WSJT 系新モード「FST4/FST4W」のプロトコルを読み解く

JP7VTF

1. WSJT 系新モード「FST4/FST4W」

近年、FT8 をはじめとする WSJT 系ディジタルモードが人気を博していますが、最近リリースされた WSJT-X v2.3.0 から新モード「FST4/FST4W」が導入されました[1]。FST4/FST4W は特に LF 帯(アマチュアバンドでは 135 kHz 帯)および MF 帯(475 kHz 帯)用に設計されたディジタルプロトコルです。
FST4 は FT8 のような双方向 QSO 用のモードで、FST4W は WSPR のような準ビーコン送信用のモードです。LF・MF 帯で双方向 QSO 用として用いられている JT9 や準ビーコン送信用モードとして用いられている WSPR と比較すると、FST4/FST4W では S/N しきい値[1]が向上しています[2,3]。また、参考文献[3]では JT9、WSPR のユーザは FST4/FST4W に移行するように勧告しています。JT9 や WSPR から FST4/FST4W への改良点としては以下が挙げられます。

- JT9 や WSPR では畳み込み符号(拘束長 K=32、符号化率 r=1/2)を用いていたが、FST4/FST4W では Low density parity check(低密度パリティ検査符号、LDPC、FST4 では(n,k)=(240,101)、FST4W では(n,k)=(240,74))を用いており、通信路容量の理論限界により近づいている。
- JT9、WSPR では frequency shift keying(FSK)を用いていたが、FST4/FST4W では GFSK(Gaussian filtered frequency shift keying)を用いており、占有帯域を狭窄化している。

---

[1] 参考文献[2,3]において「S/N しきい値」は復調できる確率が 50%以上となる、2.5 kHz 参照帯域幅における S/N 比を表しています。

本稿では、FST4/FST4W のプロトコルについてプログラミング言語 MATLAB での実装を交えながら読み解いていきます[2]。なお、本稿では FST4/FST4W のうち、FST4W についての例を示して解説します[3]。FST4 と FST4W の差異はペイロードの違い(FST4 では FT8 と同じ 74 ビットであるのに対して FST4W では WSPR と同じ 50 ビット)に起因するものですが、プロトコルに採用されている各種方式は同じであり、違うのは Redundancy Check(巡回冗長検査、CRC)の生成多項式や LDPC の生成行列であるため、FST4W の概念をそのまま FST4 に応用することができます。

2. FST4W のプロトコルの詳細と MATLAB での実装

2.1 送信したい信号の準備

FST4W で送ることのできるメッセージの構成は WSPR と同様にタイプ 1~タイプ 3 があります(ただし 50 ビットペイロードへのエンコードは WSPR と異なることに注意)。本稿ではこのうちのタイプ 1 に相当するメッセージ構成について解説します。タイプ 1 で送信することができる情報は①コールサイン②グリッドロケータ③送信電力です。また、送ることができる情報の制約についても WSPR のタイプ 1[4]と同じです。①コールサインは大文字英数字とスペースから構成される 6 文字の文字列に制限されています。②グリッドロケータも通常 6 桁で使用されるもののうちの上 4 桁のみを使用します。③送信電力についても 0~60 dBm の範囲の整数値となります。

コールサインは大文字英数字とスペースから構成される 6 桁の文字列に制限され、さらに桁ごとに使用できる文字の制限があります。また、同じ文字であっても桁によって割り当てられる整数値が異なっています。1 桁目については

---

[2] 筆者が最近よく使うプログラム言語が MATLAB だから。本当は Python なんかで書くとウケがいいのかもしれませんが。

[3] 筆者はパソコンを必要としないスタンドアロンで動作する FST4W ビーコンを自作することを目的として FST4W のプロトコルの解読を試みたため、本稿では FST4W について注目しています。

スペースおよび大文字英数字が使用でき、スペースが整数値 0、数字 0~9 が整数値 1~10、大文字英字 A~Z が 11~36 に割り当てられます。2 桁目については数字 0~9 が整数値~9 に、大文字英字 A~Z が整数値 10~35 に割り当てられ、スペースを使用することはできません。3 桁目については数字 0~9 が整数値 0~9 に割り当てられ、他の文字は使用できません。4 桁目~6 桁目については、スペースが整数値 0、大文字英字が整数値 1~27 に割り当てられ、数字は使用できません。

FST4W のタイプ 1 のメッセージで送信できるグリッドロケータは 4 桁の英数字から構成され、上 2 桁の A~R の英文字は整数値 0~17 に、下 2 桁は 0~9 の数字は整数値 0~9 にそれぞれ割り当てられます。

送信電力については 0~60 dBm の範囲の整数値を送信することができます。ただし、次の節で述べるように実際にはこれに 0.3 をかけた値を整数値化してペイロードに格納するため、その分の誤差が生じえます。

2.2 50 ビットペイロードへのエンコード

2.1 で用意した情報について、以下に示す方法で可逆圧縮し、50 ビットのペイロードに格納します。なお、FST4W のペイロード長が 50 ビットであるという点については WSPR と同様ですが、可逆圧縮の方式は FST4W と WSPR とでは異なっています。

