを配置した後，次の時間スロットの低周波数から高周波数方向に順番に並べていく．

a) インタリーブ 1

このインタリーブでは，奇数番目のシンボルを先頭から並べ，偶数番目のシンボルを後方から並べる配置変換を行う．

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

インタリーブ前

1	11	20	10
3	13	18	8
5	15	16	6
7	17	14	4
9	19	12	2

インタリーブ後

b) インタリーブ 2

このインタリーブでは，符号化系列をある間隔おきに配置する．この例ではビット配列を 4 シンボルおきに配置変換する．

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

インタリーブ前

1	7	13	19
6	12	18	5
11	17	4	10
16	3	9	15
2	8	14	20

インタリーブ後

本規格では，周波数軸方向に 36，時間軸方向に 31 のビット配列を用い，初期のビット配置は上記の例の通り，符号化ビット列を周波数軸方向に低周波数から高周波数方向に順番に並べ，最高周波数までビットを配置した後，次の時間スロットの低周波数から高周波数方向に順番に並べていく．このビット配列に上記のインタリーブを，インタリーブ 1，インタリーブ 2 の順番で適用する．インタリーブ 2 では，ビット列を 31 シンボルおきに配置変換する．復号には，上記のインタリーブの逆変換であるデインタリーブをインタリーブ 2、インタリーブ 1 の順番で行う．

5.6. データフレーム同期信号 / データフレーム同期処理

本規格におけるデータフレームは，図 4 に示す構造になる．時間軸方向に 31 の OFDM フレームと，周波数軸方向に 37 のサブキャリアから構成される．最低周波数の 1 サブキャリア（図では，サブキャリア 0）にデータ同期用の符号系列を配置する．その上の周波数の 36 のサブキャリア（図では，サブキャリア 1～36）にデータのビット系列を配置する．1 つのサブキャリア，1 つの OFDM フレームで 1 ビットの情報を配置するため，本規格では 1 つのデータフレームで 36x31 の 1116 ビットを配置する．

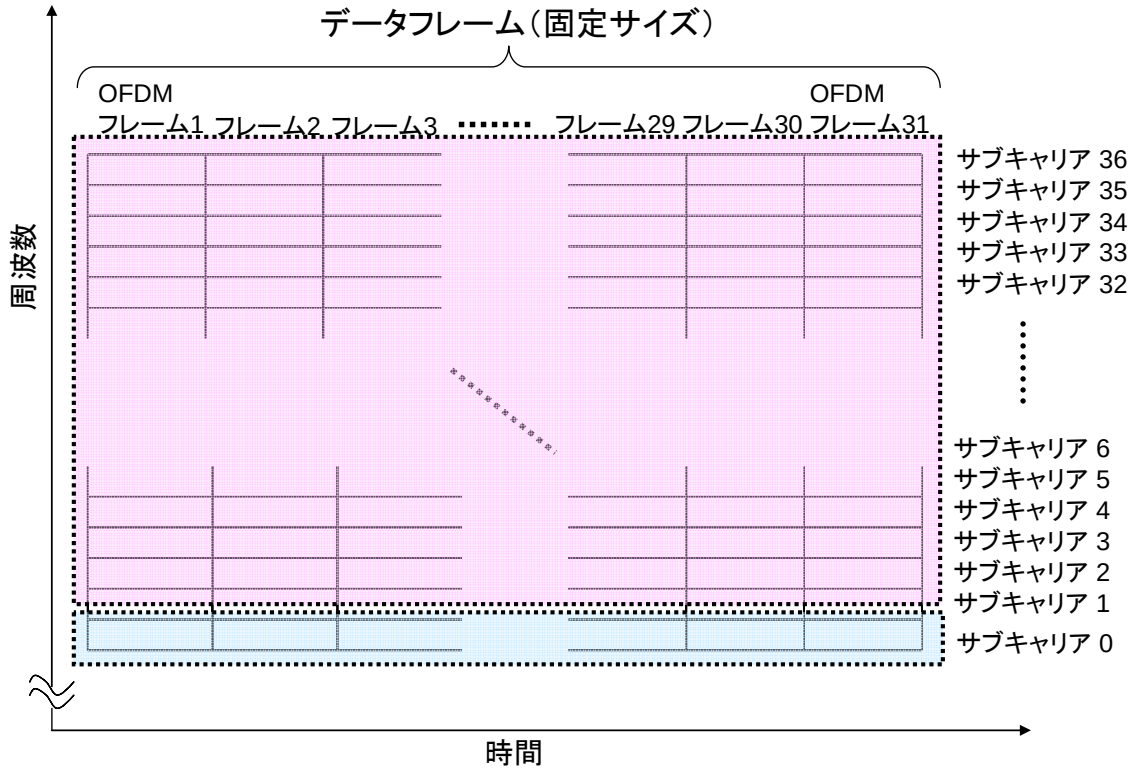

図 4．データフレームの基本構造

データフレームの先頭を識別するために，データ同期処理が必要となる．また，符号系列の種類によってシングルフレームの繰り返し伝送か，マルチフレームの伝送かを識別

できるようにする．本規格では以下に示す 2 種類の符号系列でシングルフレーム伝送かマルチフレーム伝送かを識別する．

【シングルフレーム伝送用】

0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1

【マルチフレーム伝送用】

0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1

どちらも(n = 31, k = 6, d = 15)の BCH 符号で符号化された符号長 31 の符号系列である．BCH 符号は巡回符号であるため，系列が巡回していても符号語になっており，誤り訂正の復号が可能である．このデータ同期用の符号系列は繰り返し伝送されるため，受信側でどのビットから受信を開始しても 31 ビット分の符号系列を受信すれば誤り訂正の復号が可能である．誤り訂正の復号には以下の生成符号系列を用いる．ここで，符号系列の先頭ビットの定義を決めることで，受信側でデータ同期用の符号系列が何ビット巡回しているかが判別できるようになる．先頭ビットの定義は符号系列の中で 0 が最も長く続く箇所の最初の 0 を符号系列の先頭ビットとする．上記に記載の符号系列は先頭ビットから記載されている．送信側で，この先頭ビットとデータフレームの先頭を合わせて配置しておくことで，受信側ではデータ同期用符号系列が何ビット巡回しているかを検出し，データフレームの先頭を識別できるようになる．BCH 符号の誤り訂正復号には以下の生成符号系列を用いる．

【BCH 符号生成符号系列】

1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0

シングルフレーム伝送の場合，データは分割されず，常に同一のデータフレームが伝送されるため，シーケンス番号や冗長フレームが不要である．したがって，データフレーム内にフレームヘッダは付加されずにデータだけが格納される．

5.7. スペクトル包絡計算

入力されたオーディオ信号をフーリエ変換し，オーディオ信号の周波数スペクトルを計算する．フーリエ変換は，OFDM のシンボル長と同じサンプル数の時間信号に対して適用する．本規格では，OFDM のシンボル長が 44.1kHz サンプリングで 1024 サンプルなので，フーリエ変換も 1024 サンプルの時間信号に対して行う．フーリエ変換すると 0～44.1kHz までの 1024 種類の周波数情報（振幅，位相）が得られるが，ナイキスト周波数で折り返して同じ情報になるため，情報量としては，0～22.05kHz までの 512 種類の周波数情報となる．ここで算出した周波数情報の振幅情報は，OFDM 変調やマスキング閾値計算に用いる．