はじめにコールサインを 28 ビット長のデータとなるように圧縮します。リスト 1 に計算方法を示します。[n 桁目]とは、上から n 桁目の文字を表す整数値のことです。N_6 は 28 ビットの 2 進数で表すことが出来ます。この後に、マジックナンバーを足し合わせる処理があります。ちなみに 4194304=2^{22} ですが、2063592 については筆者もよくわかりません。最後にこの計算結果の下位 28 ビット分を取り出すために 0x7FFF(16 進数)と AND 演算を行います。リスト 1 中の N_8 が最終的に得られた計算結果となります。

リスト 1 コールサインの可逆圧縮の計算

N_1=[コールサイン 1 桁目]
$N_2=N_1\times36$+[コールサイン 2 桁目]
$N_3=N_2\times10$+[コールサイン 3 桁目]
$N_4=N_3\times27$+[コールサイン 4 桁目]
$N_5=N_4\times27$+[コールサイン 5 桁目]
$N_6=N_5\times27$+[コールサイン 6 桁目]
$N_7=N_6+2063592+4194304$
$N_8=0x7FFF\ \&\ N_7$

つぎにグリッドロケータの圧縮を行います。計算方法はリスト 2 のようになり、15 ビットの 2 進数で表すことが出来ます。

リスト 2 グリッドロケータの可逆圧縮の計算

M_1=[グリッドロケータ 1 桁目]
$M_2=M_1\times18$+[グリッドロケータ 2 桁目]
$M_3=M_2\times10$+[グリッドロケータ 3 桁目]
$M_4=M_3\times10$+[グリッドロケータ 4 桁目]

さらに送信電力の圧縮を行います。送信電力(0~60 dBm、整数値)に 0.3 をかける計算を行います。その結果は 5 ビットの 2 進数で表すことができます。

最後にコールサイン、グリッドロケータ、送信電力の情報を 50 ビット長のペイロードに格納します。図 1 に 50 ビットペイロードに格納時のビット割り当てを示します。MSB 側からコールサイン、グリッドロケータ、送信電力の順に格納していきます。これらで占有するビット数は 48 ビットであり、50 ビットペイロードの LSB 側の 2 ビットについては使用せず、0 とします。

図 1 50 ビット長ペイロードに格納時のビット割り当て

これらの処理を MATLAB で実装するとリスト 3 のようになります。

リスト 3 MATLAB で実装した 50 ビットペイロードへのエンコード

```
%% 設定
callSign = 'JA7YAA';
loc = 'QM08';
pwr = 47;


%% コールサインのコーディング
callSign = upper(callSign); % すべて大文字に統一

% matlabはunicodeが使われているのでASCIIに変換
callSign_int = uint64(unicode2native(callSign,'Shift_JIS'));

% 1文字目
if callSign_int(1,1) == 32
    callSign_int(1,1) = 0; % スペースを固有の整数値に変換
elseif ((48 <= callSign_int(1,1)) & (callSign_int(1,1) <= 57))
    callSign_int(1,1) = callSign_int(1,1) - 47; % 数字を固有の整数値に変換
elseif ((65 <= callSign_int(1,1)) & (callSign_int(1,1) <= 90))
    callSign_int(1,1) = callSign_int(1,1) - 54; % 英字を固有の整数値に変換
end

% 2文字目
if ((48 <= callSign_int(1,2)) & (callSign_int(1,2) <= 57))
    callSign_int(1,2) = callSign_int(1,2) - 48;
elseif ((65 <= callSign_int(1,2)) & (callSign_int(1,2) <= 90))
    callSign_int(1,2) = callSign_int(1,2) - 55;
end

% 3文字目
callSign_int(1,3) = callSign_int(1,3) - 48;

% 4～6文字目
for m=4:6
    if callSign_int(1,m) == 32
        callSign_int(1,m) = 0; % スペースを固有の整数値に変換
    elseif ((65 <= callSign_int(1,m)) & (callSign_int(1,m) <= 90))
        callSign_int(1,m) = callSign_int(1,m) - 64; % 英字を固有の整数値に変換
    end
end
```

```
N = callSign_int(1,1).*36 + callSign_int(1,2);  % 1-2文字目
N = N.*10 + callSign_int(1,3);  % 3文字目
N = N.*27 + callSign_int(1,4);  % 4文字目
N = N.*27 + callSign_int(1,5);  % 5文字目
N = N.*27 + callSign_int(1,6);  % 6文字目

% 謎の処理
NTOKENS = 2063592;  % 謎の数字 = 2^3 * 3 * 85983
MAX22 = 4194304;    % =2^22
N = N + NTOKENS +MAX22; % 謎の足し算
N_lgc = logical(unicode2native(dec2bin(N),'Shift_JIS')-48);
bitmask28 = logical(unicode2native(dec2bin(2.^28-1),'Shift_JIS')-48);
while length(N_lgc) < 28
    N_lgc = logical([0 N_lgc]);
end
while length(bitmask28) < length(N_lgc)
    bitmask28 = logical([0 bitmask28]);
end
N = and(N_lgc,bitmask28);

%% グリッドのコーディング
loc = upper(loc);
loc_int = uint64(unicode2native(loc,'Shift_JIS'));

loc_int(1,1:2) = loc_int(1,1:2) - 65;
loc_int(1,3:4) = loc_int(1,3:4) - 48;
M1 = ((loc_int(1,1) .* 18 + loc_int(1,2)).* 10 + loc_int(1,3)).* 10 +
loc_int(1,4);

%% パワーのコーディング
M = M1.* 32 + round(pwr.*0.3);
M = M .* 4;
M_lgc = logical(unicode2native(dec2bin(M),'Shift_JIS')-48);
while length(M_lgc) < 22
    M_lgc = logical([0 M_lgc]);
end

%% データの統合
data = [N_lgc M_lgc];
```

2.3 CRC の付与

50 ビットペイロードに格納された情報について CRC を適用します。CRC の解説はさまざまな書籍や WEB ページ[5]で見ることができます。本節では CRC について簡単に説明した後に、FST4W での CRC の利用方法について説明します。

CRC とは誤り検出符号の一種です。誤り検出とはデータに誤りがないかをチェックする仕組みのことです。例えば、送信者から伝送路を経由して受信者にデータを伝えるとき、データの信頼性を確保するためには受信者の受け取ったデータに誤りがないか確認する必要があります。そこで、あらかじめ送信側で送りたいデータに相関のある冗長なデータを付加するような符号化を施します。受信側では、これらのデータについて矛盾が生じていないかをチェックします。伝送中にデータに誤りが生じた場合、これらのデータには矛盾が生じます。このため、受信側で誤り検出が可能です。このような符号を誤り検出符号と呼びます。CRC はさまざまな場所で使用されている誤り検出符号です。

では、次に CRC での計算手順について説明します。具体的な計算方法を見ていただけばわかっていただけると思いますが、CRC の計算は割り算[4]の剰余(余り)を求めるものです。CRC の性能を決めるものとして生成多項式があります。これは割り算の除数にあたるものです。例えば、8 ビットの系列「１０１０１０００」があるとして、これを送信側から受信側に送ろうとしているとします。また、生成多項式は 5 ビットの系列「１１００１」であるとします。生成多項式のビット長が n+1 ビットであるとき「n ビット CRC」と呼びます。この例の場合は 4 ビット CRC になります。

図 2 に送信部での計算について示します。まず、送ろうとしているデータについて、(生成多項式のビット数)-1 だけ左シフトします。今回の例の場合、生成多項式は 5 ビットなので、8 ビットのデータを 4 ビットだけ左シフトして「１０１０１０００００００」とします。この 12 ビットのデータを生成多項式で割ったときの剰余を求めます。図 2 のように筆算で書くとわかりやすいです。この筆算を行う際には、加減算、乗算を行う必要がありますが、これらは 2 元有限体上で定義されているものに従います。これらについては部誌 2 号の記事でも解説しています[4]が、簡単に説明すると、加減算は排他的論理和、乗算は論理積と等価になります。また、繰り上がりや繰り下がりは存在しませ

---

[4] (被除数)÷(除数)=(商) あまり (剰余)、すなわち(商)×(除数)+(剰余)=(被除数)

ん。これらを踏まえて上記の例の割り算を行うと商は「１１０００１１１」、剰余は「１１１１」となります。最後に、12 ビットのデータから剰余「１１１１」を減算し、「１０１０１０００１１１１」を得ます。最終的に得られた 12 ビットのデータは、生成多項式で割ったときの剰余がゼロになるという特徴があります[5]。伝送路を介してこの 12 ビットのデータを受信側へ送信します。受信側では、この 12 ビットのデータを被除数、生成多項式を除数として剰余を計算します。剰余がゼロでなければ伝送中に誤りが発生していることを検出できます。

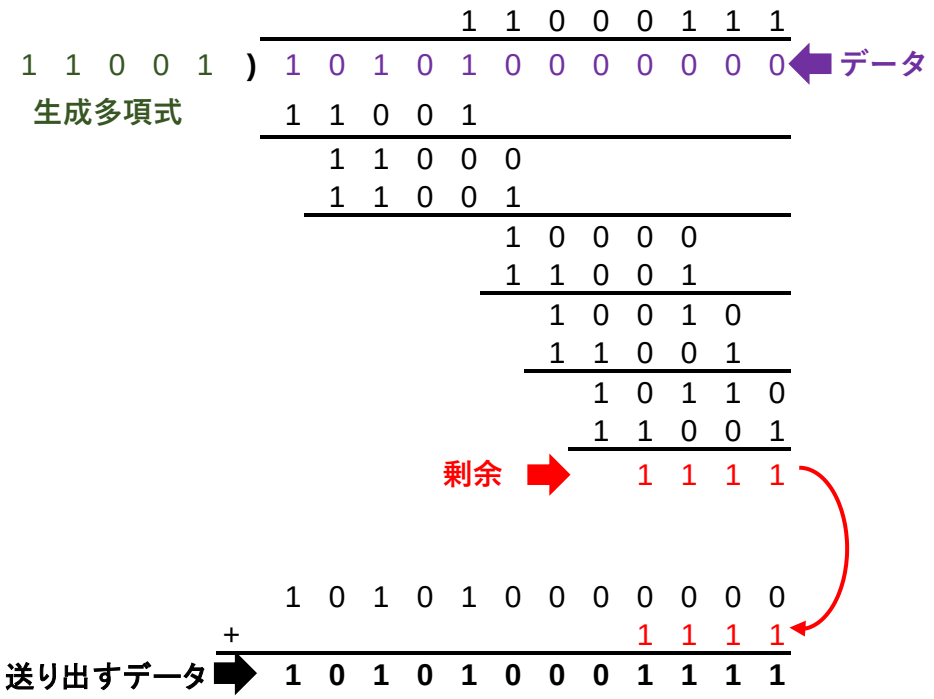


図 2 CRC の計算例

以上は CRC 自体の簡単な説明でしたが、ここからは FST4W における CRC の利用方法について説明します。FST4W で使用される CRC は 24 ビット CRC