5.8. OFDM 変調

インタリーブしたビット配列に基づき，サブキャリアを D-BPSK で変調する．D-BPSK 変調では，ひとつ前の OFDM シンボルの同サブキャリアにおける位相との相対位相差で情報ビットを識別できるようにする．伝送するビットが 0 の場合には前シンボルの位相をπシフトさせる．伝送するビットが 1 の場合には前シンボルと同位相にする．エンコードの最初の OFDM シンボルに関してはランダムなビット列（任意）を配置してよい．OFDM のデータフレームは繰り返し伝送されるので，2 周目以降のデータフレームの先頭フレームは 1 つ前のデータフレームの最終フレーム（31 フレーム目）を参照して位相を決定する．

次に OFDM 変調信号がオーディオ信号に調和するように各サブキャリアの振幅を調節する．振幅の調節にはオーディオ信号から計算したスペクトル包絡情報を用いる．図 5 に音響 OFDM のエンコード方式の基本的な概念を示す．まず，①オリジナル音源のオーディオ信号を解析しスペクトル包絡情報を求める．その後，帯域阻止フィルタで OFDM 伝送周波数帯域の成分を除去する．②は OFDM 伝送周波数帯域を除去した帯域除去オーディオ信号である．次に，入力されたデータに基づき OFDM 伝送周波数帯域のサブキャリアを変調した③OFDM 変調信号を生成する．この OFDM 変調信号のサブキャリアのパワーを，①から得たオリジナル音源のスペクトル包絡に合わせて調節し，④振幅調整した信号を生成する．最後に，②帯域除去オーディオ信号と④OFDM 伝送周波数帯域信号を合成して⑤合成オーディオ信号を生成する．これにより，OFDM 変調信号を聴覚に不快にならないように変形して音声・音楽に重畳できる．ノイズとの振幅比を維持するために，スペクトル包絡でパワーが少ないサブキャリアについては，サブキャリアの振幅をスペクトル包絡以上に設定しておくことで受信耐性を高くできる．

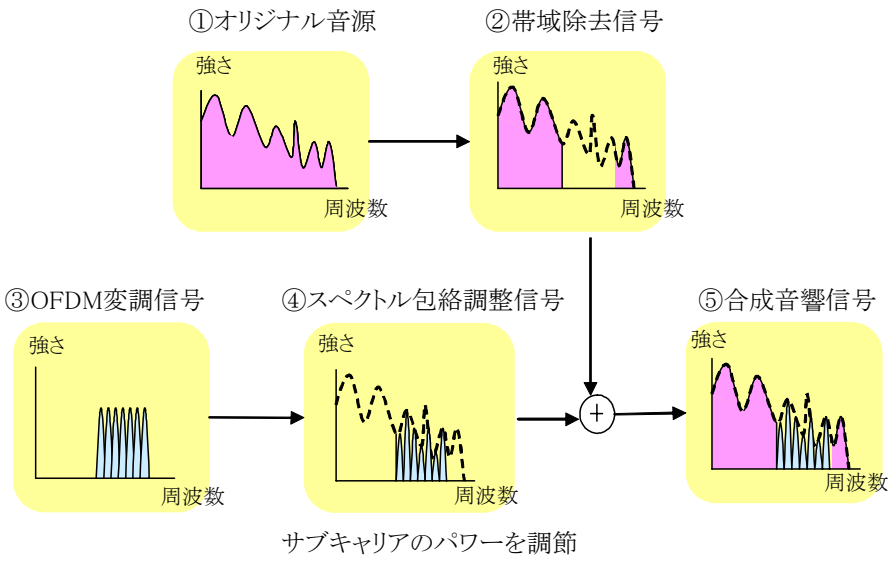

図 5. 音響 OFDM エンコード方式

5.9. ガード時間生成

OFDM にはガード時間を挿入することで反射波等によるマルチパス干渉に対処できるという特徴もある．図 3 の(a)に示すように，各 OFDM フレームは OFDM シンボル区間とガード時間の区間で構成される．ガード時間の区間はデータシンボルの後方を複製（Cyclic prefix）して生成される．これにより，ガード時間と OFDM フレームは位相が連続になり，この区間長以内で遅延波が重なっても直交性を保つことができる．このガード時間により，スピーカの減衰振動や反射波による遅延波にも対応でき，ロバストなデータ伝送が可能になる．

図 6 の(a)で伝送性能としては十分であるが，ガード時間と前 OFDM フレームの境界は位相が不連続になり，音響 OFDM では，これらの伝送信号が人間の耳に聞こえるため，位相不連続な部分が不快な音に聞こえてしまう場合がある．そのため，音響 OFDM では，図 6 の(b)に示すように後方にもガード時間を挿入する．後方のガード時間は OFDM シンボルの前方をコピーして生成する．この後方ガード時間と，ひとつ後ろの OFDM フレームの前方ガード時間をクロスフェードさせてオーバーラップすることで，フレーム間の位相の不連続を軽減する．本規格では，OFDM シンボルが 44.1kHz で 1024 サンプル，ガード時間が 1008 サンプルで OFDM フレーム長は 2032 サンプルとなる．後方ガード時間のサンプル数は任意であるが 200 サンプル程度を推奨する．また，後方ガード時間の代わりにマスカー音を挿入して位相不連続点で生ずるノイズを時間マスキングすることもできる．このマスカー音として， 単一または複数の周波数の正弦波を挿入し，この周波数をフレーム境界毎に変化させることによって，メロディを構成することもできる．このメロディを合図音として，伝送信号が入っていることをユーザに気づかせるために使うこともできる．

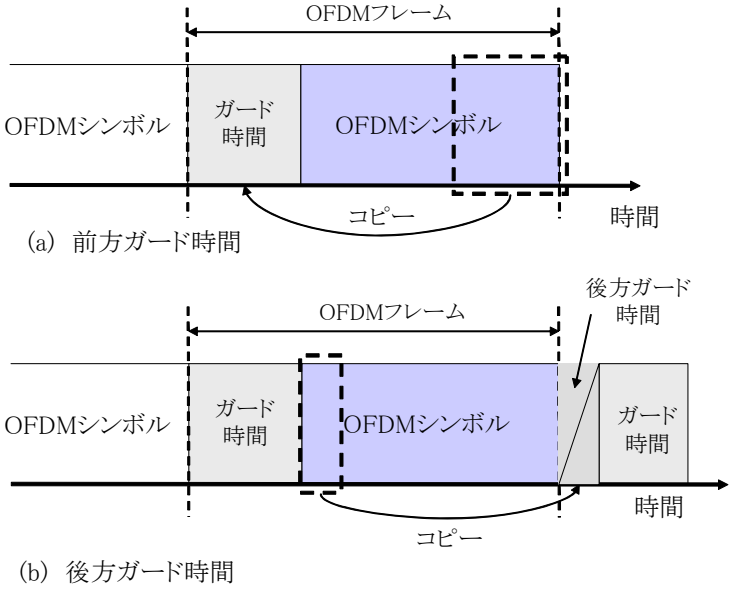

図 6. 音響 OFDM ガード時間

5.10. マスキング閾値計算

OFDM のフレーム同期信号をオーディオ信号の低域に聴覚には認識できないように重畳するために，オーディオ信号の周波数マスキング閾値を算出する必要がある．周波数マスキング閾値の計算方法は指定しないが，概ねオーディオ信号の振幅から 15dB 程度低い振幅で設定される．したがって，オーディオ信号のスペクトル包絡計算で算出したオーディオ信号の周波数情報（振幅情報）に基づいて周波数マスキング閾値を計算する．本規格では，1378Hz に中心周波数を持つフレーム同期信号を重畳するため，オーディオ信号の 1378Hz 付近の振幅情報に基づいて周波数マスキング閾値を算出する．図 7 に示すとおり，シンボル同期信号の振幅はここで算出したマスキング閾値に基づいて決定される．