[5] 通常の計算でいうならば、7÷2=3 あまり 1 で割り切ることができないが、(被除数)-(剰余)=7-1=6 というように被除数から剰余を引いてしまえば、6÷2=3 あまり 0 というように、割り切れるようになります。

です(すなわち、生成多項式のビット長は 25 ビット)。また、生成多項式は 16 進法で 0x100065B です[3]。50 ビットペイロードに対して 24 ビット CRC を適用するので、CRC を付与後のデータは 74 ビットとなります。

はじめに 2 元有限体上での除算の実装例をリスト 4 に示します。関数 crcRemainder は被除数と除数(生成多項式)を引数として受け取り、剰余を返します。引数と返値は logical 型の行ベクトルであり、剰余を表すベクトルの要素数は被除数のそれに一致します。

リスト 4 2 元有限体上での除算の実装

```
function y = crcRemainder(data,poly)
% 2元有限体上で除算を行う
% 入力
% data : logical型配列 データ
% poly : logical型配列 生成多項式
l_data = length(data);
l_poly = length(poly);

for m=0:(l_data-l_poly)
    if data(1,m+1) == 1
        data(1,(m+1):(m+l_poly)) = xor(data(1,(m+1):(m+l_poly)),poly);
    end
end
y = data;
end
```

FST4W における CRC の計算の実装例をリスト 5 に示します。リスト 4 の関数をそのまま使った実装となっています。

リスト 5 FST4W での CRC の計算

```
%% CRCを付与し50 bitから74 bitのデータへ
data = [data zeros(1,24)];
data_crc = data + crcRemainder(data,...
    logical(unicode2native(dec2bin(hex2dec('100065B')),'Shift_JIS')-48));
```

2.4 LDPC を用いた前方誤り訂正符号の生成

FST4/FST4W では誤り訂正符号として LDPC を用いています。WSJT 系モードでは FT4/FT8 なども LDPC を用いています。開発者の意図として、FST4

は JT9、FST4W は WSPR からの代替を標榜としているようです。JT9 や WSPR[4]は畳み込み符号を使用していますが、FST4/FST4W からは LDPC を使用することで効率よく訂正能力を向上させているのでしょう。なお、本稿では LDPC に関する理論はここでは述べず[6]、FST4W の信号生成に必要な事柄について記します。

LDPC を含む線形符号の符号化は行列を用いて記述でき、(1)式のように表すことが出来ます[6]。ここで$\boldsymbol{x}$は符号化後のデータ(符号語)を表すn列の行ベクトル、$\boldsymbol{m}$は符号化前のデータを表すk列の行ベクトル、Gは生成行列と呼ばれる符号語を生成するための$k \times n$行列です。Gの選び方は訂正能力を決める因子の 1 つです。FST4 では$(n, k) = (240,101)$、FST4W では$(n, k) = (240,74)$となっています。

$$\boldsymbol{x} = \boldsymbol{m}G \tag{1}$$

さらに、組織化符号の場合、Gは(2)式のように表すことができます。ここで、Iは$k \times k$単位行列、Pは$k \times (n - k)$行列となります。組織化符号の特徴として、符号語$\boldsymbol{x}$には符号化前のデータ$\boldsymbol{m}$が要素としてそのまま含まれていることが挙げられます[7]。FST4/FST4W にも組織化符号が採用されています。

$$G = (I \quad P) \tag{2}$$

FST4/FST4W の信号生成をするだけであれば、上記の計算方法と生成行列Gを構成する行列Pの中身を知っていればよいのです。

ここからは、FST4W での具体的な符号化の手順を説明します。まず、(2)式の行列Pの具体的な数値を WSJT-X のソースコード中から探します。これは WSJT-X の公式サイトからダウンロードできるソースコード[7]の lib¥fst4¥ldpc_240_74_generator.f90 に記載されています。リスト 6 に MATLAB で実装したGの定義を示します。リスト 6 の for 文で構成される部分はテキスト形式で格納された 16 進数を logical 型行列に変換するための処理に

---

[6] それだけで本が 1 冊かけてしまうほどの分量があるので、本稿で書くのは現実的ではありません。そもそも、筆者は十分理解できていないので書けません。

[7] $\boldsymbol{x} = \boldsymbol{m}G = \boldsymbol{m}(I \quad P) = (\boldsymbol{m}I \quad \boldsymbol{m}P) = (\boldsymbol{m} \quad \boldsymbol{m}P)$

なります。Ldpc_240_74_generation.f90 には(240-74)×240=166×240 行列として格納されているため、(2)式の定義に従うとこれはP^Tです。リスト 6 の最後の行で記述される変数 generatorMatrix がGにあたり、(2)式をそのまま実装していることが見て取れます。