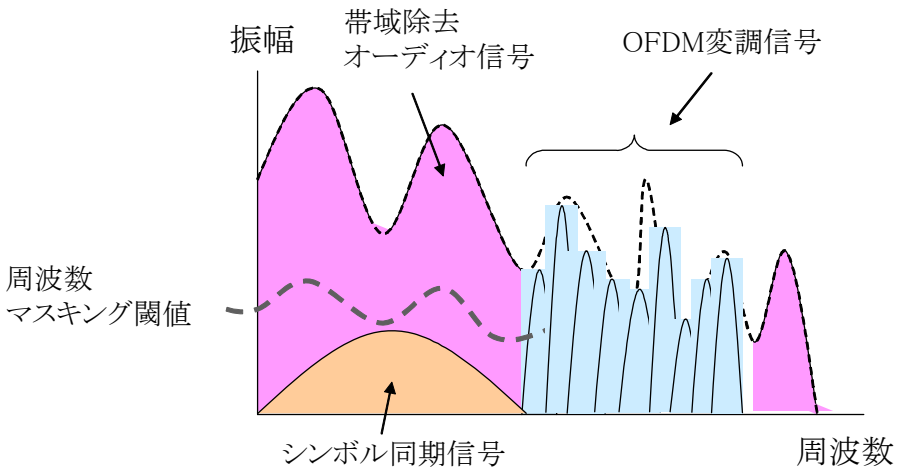

図 7. OFDM フレーム同期信号

5.11. シンボル同期信号生成 / シンボル同期

受信側で OFDM の復調をするためには，OFDM 送信シンボルの境界を検出する必要がある．ガード時間と OFDM 変調信号との相関で検出する方法もあるが，反射などで遅延波があると精度が落ちるため，シンボル同期用の信号を追加する．送信側では，シンボル同期用の符号系列を PSK で位相変調してシンボル同期信号を生成し，低域のオーディオ信号に重畳する．シンボル同期用の符号系列は，表 1 に示すシンボル同期信号生成多項式によって生成される M 系列符号である．受信側では，受信信号の低域信号とシンボル同期信号の相関計算をし，相関値の最も高くなるポイントを OFDM 送信シンボルの先頭とし，OFDM 復調を行う．

このシンボル同期信号は，拡散信号のため，一種の白色ノイズのように聞こえることがある．この雑音感を低減するため，聴覚心理モデルを利用して，音声・音楽の低域信号に周波数マスキングの閾値以下になるようにして重畳することが望まれる．図に音響 OFDM における OFDM フレーム同期信号のスペクトルを示す．まず，音声・音楽がある低周波数

帯域の周波数マスキングの閾値を計算する. 次に M 系列で低周波数域帯に拡散した OFDM フレーム同期信号を周波数マスキングの閾値以下にレベルを調節して重畳する．これにより OFDM フレーム同期信号の音は聴覚には知覚できなくなる.

5.12. 帯域除去フィルタ / 帯域通過フィルタ

エンコーダでは入力されたオーディオ信号に対して，OFDM 変調信号を重畳する周波数帯域の信号を除去する．信号を除去するために，直線位相 FIR の BEF (Band Elimination Filter)を用いる.

デコーダでは，受信したオーディオ信号に対して，シンボル同期用の信号と OFDM 信号を分離するために直線位相 FIR の LPF (Low Pass Filter)を用いる.

5.13. OFDM 復調

OFDM 復調は，帯域通過フィルタで抽出したオーディオ信号に対して，フレーム同期で識別した OFDM フレームの先頭に基づいて OFDM シンボル区間を取り出し，OFDM シンボル区間の時間信号をフーリエ変換して周波数情報を取得する．得られた周波数情報のうち，OFDM のサブキャリア周波数にあたる周波数情報に注目する．誤り訂正復号に硬判定したビット列を用いるのであれば，ひとつ前の OFDM シンボルの同サブキャリアの位相と比較して，位相差が－π/2 から＋π/2 であれば 1，それ以外であれば 0 とする．誤り訂正能力を高くするために軟判定ビタビ復号を用いる場合は，ひとつ前の OFDM シンボルの同サブキャリアの振幅・位相と現シンボルの振幅・位相を用いて受信ビット系列の軟値を算出する．軟値の算出式は，$\mathrm{Re}[z_n \cdot z_{n-1}]$ で計算でき，z_n は時刻 n におけるキャリアの振幅を $(x^2+y^2)^{0.5}$ で，絶対位相を $tan^{-1}(x/y)$.で表せる複素数 $x+jy$ となる．音響 OFDM では，ひとつ前のシンボルと現シンボルの振幅が異なるため，厳密には，ひとつ前のシンボルと現シンボルの位相だけを算出し、その位相差から軟値を計算することが望ましい.

6. 付録：参考ソースコード

6.1. Reed-Solomon 符号

データ分割時における冗長フレーム作成のために，Reed-Solmon 符号を用いる．Reed-Solomon 符号化には下記に示す Vandermonde 行列を用いる．この行列の n 行目は，$(1^n, 2^n, 3^n, 4^n, ...)$ となっている．各分割フレームを先頭から 1 バイトずつ取り出し，冗長フレームも 1 バイトずつ生成していく．下記に 4 つの分割フレームから 4 つの冗長フレームを生成する例を示す．4 つの分割フレームから 1 バイトずつのデータを取り出し，それぞれを(x_1, x_2, x_3, x_4)とする．このベクトルを Vandermonde 行列に掛けて算出されたベクトルは(f_1, f_2, f_3, f_4)となり，それぞれの項が各冗長フレームの 1 バイトのデータとなる．先頭のバイトデータから最終のバイトデータまで 1 バイトずつ同様の処理で符号化をする．

Encoder　※計算は全てGF(256)のガロア体で演算する.

$$\begin{bmatrix} 1^1 & 2^1 & 3^1 & 4^1 \\ 1^2 & 2^2 & 3^2 & 4^2 \\ 1^3 & 2^3 & 3^3 & 4^3 \\ 1^4 & 2^4 & 3^4 & 4^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}$$

$x_1, x_2, x_3, x_4, f_1, f_2, f_3, f_4$ のどの4つのシンボルからでも x_1, x_2, x_3, x_4 を復元できる

Decoder　例えば f_1, f_2, x_2, f_4 から復元

$$\begin{bmatrix} 1^1 & 2^1 & 3^1 & 4^1 \\ 1^2 & 2^2 & 3^2 & 4^2 \\ 0 & 1 & 0 & 0 \\ 1^4 & 2^4 & 3^4 & 4^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ x_2 \\ f_4 \end{bmatrix}$$

左側の行列の逆行列を計算し、
右側のベクトルに掛けると、
x_1, x_2, x_3, x_4 が求まる.