リスト 6 MATLAB で実装した生成行列の定義

```
%% LDPC生成行列の準備
generatorMatrix = [];
generationMatrixChara = ...
  ['de8b3201e3c59f55a14';
  '2e06d352ebc5b74c4fc';
                                        (中略)
  'b19fcd7111a335c52ec'];

for m = 1:166
    for n=0:3
        temp = logical(unicode2native(dec2bin(hex2dec(...
            generationMatrixChara(m,(n.*4+1):(n.*4+4)))),'Shift_JIS')-48);
        while length(temp) < 16
            temp = [false temp];
        end
        generatorMatrix(m,(n.*16+1):(n.*16+16)) = temp;
    end

    temp = logical(unicode2native(dec2bin(hex2dec(...
        generationMatrixChara(m,17:19))),'Shift_JIS')-48);
    while length(temp) < 12
        temp = [false temp];
    end
    temp(:,11:12) = [];
    generatorMatrix(m,65:74) = temp;
end

generatorMatrix = [eye(74,74) generatorMatrix'];
```

この generatorMatrix を使用した LDPC への符号化をリスト 7 に示します。(1)式との対応を見ると、変数 data_crc(リスト 5 参照)が$\boldsymbol{m}$、generationMatrix がGとなります。また、この行列演算は 2 元有限体上で行われるため、実数体上の通常の行列演算を行ったあと(演算子「*」)に mod(…,2)をすることで変換を行っています。LDPC で符号化することで、74 ビットあったデータは 240 ビットの符号語に変換されます。

リスト 7 MATLAB で実装した LDPC への符号化

```
%% LDPCで符号化
data_ldpc = logical(mod(data_crc * generatorMatrix,2));
```

2.5 トーンインデックスへのマッピング

240 ビットの符号語について、1 シンボルを 2 ビットとして 120 シンボルの系列に変換します。たとえば(1)の演算で得られた符号語$\boldsymbol{x}$(あるいはリスト 7 の変数 data_ldpc)が[0 1 1 1 1 0 0 0 …]であった場合、[1 3 2 0 …]というように 4 値(0～3)のシンボルに変換します。

次にこの 120 シンボルの系列についてグレーエンコード(グレーコードへの変換)します。この時の変換表は FT8/FT4 のプロトコルについて述べられている参考文献[8]に記載されており、表 1 のようになります。グレーコードとは隣り合うシンボル同士のハミング距離が 1 となるようなコードです。ハミング距離とは、ある 2 つの 2 進数同士を比較したときにビット反転しているビットの数であり、言い換えると 2 つの 2 進数の排他的論理和の結果で 1 になっているビットの数です。表 1 の 2 進法表記のグレーコードの列を見ると、隣り合うシンボル同士は 1 ビットしか違っていません。表 1 に従ってグレーエンコーディングを行うと、たとえば[1 3 2 0 …]という系列は[1 2 3 0…]というように変換できます。

表 1 グレーエンコーディングの変換表

チャネルシンボル (変換前)		グレーコード (変換後)	
10進数	2進数	2進数	10進数
0	00	00	0
1	01	01	1
2	10	11	3
3	11	10	2

では上記の処理を MATLAB で実装してみます。リスト 8 に実装例を示します。LDPC の符号語である変数 data_ldpc がグレーエンコーディング後のチャネルシンボルである変数 chSym に変換されています。

リスト 8 MATLAB におけるトーンインデックスへのマッピングの実装例

```
%% トーンインデックスにマッピングし、120チャネルシンボルのシーケンスに変換
for m=0:119
    temp = data_ldpc(1,2.*m+1) .* 2 + data_ldpc(1,2.*m+2);
    if temp == 2
        temp = 3;
    elseif temp == 3
        temp = 2;
    end
    chSym(1,m+1) = temp;
end
```

2.6 同期ワードの挿入

受信部で信号の同期をとるためにチャネルシンボルに対して同期ワードを挿入します。「同期」とはどこから信号が始まるのか、どこから各シンボルのサンプリングを行うのかを決定するためのプロセスです。WSJT-X は復調を行う際に同期ワードの位置を検出することでタイミングの同期を行います。

同期ワードの挿入に先立って、前述の 120 チャネルシンボルについて 40 シンボルごとに分割します。120 チャネルシンボルを$a_n = a_0, a_1, \dots, a_{119}$としたとき、

$$\begin{aligned} M_A &= \{a_0, a_1, \dots, a_{29}\} \\ M_B &= \{a_{30}, a_{31}, \dots, a_{59}\} \\ M_C &= \{a_{60}, a_{61}, \dots, a_{89}\} \\ M_D &= \{a_{90}, a_{91}, \dots, a_{119}\} \end{aligned} \tag{3}$$

となるように分割します。ここで、同期ワードを

$$\begin{aligned} S_1 &= \{0,1,3,2,1,0,2,3\} \\ S_2 &= \{2,3,1,0,3,2,0,1\} \end{aligned} \tag{4}$$

と定義します。同期ワードを以下のように挿入することで 160 チャネルシンボルの系列b_nを形成します。

$$b_n = \{S_1, M_A, S_2, M_B, S_1, M_C, S_2, M_D, S_1\} \tag{5}$$

では、これらの処理の MATLAB における実装をリスト 9 に示します。見た通りそのまま実装しており、120 チャネルシーケンスである変数 chSym が 160 チャネルシーケンスである変数 b_n に変換されています。