データフレームのヘッダには各々シーケンス番号が付加される．分割フレームの場合は，先頭から何番目のフレームかを示し，冗長フレームの場合は Vandermonde 行列の何行目で演算されて生成されたデータかを示す．上記の例の場合，f_1 のシーケンス番号は 1 に，f_2 のシーケンス番号は 2 になる．受信側における復号化処理では，上記の例に示すとおり，受信したフレームを並べて，それに相当する生成行列を作り，その生成行列の逆行列を計算することで(x_1, x_2, x_3, x_4)を復元できる．計算は全て GF(256)のガロア体で行う．

Code1: ガロア体演算初期設定 r

```
#define RS_GFSIZE (256)
#define RS_GFPRIM (285)

static unsigned char rs_gflog[RS_GFSIZE];
static unsigned char rs_gfinverse[RS_GFSIZE];

void rs_gfinit(){
  unsigned int n, log;

  n = 1;
  for(log = 0; log < (RS_GFSIZE - 1); log++){
    rs_gflog[n] = log;
    rs_gfinverse[log] = n;
    n = n << 1;
    if(n >= RS_GFSIZE){
      n = n ^ RS_GFPRIM;
    }
  }
}
```

Code1 は，リードソロモン符号の演算に用いるガロア体の初期化を行う．ガロア体は GF(256)であり，乗算，除算のためのテーブルを作成する．GF(256)は GF(2)の拡大体であり，生成多項式 $x^8+x^4+x^3+x^2+1$ を用いて生成される．後述する BCH 符号に用いるガロア体とは異なる．

Code2: ガロア体乗算

```
unsigned char rs_gfmult(a, b)
     unsigned char a, b;
{
  unsigned int sum;

  if(a == 0 || b == 0){
    return 0;
  }
  sum = rs_gflog[a] + rs_gflog[b];
  if(sum >= RS_GFSIZE-1){
    sum -= (RS_GFSIZE - 1);
  }
  return rs_gfinverse[sum];
}
```

Code2 はリードソロモン符号の演算に用いるガロア体の乗算 rs_gfmult である．a*b の演算結果を返す.

Code3: ガロア体除算

```
unsigned char rs_gfdiv(a, b)
     unsigned char a, b;
{
  int sum;

  if(a == 0) return 0;
  if(b == 0) return -1;
  sum = rs_gflog[a] - rs_gflog[b];
  if(sum < 0) sum += (RS_GFSIZE - 1);
  return rs_gfinverse[sum];
}
```

Code3 はリードソロモン符号の演算に用いるガロア体の除算 rs_gfdiv である．a/b の演算結果を返す.

Code4: ガロア体べき乗

```
unsigned char rs_gfpower(a, b)
     unsigned char a, b;
{
  int i;
  unsigned char sum;

  sum = 1;
  for(i=0; i < b; i++){
    sum = rs_gfmult(sum, a);
  }
  return sum;
}
```

Code4 はリードソロモン符号の演算に用いるガロア体のべき乗 rs_gfpower である．a^b の演算結果を返す.

Code5: リードソロモン符号化

```
void rs_encode(data, fec, ndata, nfec)
     unsigned char *data, *fec;
     unsigned int ndata, nfec;
{
  int i, j, k;

  bzero(fec, (nfec * sizeof(unsigned char)));
  for(i = 0; i < ndata; i++){
    for(j = 0; j < nfec; j++){
      fec[j] ^= rs_gfmult(data[i], rs_gfpower((i+1), (j+1)));
    }
  }
}
```

Code5 はリードソロモン符号の符号化処理 rs_encode である．データの配列 data，生成する冗長データの配列 fec，データ数 ndata，冗長データ数 nfec を引数とする．

Code6: リードソロモン符号化

```
void rs_decode(d_data, d_seq, f_data, f_seq, r_data, r_seq, ndata, nfec)
     unsigned char *d_data, *f_data, *r_data;
     unsigned char *d_seq, *f_seq, *r_seq;
     int ndata, nfec;
{
  int i, j, k;
  unsigned char **matrix;
  unsigned char multiple;
  int total = ndata + nfec;

  matrix = (unsigned char **)malloc(nfec * sizeof(unsigned char *));
  for(i = 0; i < nfec; i++){
    matrix[i] = (unsigned char *)malloc(nfec * sizeof(unsigned char));
  }

  for(i = 0; i < ndata; i++){
    for(j = 0; j < nfec; j++){
      f_data[j] ^= rs_gfmult(d_data[i], rs_gfpower(d_seq[i], f_seq[j]));
    }
  }

  for(i = 0; i < nfec; i++){
    for(j = 0; j < nfec; j++){
      matrix[j][i] = rs_gfpower(r_seq[i], f_seq[j]);
    }
  }

  for(i = 0; i < nfec; i++){
    multiple = matrix[i][i];
    f_data[i] = rs_gfdiv(f_data[i], multiple);
    for(j = 0; j < nfec; j++){
      matrix[i][j] = rs_gfdiv(matrix[i][j], multiple);
    }
```

```
    for(j = 0; j < nfec; j++){
      if(j != i){
        multiple = matrix[j][i];
        f_data[j] ^= rs_gfmult(f_data[i], multiple);
        for(k = 0; k < nfec; k++){
          matrix[j][k] ^= rs_gfmult(matrix[i][k], multiple);
        }
      }
    }
  }

  for(i = 0; i < nfec; i++){
    r_data[i] = f_data[i];
  }

}
```

Code6 は，リードソロモン符号の復号化処理 rs_decode である．復号に用いるデータの配列 d_data，対応するデータのシーケンス番号の配列 d_seq，復号に用いる冗長データの配列 f_data，対応する冗長データのシーケンス番号の配列 f_seq，消失したデータの復元用データ配列 r_data，対応する消失データのシーケンス番号の配列 r_data，復号に用いるデータの数 ndata，復号に用いる冗長データの数 nfec を引数とする．

6.2. CRC、畳み込み符号、軟判定ビタビ復号、インタリーブ

音響 OFDM では通信路符号化に，CRC と畳込み符号を用いる．ここでは，CRC のエンコーダ・デコーダおよび畳込み符号，軟判定ビタビ復号のプログラム例を記載する．

Code7: CRC Encoder

```
#define CRCbit                  (7)             // Number of CRC Bit
#define INFO_LENGTH         (360)               // Number of Data Bit
int crcg[CRCbit + 1] = {1,0,0,0,0,0,1,1};       // CRC Generator Polynomial

void crc_encode(char *info, char *data)
{
    int i, j;
    char crc[INFO_LENGTH + CRCbit];
    bzero(crc, INFO_LENGTH + CRCbit);

     for(i = 0; i < INFO_LENGTH; i++){
          data[i] = info[i];
          crc[i] = info[i];
    }

    for(i = 0; i < INFO_LENGTH; i++)
        if(crc[i])
            for(j = 0; j < CRCbit + 1; j++)
                crc[i+j] ^= crcg[j];

    for(i = INFO_LENGTH; i < INFO_LENGTH + CRCbit; i++)
        data[i] = crc[i];

}
```

Code7 は, CRC エンコーダの関数 crc_encode であり, 入力の引数 info のビット列から CRC チェックサムを計算し，CRC のチェックサムを info のビット列の後方に付け足したものを引数 data に格納する．

Code8: CRC Decoder

```
#define CRCbit                  (7)             // Number of CRC Bit
#define INFO_LENGTH         (360)               // Number of Data Bit
int crcg[CRCbit + 1] = {1,0,0,0,0,0,1,1};       // CRC Generator Polynomial

int crc_decode(char *data){
      int i, j;
      char info[INFO_LENGTH], crc[INFO_LENGTH + CRCbit];

      for(i = 0; i < INFO_LENGTH; i++){
          info[i] = data[i];   crc[i] = data[i];
     }

     for(i = 0; i < INFO_LENGTH; i++)
          if(crc[i])
              for(j = 0; j < CRCbit + 1; j++)
                  crc[i+j] ^= crcg[j];

     for(i = INFO_LENGTH; i < INFO_LENGTH + CRCbit; i++)
           if(crc[i] ^ data[i])
              return(0);

      return(1);

}
```

Code8 は，CRC デコーダの関数 crc_decode であり，入力の引数 data のビット列に対して CRC チェックを行い，CRC が正しければ 1 を返し，CRC が間違っていれば 0 を返す.

Code9: 畳込み符号化

```
#define Nregister              (5)
#define Nrate                  (3)
#define DATA_LENGTH    (372)

void conv_encode(char *data, char *code)
{
    int i, j, k;
    int registers[Nregister + 1];
    int exclusive[Nrate];

    for(i = 0; i < (Nregister + 1); i++)
        registers[i] = 0;

    for(i = 0; i < DATA_LENGTH; i++){
        for(j = Nregister; j > 0; j--)
            registers[j] = registers[j-1];
        registers[0] = data[i];

        for(j = 0; j < Nrate; j++)
            exclusive[j] = 0;

        for(j = 0; j < (Nregister + 1); j++)
            for(k = 0; k < Nrate; k++)
                exclusive[k] ^= registers[j] & g[k][j];

       for(j = 0; j < Nrate; j++)
            code[(i * Nrate) + j]     = exclusive[j];

    }
}
```

Code9 は畳込み符号化の関数 conv_encode であり，引数の data によるビット列を畳み込み符号化して符号化出力のビット列を引数の code に格納する．

Code10: 軟判定ビタビ復号

```
#define Nregister              (5)
#define Nrate                  (3)
#define Nstate                 (32)
#define DATA_LENGTH    (372)
int g[Nrate][Nregister + 1] = { {1,0,0,1,1,1}, {1,0,1,0,1,1}, {1,1,1,1,0,1} };

double branch_metric_soft(int state, int branch, double *data)
{
    int i, j;
    double metric;
    int exclusive[Nrate];

    for(i = 0; i < Nrate; i++)
        exclusive[i] = 0;

    for(i = 0; i < Nregister; i++)
        for(j = 0; j < Nrate; j++)
            exclusive[j] ^= ((state >> i) & 0x01) & g[j][(Nregister - i)];

    for(i = 0; i < Nrate; i++)
        exclusiver[i] ^= branch & g[i][0];

    metric = 0.0;
    for(i = 0; i < Nrate; i++){
        if(exclusive[i]){
            metric += data[i];
        }else{
            metric -= data[i];
        }
    }
    return metric;
}
```

```
void viterbi_decode_soft(double *recv, char *recover)
{
    int i, state, pstate, branch, pbranch;
    double metric[Nstate][2], path_metric[Nstate], next_path_metric[Nstate];
    int path_memory[DATA_LENGTH][Nstate];
    double binary[2];
    int nerror;

    path_metric[0] = 0;
    for(i = 1; i < Nstate; i++)
        path_metric[i] = (-100) * DATA_LENGTH;;

   for(i = 0; i < DATA_LENGTH; i++){
        for(state = 0; state < Nstate; state++)
            for(branch = 0; branch < 2; branch++)
                metric[state][branch]
                        = branch_metric_soft(state, branch, &recv[(i*Nrate)]);

        for(state = 0; state < Nstate; state++){
            pbranch = (state >> (Nregister - 1)) & 0x01;
            pstate = (state << 1) & 0x1f;
            binary[0] = path_metric[pstate] + metric[pstate][pbranch];
            pstate = pstate + 1;
            binary[1] = path_metric[pstate] + metric[pstate][pbranch];

            if(binary[0] > binary[1]){
                path_memory[i][state] = 0;
                next_path_metric[state] = binary[0];
            }else{
                path_memory[i][state] = 1;
                next_path_metric[state] = binary[1];
            }
        }
```

```
        for(state = 0; state < Nstate; state++)
            path_metric[state] = next_path_metric[state];

    }

    /* Trace Back */
    state = 0;
    for(i = (DATA_LENGTH - 1); i >= 0; i--){
        recover[i] = (state >> (Nregister - 1)) & 0x01;
        if(path_memory[i][state] == 0){
            state = (state << 1) & 0x1f;
        }else{
            state = ((state << 1) & 0x1f) + 1;
        }
    }

}
```

Code10 は軟判定ビタビ復号の関数 viterbi_decode_soft とそれに用いるブランチメトリックを計算する関数 branch_metric_soft である．viterbi_decode_soft は，軟値のビット列である recv を第 1 引数とし，軟判定ビタビ復号をして誤りビットを訂正したビット列を recover に格納する．

Code11: インタリーブ

```
#define CODE_LENGTH (1116)

void code_interleave(char *code)
{
    int i, m, n;
    char tmp1[CODE_LENGTH];
    char tmp2[CODE_LENGTH];
```

```
    for(i = 0; i < CODE_LENGTH; i++){
        if(i % 2){
            tmp1[CODE_LENGTH - 1 - (i / 2)] = code[i];
        }else{
            tmp1[i / 2] = code[i];
        }
    }

    m = 0; n = 0;
    for(i = 0; i < CODE_LENGTH; i++){
        tmp2[m+n] = tmp1[i];
        m += Ninterleave;
        if(m >= CODE_LENGTH){
            m = 0;    n++;
        }
    }

    for(i = 0; i < CODE_LENGTH; i++)
        code[i] = tmp2[i];

}
```

Code11 は符号化ビット列をインタリーブする関数 code_interleave であり，引数のビット列 code に対して，本稿で記載したインタリーブ 1 とインタリーブ 2 を順番に行い，インタリーブ後のビット列を code に格納する．

Code12: デインタリーブ

```
#define CODE_LENGTH (1116)

void code_deinterleave(double *code)
{
    int i, m, n;
    double tmp1[CODE_LENGTH];
    double tmp2[CODE_LENGTH];
```

```
    m = 0; n = 0;
    for(i = 0; i < CODE_LENGTH; i++){
        tmp1[i] = code[m+n];
        m += Ninterleave;
        if(m >= CODE_LENGTH){
            m = 0;    n++;
        }
    }

    for(i = 0; i < CODE_LENGTH; i++){
        if(i % 2){
            tmp2[i] = tmp1[CODE_LENGTH - 1 - (i / 2)];
        }else{
            tmp2[i] = tmp1[i / 2];
        }
    }

    for(i = 0; i < CODE_LENGTH; i++)
        code[i] = tmp2[i];

}
```

Code12 は，受信した軟値系列をデインタリーブする関数 code_deinterleave であり，引数である軟値の系列 code に対して，本稿で記載したインタリーブ方法の逆変換をインタリーブ 2，インタリーブ 1 の順番で行い，変換後の系列を code に格納する．

6.3. シンボル同期処理

音響 OFDM では OFDM シンボルの同期に，PN 系列による同期信号を生成する．ここでは，シンボル同期信号の生成および同期処理のプログラム例を記載する．

Code13: OFDM シンボル同期信号生成

```
#define Fsmp                (2032)
#define CYCLE               (63)
#define CHIPWIDTH           (32)
#define GENERATOR           (11)
#define MAXBIT              (0x0040)
#define XORBIT              (0x0043)

char * pngen()
{
        int i;
        unsigned int shift = GENERATOR;
        char *pn;
         pn = (char *)calloc(CYCLE, sizeof(char));

        for(i = 0; i < CYCLE; i++){
            if(shift & 0x01){
                pn[i] = 1;
            }else{
                pn[i] = 0;
            }
            shift = shift << 1;
            if(shift & MAXBIT){
                shift = shift ^ XORBIT;
            }
        }
        return(pn);
}
```

```
void syncgen(char *xn, char *pn)
{
        int    i, j;
        double pskw =    (2 * M_PI) / CHIPWIDTH;
        bzero(xn, (sizeof(double) * Fsmp));

         for(i = 0; i < CYCLE; i++){
              if(pn[i]){
                   for(j = 0; j < CHIPWIDTH; j++)
                        xn[(i * CHIPWIDTH) + j] = sin(pskw * j);
              }else{
                   for(j = 0; j < CHIPWIDTH; j++)
                        xn[(i * CHIPWIDTH) + j] = (-1) * sin(pskw * j);
              }
        }
}
```