リスト 9 MATLAB における同期ワード挿入の実装

```
%% 8シンボル同期ワードを挿入
M_A = chSym(1,1:30);
M_B = chSym(1,31:60);
M_C = chSym(1,61:90);
M_D = chSym(1,91:120);

S_1 = [0 1 3 2 1 0 2 3];
S_2 = [2 3 1 0 3 2 0 1];

b_n = [S_1 M_A S_2 M_B S_1 M_C S_2 M_D S_1];
```

2.7 ガウスフィルタによる帯域制限

ベースバンド信号に対する最後の処理として、ガウスフィルタによる帯域制限を行います。ガウスフィルタとは伝達関数がガウス関数型になっているフィルタのことです。FT4 や FT8 ではベースバンド信号に対してガウスフィルタを Low pass filter(LPF)として適用することで信号の狭帯域化をはかっていますが[8]、FST4/FST4W においても同様の処理を行っています。ここでは FIR(Finite impulse response)ディジタルフィルタにおけるガウスフィルタの設計を簡単に述べたのちに MATLAB の実装について説明します。

離散時間信号を扱うディジタルフィルタの説明に先立って、連続時間信号におけるガウスフィルタの特性について説明します。フィルタの特性を表す指標である 3 dB 帯域について考えます。これはフィルタの伝達関数において出力電力(振幅の自乗)が入力電力の$1/2$(振幅は$1/\sqrt{2}$)となるような帯域幅B(ここでは半値半幅)を表します。では、ガウスフィルタの 3 dB 帯域と時間波形にはどのような関係があるのでしょうか？まず、ガウス関数のフーリエ・逆フーリエ変換はガウス関数であることが知られており、(6)式が成り立ちます。なお、(6)式においてtは時間、ωは角周波数を表します。またαは正の実数とします。

$$\mathcal{F}[\exp(-\alpha t^2)] = \int_{-\infty}^{+\infty} exp(-\alpha t^2) \cdot e^{-j\omega t} = \sqrt{\frac{\pi}{\alpha}} \exp\left(-\frac{\omega^2}{4\alpha}\right) \quad (6)$$

すなわち、時間波形がガウス関数型ならば、その周波数スペクトルもガウス関数となります。線形フィルタにおいてはそのインパルス応答がフィルタの特性を表しています。ここでガウスフィルタのインパルス応答波形$h(t)$を以下のように仮定します。

$$h(t) = \sqrt{\frac{\alpha}{\pi}} \exp(-\alpha t^2) \quad (7)$$

(7)式をフーリエ変換すると(8)式となります。なお、$f = \omega/(2\pi)$は周波数を表します。インパルス応答のフーリエ変換はフィルタの伝達関数に一致します。(8)式は正規化されたガウスフィルタの伝達関数を表しています。

$$H(f) = \mathcal{F}[h(t)] = \exp\left(-\frac{(\pi f)^2}{\alpha}\right) \quad (8)$$

では(8)式を用いてαとBの関係を導きます。これは以下のようになります。

$$H(B) = \exp\left(-\frac{(\pi B)^2}{\alpha}\right) = \frac{1}{\sqrt{2}}$$

$$\therefore \alpha = \frac{2(\pi B)^2}{\ln(2)} \quad (9)$$

次に、連続時間領域のフィルタ特性の考察をもとに離散時間領域におけるフィルタの実装について考えます。今回は FIR フィルタでのガウスフィルタの実装について考えます。FIR フィルタとは図 3 に示す構成のディジタルフィルタです。ディジタルフィルタの詳細については各種参考書をご覧ください(たとえば東北大の電気系で使用されている教科書ならこちら[9])。

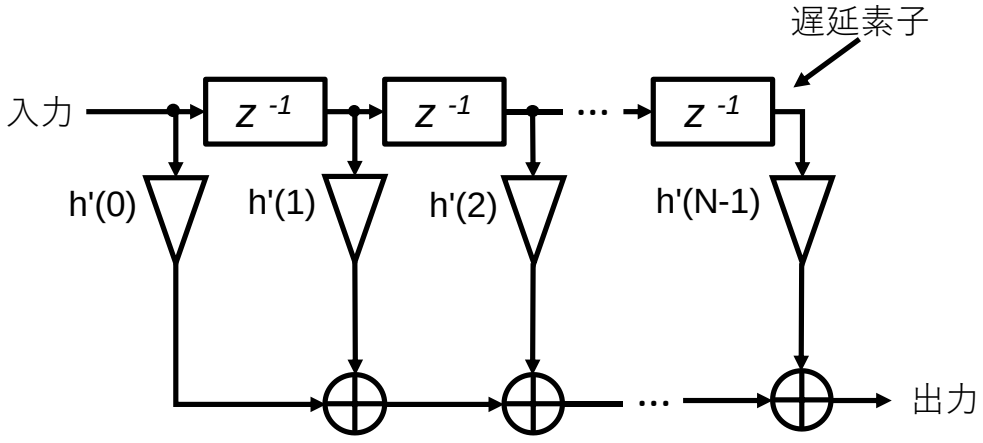


図 3 FIR フィルタの構成

図 3 の FIR フィルタの伝達関数は各タップの係数$h'(n)$によって決まります。FIR フィルタの場合、アナログフィルタのインパルス応答を離散時間化したものを各タップの係数とすることで伝達関数を設計することが出来ます。$h(t)$を連続時間領域におけるインパルス応答、Nを FIR フィルタのタップ数(奇数)、T_sをサンプリング間隔(サンプリング周波数の逆数)としたときの、n個目(nは 0 から$N-1$)のタップの係数$h'(n)$は以下のように計算して定められます。また、因果律を満たすために時間シフトを施しています。