Code13 は OFDM フレーム同期信号生成用のビット系列生成関数 pngen と，信号生成関数 syncgen である．pngen は信号生成に用いるビット系列を生成して戻り値として返す．ビット系列の生成には，生成多項式 x^6+x+1 の M 系列符号を用いる． syncgen は，pngen で生成したビット系列を引数 pn として，pn から 1 サンプルあたりの角度変移 pskw の正弦波を用いて OFDM フレーム同期信号を生成する．生成した信号を引数 xn に格納する．

Code 14: OFDM シンボル同期処理

```
#define Fsmp                (2032)
#define CYCLE                (63)
#define CHIPWIDTH        (32)

int symbol_sync(short *x,    char *pn)
{
        int i, j, k, n, tw;
        int sum[Fsmp], val, max;

        for(i = 0; i < Fsmp; i++)
            sum[i] = 0;
```

```
    for(i = 0; i < (Nframe + 1); i++){
        for(j = 0; j < Fsmp; j++){
            n = (i * Fsmp) + j;
            for(k = 0; k < CYCLE; k++){
                if(pn[k]){
                    sum[j] += x[n];
                }else{
                    sum[j] -= x[n];
                }
                n += CHIPWIDTH;
            }
        }
    }

    n = 0;    max = 0;
    tw = CHIPWIDTH / 2;
    for(i = 0; i < Fsmp - tw; i++){
        val = sum[i] - sum[(i + tw)];
        if(val > max){
            max = val;    n = i;
        }
    }
    for(i = Fsmp - tw; i < Fsmp; i++){
        val = sum[i] - sum[(i + tw - Fsmp)];
        if(val > max){
            max = val;    n = i;
        }
    }
    n = (n + Fsmp - (tw / 2)) % Fsmp;
    return n;
}
```

Code14 は，OFDM シンボル同期をとる関数 symbol_sync であり，入力信号系列 x と PN 系列 pn を引数として，x と pn の相関をとり，相関値が最も高くなるサンプル点を先頭からのオフセットとして戻り値で返す.

6.4. データ同期処理

音響 OFDM ではデータフレームの同期に，BCH 符号による巡回符号系列を用いる．巡回符号系列はシングルフレーム伝送用とマルチフレーム伝送用に以下の 2 種類の符号系列を用いる．

【シングルフレーム伝送用】

0,0,0,0,0,1,1,1,0,0,1,0,0,0,1,0,1,0,1,1,1,1,0,1,1,0,1,0,0,1,1

【マルチフレーム伝送用】

0,0,0,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,1,0,0,0,1,1,0,1,1,1,0,1,0,1

受信側では符号系列の種類でシングルフレーム伝送かマルチフレーム伝送かを判別し，巡回ビット数でデータフレームの同期を行う．ここでは，伝送された巡回符号系列の誤り訂正を行う BCH 復号化処理のプログラム例を記載する．

Code15: ガロア体初期化（BCH 符号）

```
#define BCH_CODELEN     (31)    /* code length */
#define BCH_INFOLEN      (6)    /* information length within a code */
#define DEGREE          (25)    /* degree of generator polynomial */
#define DIST            (15)    /* code distance */
#define Nerror           (7)    /* number of lost bits to be recovered */
#define Ncode            (2)    /* number of lost bits to be recovered */
#define GFSIZE          (32)
#define GFPRIM          (37)

unsigned char gflog[GFSIZE];
unsigned char gfexp[GFSIZE];

/*
 *  Generator Polynomial
 *  n = 31,  k = 6,  d = 15 (t = 7)
 */
unsigned char gen_poly[BCH_CODELEN] =
{1,1,1,0,0,1,0,0,0,1,0,1,0,1,1,1,1,0,1,1,0,1,0,0,1,1,0,0,0,0,0};
```

```
void gf_init()
{
  unsigned int n, log;

  n = 1;
  for(log = 0; log < (GFSIZE - 1); log++){
    gflog[n] = log;
    gfexp[log] = n;
    n = n << 1;
    if(n >= GFSIZE){
      n = n ^ GFPRIM;
    }
  }
}
```

Code15 は，リードソロモン符号の演算に用いるガロア体の初期化を行う．ガロア体は GF(32)であり，乗算，除算のためのテーブルを作成する．GF(32)は GF(2)の拡大体であり，生成多項式 x^5+x^2+1 を用いて生成される．

Code16: ガロア体乗算（BCH 符号）

```
unsigned char gfmult(a, b)
     unsigned char a, b;
{
  unsigned int sum;

  if(a == 0 || b == 0){
    return 0;
  }
  sum = gflog[a] + gflog[b];
  if(sum >= (GFSIZE - 1)){
    sum -= (GFSIZE - 1);
  }
  return gfexp[sum];
}
```

Code16はBCH符号の演算に用いるガロア体の乗算gfmultであり，a*bの演算結果を返す．

Code17: ガロア体除算（BCH 符号）

```
unsigned char gfdiv(a, b)
     unsigned char a, b;
{
  int sum;

  if(a == 0) return 0;
  if(b == 0) return -1;
  sum = gflog[a] - gflog[b];
  if(sum < 0) sum += (GFSIZE - 1);
  return gfexp[sum];
}
```

Code17 は BCH 符号の演算に用いるガロア体の除算 gfdiv であり，a/b の演算結果を返す.

Code18: ガロア体べき乗（BCH 符号）

```
unsigned char gfpower(a, b)
     unsigned char a, b;
{
  int i;
  unsigned char sum;

  sum = 1;
  for(i=0; i < b; i++){
    sum = gfmult(sum, a);
  }
  return sum;
}
```

Code18 は BCH 符号の演算に用いるガロア体のべき乗 gfpower であり，a^b の演算結果を返す.

Code19: BCH 復号化

```
int bch_decode(a)
     unsigned char *a;
{
  int i, j, k, n;
  int nerror = Nerror;
  int complete;
  unsigned char s[DIST];                  /* syndrome vector */
  unsigned char elp[Nerror];              /* error locator polynomial */
  unsigned char errorpos[Nerror];         /* error position */
  unsigned char matrix[Nerror][Nerror];   /* decoding matrix */
  unsigned char inverse[Nerror][Nerror];  /* inverse matrix */
  unsigned char val, check;

  /* calculating syndromes */
  complete = 1;
  for(i = 0; i < DIST; i++){
    s[i] = 0;
    for(j = 0; j < BCH_CODELEN; j++){
      if(a[j]){
        s[i] ^= gfexp[(i * j) % (GFSIZE - 1)];
      }
    }
    if(s[i] != 0){
      complete = 0;
    }
  }

  if(complete){
    return(0);
  }
```

```
while(complete == 0){
  complete = 1;
  /* calculating decoding matrix */
  for(i = 0; i < nerror; i++){
    for(j = 0; j < nerror; j++){
      matrix[i][j] = s[(nerror + i) - j];
    }
  }
  /* inverting matrix (gaussian elimination) */
  for(i = 0; i < nerror; i++){
    for(j = 0; j < nerror; j++){
      if(i == j){
        inverse[i][j] = 1;
      }else{
        inverse[i][j] = 0;
      }
    }
  }
  for(i = 0; i < nerror; i++){
    n = matrix[i][i];
    if(n == 0){
      complete = 0;
      nerror--;
      break;
    }
    for(j = 0; j < nerror; j++){
      matrix[i][j] = gfdiv(matrix[i][j], n);
      inverse[i][j] = gfdiv(inverse[i][j], n);
    }
```

```
      for(k = 0; k < nerror; k++){
        if(k == i){
          continue;
        }
        n = matrix[k][i];
        for(j = 0; j < nerror; j++){
          matrix[k][j]  ^= gfmult(matrix[i][j], n);
          inverse[k][j] ^= gfmult(inverse[i][j], n);
        }
      }
    }
  }
  /* matrix multiplication (inverse * syndromes) */
  for(i = 0; i < nerror; i++){
    elp[i] = 0;
    for(j = 0; j < nerror; j++){
      elp[i] ^= gfmult(inverse[i][j], s[(nerror + 1) + j]);
    }
  }
  /* determine error position */
  n = 0;
  for(i = 0; i < BCH_CODELEN; i++){
    val = 1;
    for(j = 0; j < nerror; j++){
      val ^= gfmult(elp[j], gfexp[(i * (j+1)) % (GFSIZE - 1)]);
    }
    if(val == 0){
      if(i == 0){
        errorpos[n] = 0;
      }else{
        errorpos[n] = BCH_CODELEN - i;
      }
      n++;
    }
  }
```

```
  /* error check */
  check = BCH_CODELEN;
  for(i = 0; i < nerror; i++){
    if(errorpos[i] >= check){
      return -1;
    }
    if((i!=0) || (errorpos[i] != 0)){
      check = errorpos[i];
    }
  }

  /* recover error */
  for(i = 0; i < nerror; i++){
    if(errorpos[i] < BCH_CODELEN){
      a[errorpos[i]] ^= 0x01;
    }
  }

  return(nerror);

}
```

Code19 は，BCH 符号の符号化処理であり，ビット列 a を BCH 復号し，誤りのあるビットを訂正する．訂正したビット数を戻り値として返す.

6.5. デコーダサンプルソースコード

上記で記載の関数を利用したデコーダのサンプルソースコードについて記載する。まず、Code20 にデコーダのサンプルソースコードで必要となるパラメータを示す.

Code20: パラメータ

```
#define SAMPLERATE   (44100)
#define CHANNEL        (1)
#define Nsmp         (1024)
#define Nspc          (512)
#define Ntap          (127)
#define Mtap           (63)

#define Nband          (36) // Number of OFDM Carriers
#define Nframe         (31) // Number of Symbols in a data frame
#define Gsmp          (800) // Guard Interval
#define Ssmp          (208) // Smoothing Interval
#define Fsmp         (2032)
#define BEF_LOW      (7200)
#define BEF_HIGH     (8500)
#define PNBand     (5512.5)
#define FShift       (6400)
#define Amplify       (1.0)
#define DATA_SIZE    (45) // Number of Data byte
#define MaxFrame     (14) // Maximum Number of Data and Fec Frame
#define MaxData       (7) // Maximum Number of Data
#define MaxFec        (7) // Maximum Number of FEC
#define MDF_PSIZE    (44) // Payload Size in Data Frame

struct mdf_header{
  unsigned int reserved:1;
  unsigned int fec:1;
  unsigned int nframe:3;
  unsigned int seq:3;
};
```

Code21 には、デコーダのサンプルソースコードのメイン関数を記載する.