$$h'(n) = h\left(\left(n - \frac{N-1}{2}\right)T_s\right), (0 \leq n) \tag{10}$$

(7)、(9)、(10)式を用いて各タップの係数を決定すればガウス型の FIR フィルタを実装することが出来ます[8]。

では、上記の理屈を踏まえ MATLAB での実装を行います。初めに、FIR ガウスフィルタの各タップの係数を計算する関数をリスト 10 に示します。引数として 3 dB 帯域幅、タップ数、サンプリング周波数を渡すと、各タップの係数を列ベクトルとして返します。

リスト 10 MATLAB における FIR ガウスフィルタの各タップの計算の実装例

```
function y = gaussianImpulse(B3dB,N,Fs)
% ガウシアンフィルタのインパルス応答
% B3dB : 3 dB帯域幅 [Hz]
% N : 次数(奇数)
% サンプリング周波数 [Hz]
c = 2 .* (pi .* B3dB).^2 ./ log(2);
t = ((0:(N-1)) - (N-1)./2) ./ Fs;
y = exp(-c .* t.^2);
%y = y .* hann(N)';   % 窓関数としてハン窓をかけるならコメントアウト
y = y ./ sum(y);
end
```

また、リスト 10 の関数を使用した FST4W でのガウスフィルタによる帯域制限の実装をリスト 11 に示します。なお、リスト 11 では後に述べる周波数偏移変調に関するパラメータの定義も含まれていますが、今は無視してくださ

---

[8] 窓関数の説明をしていませんが、今回は方形窓を使うということで許してください。

い。FST4W には 120 秒、300 秒、900 秒、1800 秒の T/R シーケンスがあり、それぞれについてボーレートが異なります。今回は 120 秒の場合についての実装例を示しています。リスト 9 までの実装では 1 チャネルシンボルあたり 1 サンプリングポイントで信号処理を行っていましたが、リスト 11 では 128 倍[9]にオーバーサンプリングしています。また、ゼロ次ホールドも施しています[10]。肝心のガウスフィルタによる帯域制限は最後の行で実装されています。

120 秒 T/R シーケンスにおける FST4W のボーレートは約 1.46 baud(12000 ÷8192)ですが、この T/R シーケンスにおけるガウスフィルタの 3 dB 帯域幅は約 2.93 Hz となります(FST4/FST4W は BT product=2)。また、タップ数は 51 としています[11]。

リスト 11 FST4W におけるガウスフィルタによる帯域制限の実装

```
%% GFSKで変調
%% 設定
fc = 1400;  % 中心周波数 [Hz]
Nsym = 8192/128; % 1シンボル当たりのサンプリング数(サンプリング周波数12000 Hz換算)
dftone = 12000/8192;  % 周波数スペーシング [Hz]
Fs = 12000; % サンプリング周波数 [Hz]
Amp = 0.1;  % 振幅 [a.u.]

%% 実際に計算
phase = 0;
out_wav = zeros(Fs .* 120,1);

% オーバーサンプリングと0次ホールド
b_n_x128 = zeros(1,length(b_n) .* 128);
for m=1:128
    b_n_x128(m:128:length(b_n_x128)) = b_n;
end

% ガウスフィルタを適用
b_n_g = conv(b_n_x128,gaussianImpulse(2*12000/8192,51,12000/(8192/128)));
```

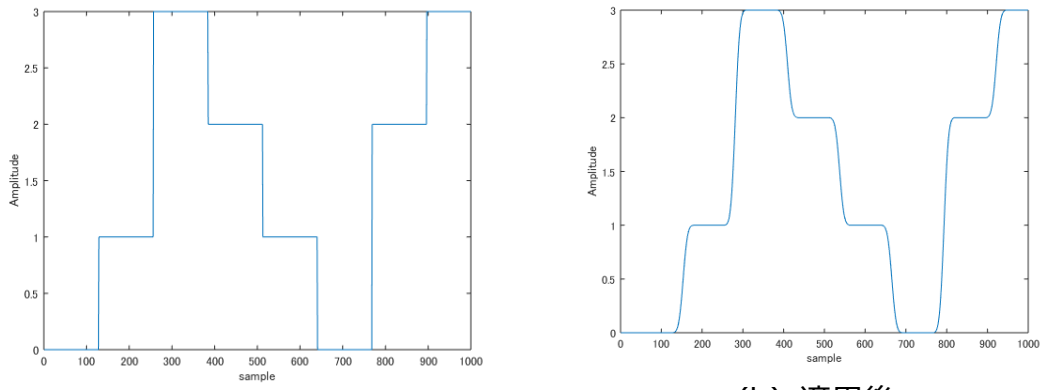
---

[9] テキトウに決めました(大丈夫か？)。ボーレートよりも十分大きく、処理が重くならない程度のサンプリング周波数でオーバーサンプリングするのが望ましいでしょう。

[10] いらないかもしれないけど。

[11] これもテキトウ。特性を見た感じ「大丈夫そう」だったので(科学技術にあるまじき主観的判断)。

ガウスフィルタの適用前後におけるベースバンド信号の時間波形を図 4 に示します。適用前は矩形的な波形をしているのに対して、適用後にはスムージングされていることがわかります。



(a) 適用前　　(b) 適用後

図 4 ガウスフィルタ適用前後のベースバンド波形

2.8 周波数偏移変調

最後にキャリア信号に対してベースバンド信号で周波数偏移変調(Frequency shift keying, FSK)を行います。FST4/FST4W ではガウスフィルタを用いてベースバンド信号に帯域制限をかけたうえで FSK しており、これは GFSK(Gaussian filtered FSK)と呼ばれます。FST4/FST4W では 1 チャネルシンボルにつき 4 値の値を取るため、4 本のトーンを立てることになります。また、FST4/FST4W ではトーンの間隔はボーレートに等しくしています(変調指数 1)。120 秒 T/R シーケンスにおける FST4W のトーン間隔は約 1.46 Hz となります。

リスト 12 に MATLAB における実装を示します。なお、特性を決めるパラメータについてはリスト 11 の初めで定義しています。今回はキャリア周波数を 1400 Hz としました。GFSK で変調した結果は audiowrite 関数を用いて WAV ファイルとして保存しています。

リスト 12 MATLAB における GFSK の実装

```
for Csym = 1:160*128
    for Csample = 1:Nsym
        phase = mod(phase + 2.*pi.*(fc + (dftone .* b_n_g(1,Csym))./Fs,2.*pi);
        out_wav(Csym.*Nsym+Csample,1) = Amp .* sin(phase);
    end
end

out_wav((length(out_wav)-round(12000.*0.9)):end,:) = [];
out_wav = [zeros(round(12000.*0.9),1);out_wav]; % DT(delay time)のあわせこみ

audiowrite([callSign,loc,num2str(pwr),'_FST4W_','.wav'],out_wav,Fs);
```

2.9 生成した信号の確認

最後にこれまでのプログラムをもちいて生成した信号について、実際に WSJT-X にデコードさせてみます。リスト 12 では生成した信号を WAV ファイルとして出力させていましたが、WSJT-X には WAV 信号を読み込む機能があります(メニューバー->ファイル->開く)。試しにコールサインを「JA7YAA」、グリッドロケータを「QM08」、空中線電力を「47」として信号を生成してデコードさせてみます。すると図 5 のようにデコードできていることが確認できました。

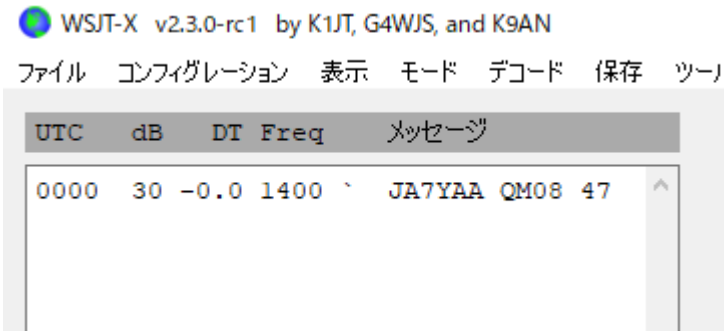


図 5 WSJT-X を用いた生成信号のデコード結果

3. まとめ

本稿では WSJT-X の新モード FST4/FST4W のうち、FST4W のプロトコルについて順を追って説明しました。今回は FST4 については説明を省略しましたが、FST4 についても FST4W と同様の原理で符号化を行っています。

参考文献

[1] Joe Tayloer, “Release: WSJT-X 2.3.0-rc1,” “https://physics.princeton.edu/pulsar/K1JT/Release_Notes.txt,” Sep. 28, 2020.

[2] Joe Tayloer, “WSJT-X 2.2 User Guide Version 2.2.2,” “https://physics.princeton.edu/pulsar/K1JT/wsjtx-doc/wsjtx-main-2.2.2.html#PROTOCOLS,” 2020/10/24 閲覧

[3] Steve Franke, Bill Somerville and Joe Taylor, “Quick-Start Guide to FST4 and FST4W,” “https://physics.princeton.edu/pulsar/k1jt/FST4_Quick_Start.pdf,” 2020/10/24 閲覧

[4] 部員 W, “MATLAB を用いた WSPR の生成：通信路符号化の例,” 東北大学アマチュア無線部部誌 Vol.2 秋号, pp.26-39, 2020.

[5] Ross N. Williams, “A Painless Guide to CRC Error Detection Algorithms,” “https://www.zlib.net/crc_v3.txt,” 2020/10/24 閲覧

[6] 和田山 正, “誤り訂正技術の基礎,” (2010), 森北出版

[7] “Source code for WSJT-X 2.3.0-rc1,” https://physics.princeton.edu/pulsar/K1JT/wsjtx-2.3.0-rc1.tgz, 2020/10/24 閲覧

[8] Steve Franke, Bill Somerville and Joe Taylor, “The FT4 and FT8 Communication Protocols,” QEX, July/August 2020, pp.7-17.

[9] 樋口 龍雄,川又 政征, “MATLAB 対応 ディジタル信号処理,” (2015), 森北出版