Code21; メイン関数

```
#define min(a, b) ((a) > (b) ? (b) : (a))
#define max(a, b) ((a) > (b) ? (a) : (b))

int main(argc, argv)
     int argc;
     char **argv;
{
  int i, j, k, m, n, op, in, out;
  short *p, sig;
  char *input, *readbuf;

  /* for Signal Processing */
  int sync, bpsk[Nband];
  double maxf, valf;
  double h[Ntap], z[Ntap], x[Fsmp], yl[Fsmp], yh[Fsmp];
  double dt[Nsmp], pw[Nspc], re[Nsmp], im[Nsmp];
  double re_s, im_s, re_p[Nband], im_p[Nband];

  /* for PN and SYNC code */
  int codenumber, codeshift, coderemain, datashift, codein = 0;
  char *pn, scode[Nframe];
  double rbuf[Fsmp], rsum[Fsmp];

  /* for Convolutional code */

  unsigned char info[INFO_LENGTH];
  unsigned char data[DATA_LENGTH];
  double code[CODE_LENGTH];
  double code_next[CODE_LENGTH];
  double recv[CODE_LENGTH];
  unsigned int codepos = 0;
  int nerror;
```

```
/* Multi Frame Message */
char message[DATA_SIZE];
char multi_message[MaxData][(DATA_SIZE - 1)];
char d_message[MaxData][MDF_PSIZE];
char f_message[MaxData][MDF_PSIZE];
char r_message[MaxData][MDF_PSIZE];
struct mdf_header *mdf;
unsigned char d_seq[MaxData], f_seq[MaxData], r_seq[MaxData];
unsigned char t_data[MaxData], t_fec[MaxData], t_rec[MaxData];
unsigned int recover, ndata = 0, nfec = 0;
int seek_current = 0;

if(argc == 2){
  input = argv[0];
}

/* Parameter Initialization */
fft_init();
gf_init();
rs_gfinit();
lpf_init(PNBand, h);
pn = (char *)pngen();
bzero(rsum, sizeof(double) * Fsmp);

n = (FShift * Nsmp) / SAMPLERATE;
sync = n; n++;
for(i = 0; i < Nband; i++){
  bpsk[i] = n; n++;
}

for(i = 0; i < MaxData; i++){
  d_seq[i] = 0;
  f_seq[i] = 0;
  r_seq[i] = 0;
}
```

```
  /* File Open */
  in = audio_file_open(input);
  out = temporal_file_open("tmpfile");

  if(wave_hdr_check(in) < 0){
    printf("Not Supported Wave File¥n");
  }

  lseek(in, offset, SEEK_CUR);
  readbuf = (char *)malloc(Fsmp * 2);

  /* Filter Preparation */
  n = read(in, (char *)readbuf, (Ntap * 2));
  p = (short *)readbuf;
  for(i = 0; i < Ntap; i++){
    z[(Ntap - 1) - i] = *p;
    p++;
  }

restart:
  lseek(out, 0, SEEK_SET);
  codepos = 0;
  n = read(in, (char *)readbuf, (Fsmp * 2));
  if(n < (Fsmp * 2)){
    exit(1);
  }
  p = (short *)readbuf;

  /* Separating Low Band and High Band */
  for(j = 0; j < Fsmp; j++){
    yl[j] = 0.0;
    yh[j] = z[Mtap];
    for(k = 0; k < Ntap; k++){
      yl[j] += (h[k] * z[k]);
    }
```

```
    for(k = 0; k < (Ntap - 1); k++){
      z[(Ntap - 1) - k] = z[(Ntap - 2) - k];
    }
    z[0] = *p; p++;
  }
  for(j = 0; j < Fsmp; j++){
    yh[j] -= yl[j];
    n = write_data(out, &yh[j], sizeof(double));
    rbuf[j] = yl[j];
  }

again:
  if(codein){
    lseek(out, 0, SEEK_END);
    codepos = 0;
  }

  /* Decoding PN Signal */
  for(i = 0; i < (Nframe + 1); i++){

    /* Read Data */
    n = read_audio(in, (char *)readbuf, (Fsmp * sizeof(short)));
    if(n < (Fsmp * sizeof(short))){
      break;
    }
    p = (short *)readbuf;

    /* Separating Low Band and High Band */
    for(j = 0; j < Fsmp; j++){
      yl[j] = 0.0;
      yh[j] = z[Mtap];
      for(k = 0; k < Ntap; k++){
        yl[j] += (h[k] * z[k]);
      }
```

```
      for(k = 0; k < (Ntap - 1); k++){
        z[(Ntap - 1) - k] = z[(Ntap - 2) - k];
      }
      z[0] = *p; p++;
    }
    for(j = 0; j < Fsmp; j++){
      yh[j] -= yl[j];
    }

    for(j = 0; j < Fsmp; j++){
      rsum[j] += peak(rbuf, j, pn);
      rbuf[j] = yl[j];
      n = write_data(out, &yh[j], sizeof(double));
    }

  }

  /* Frame Head Adjustment */
  maxf = 0.0; n = 0;
  for(i = 0; i < Fsmp; i++){
    valf = rsum[i] - rsum[(i + (CHIPWIDTH / 2)) % Fsmp];
    if(valf > maxf){
      maxf = valf;
      n = i;
    }
  }
  n = (n + Fsmp - (CHIPWIDTH / 4)) % Fsmp;

  if(codein){
    lseek(out, seek_current, SEEK_SET);
  }else{
    lseek(out, n * sizeof(double), SEEK_SET);
    seek_current = n * sizeof(double);
  }
```

```
/* Decoding Reference Phase */
if(!codein){
  for(j = 0; j < Fsmp; j++){
    n = read_data(out, &x[j], sizeof(double));
  }
  seek_current += (Fsmp * sizeof(double));

  for(j = 0; j < Nsmp; j++){
    dt[j] = x[Gsmp + j];
  }
  fft(dt, pw, re, im, 0);

  re_s = re[sync];
  im_s = im[sync];
  for(j = 0; j < Nband; j++){
    re_p[j] = re[bpsk[j]];
    im_p[j] = im[bpsk[j]];
  }

}

/* Decoding Data Signal */
for(i = 0; i < Nframe; i++){

  for(j = 0; j < Fsmp; j++){
    n = read_data(out, &x[j], sizeof(double));
  }
  seek_current += (Fsmp * sizeof(double));

  for(j = 0; j < Nsmp; j++){
    dt[j] = x[Gsmp + j];
  }

  fft(dt, pw, re, im, 0);
```

```
    valf = (re_s * re[sync]) + (im_s * im[sync]);
    re_s = re[sync];
    im_s = im[sync];

    if(valf > 0){
      scode[i] = 1;
    }else{
      scode[i] = 0;
    }

    for(j = 0; j < Nband; j++){
      valf = (re_p[j] * re[bpsk[j]]) + (im_p[j] * im[bpsk[j]]);
      re_p[j] = re[bpsk[j]];
      im_p[j] = im[bpsk[j]];
      /* Soft Decision */
      if(valf > 0){
        recv[codepos + j] = max(MIN_SCALE, log10(valf));
      }else{
        recv[codepos + j] = min(-MIN_SCALE, -log10(-valf));
      }
    }
    codepos += Nband;
  }
  nerror = bch_decode(scode);
  if(nerror < 0){
    if(codein){
      for(i = 0; i < coderemain; i++){
        code_next[i] = recv[(codeshift * Nband) + i];
      }
    }
    if(!codein){
      goto restart;
    }
    goto again;
  }
```

```
codeshift = bch_polyhead(scode);
codenumber = bch_codenumber(scode);

if(codenumber == 0){
  /* single frame */
  for(i = 0; i < CODE_LENGTH; i++){
    code[i] = recv[((codeshift * Nband) + i) % CODE_LENGTH];
  }
  code_deinterleave_soft(code);

  /* Convolutional Decode */
  viterbi_decoder_soft(code, data);

  if(info_decoder(data, info) == 0){
    printf("¥nDecode Error¥n¥n");
    exit(1);
  }

  poly2data(info, message, INFO_LENGTH);
  printf("¥n¥n%s¥n¥n", message);

}else if(codenumber == 1){

  /* multi frame */
  coderemain = CODE_LENGTH - (codeshift * Nband);

  if(codein){

    for(i = 0; i < coderemain; i++){
      code[i] = code_next[i];
    }
    for(i = 0; i < (codeshift * Nband); i++){
      code[coderemain + i] = recv[i];
    }
```

```
      code_deinterleave_soft(code);
      /* Convolutional Decode */
      viterbi_decoder_soft(code, data);

      if(info_decoder(data, info) == 0){
        printf(" Decode Error");
        for(i = 0; i < coderemain; i++){
          code_next[i] = recv[(codeshift * Nband) + i];
        }
        goto again;
      }else{
        bzero(message, DATA_SIZE);
        poly2data(info, message, INFO_LENGTH);
        mdf = (struct mdf_header *)message;
        if(mdf->fec){
          bcopy(&message[1], f_message[nfec], MDF_PSIZE);
          f_seq[nfec] = mdf->seq;
          nfec++;
        }else{
          bcopy(&message[1], d_message[ndata], MDF_PSIZE);
          d_seq[ndata] = mdf->seq;
          ndata++;
          bcopy(&message[1], multi_message[(mdf->seq) - 1], MDF_PSIZE);
        }
      }
    }
     for(i = 0; i < coderemain; i++){
      code_next[i] = recv[(codeshift * Nband) + i];
     }
     if(!codein){
      codein = 1;
      goto again;
     }
     if(ndata + nfec < mdf->nframe){
      goto again;
     }
```

```
    /* FEC Decoding */
    recover = 1;
    for(i = 0; i < nfec; i++){
      for(j = 0; j < ndata; j++){
        if(d_seq[j] == recover){
          recover++;
          j = 0;
        }
      }
      r_seq[i] = recover;
      bzero(r_message[i], MDF_PSIZE);
      recover++;
    }
    for(i = 0; i < MDF_PSIZE; i++){
      for(j = 0; j < ndata; j++){
        t_data[j] = d_message[j][i];
      }
      for(j = 0; j < nfec; j++){
        t_fec[j] = f_message[j][i];
      }
      rs_decode(t_data, d_seq, t_fec, f_seq, t_rec, r_seq, ndata, nfec);
      for(j = 0; j < nfec; j++){
        r_message[j][i] = t_rec[j];
      }
    }
    for(i = 0; i < nfec; i++){
      bcopy(r_message[i], multi_message[r_seq[i] - 1], MDF_PSIZE);
    }
    printf("¥n¥n%s¥n¥n", multi_message[0]);
  }
  unlink("tmpfile");
}
```

```
/* Peak Calculation */
double peak(rbuf, num, pn)
     double *rbuf;
     int num;
     char *pn;
{
  int i, n;
  double sum = 0.0;
  n = num;

  for(i = 0; i < CYCLE; i++){
    if(pn[i]){
      sum += rbuf[n];
    }else{
      sum -= rbuf[n];
    }
    n += CHIPWIDTH;
    if(n >= Fsmp){
      n -= Fsmp;
    }
  }
  return(sum);
}
```

このサンプルソースコードは、UNIX系のOSで動作するように作られている。そのため、UNIX系特有の関数であるlseekやbzero等の関数を利用している．また、メイン関数の中にあるaudio_file_open()関数は、引数で指定されたファイル名のWAVEファイルをオープンする関数であり、temporal_file_open()は、引数で指定されたファイルをデコード処理用一時ファイルとして書き込み可能なファイルをオープンする．これらは使用するUNIX系OSのAPIに合わせて別途作る必要がある。また、fft(dt, pw, re, im, 0)関数は、dtで指定された信号値の配列をフーリエ変換し、そのパワー値をpwに、実数値をreに、虚数値をimに格納する関数である．これも、使用するUNIX系OSのAPIに合わせて別途作る必要がある。