

低電力マイクロコントローラを使ったFFTアプリケーションの開発

従来は大型のマイクロプロセッサやASIC、DSPにしか集積されなかった周辺回路が低電力マイクロコントローラ(μC)にも集積されるようになるにつれて、低消費電力で複雑なアルゴリズムを処理することが可能になってきました。このアーティクルでは、シングルサイクルハードウェア乗算器を使った高速フーリエ変換(FFT)アプリケーション(低電力μCを使ったもの)を紹介します。

このFFTアプリケーションは、入力電圧(図1の V_{IN})のスペクトラムをリアルタイムに算出します。これを実現するために、アナログ-デジタルコンバータ(ADC)は V_{IN} をサンプリングし、そのサンプルをμCに転送します。μCは、そのサンプルに対して256点FFT計算を実行し、入力電圧のスペクトラムを得ます。計算結果を確認することができるように、スペクトラムの振幅もμCが計算し、結果をパソコンに転送してリアルタイムに表示されます。

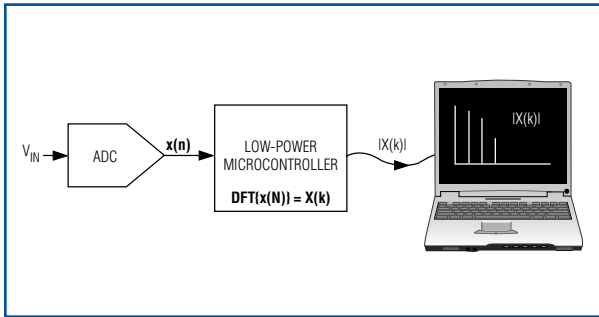


図1. 入力電圧のスペクトラムをFFTアプリケーションで計算します。

このFFTアプリケーションのファームウェアはMAXQ2000ファミリの16ビット低電力μC用にC言語で書かれています。このプロジェクトのファームウェアと回路図が japan.maxim-ic.com/AN3722 からダウンロード可能です。興味のある方は参照されるとよいでしょう。

バックグラウンド

サンプリングした入力信号のスペクトラムを求めるためには、入力サンプルの離散フーリエ変換(DFT)を行う必要があります。DFTは次式のように定義されます。

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \quad \text{for } 0 \leq k \leq N-1 \quad (\text{式1})$$

ここで、 N はサンプル数、 $X(k)$ はスペクトラム、 $x(n)$ は入力サンプルのセットです。この総和をオイラーの恒等関数で展開し、入力サンプルとスペクトラムを実数成分と虚数成分に分離すると、以下の式が得られます。

$$\begin{aligned} X_{\text{Re}}(k) &= \sum_{n=0}^{N-1} \left[x_{\text{Re}}(n) \cos\left(\frac{2\pi kn}{N}\right) + x_{\text{Im}}(n) \sin\left(\frac{2\pi kn}{N}\right) \right] \\ &= \sum_{n=0}^{N-1} \left[x_{\text{Re}}(n) \cos\left(\frac{2\pi kn}{N}\right) \right] \end{aligned} \quad (\text{式2})$$

$$\begin{aligned} X_{\text{Im}}(k) &= -\sum_{n=0}^{N-1} \left[x_{\text{Re}}(n) \sin\left(\frac{2\pi kn}{N}\right) - x_{\text{Im}}(n) \cos\left(\frac{2\pi kn}{N}\right) \right] \\ &= -\sum_{n=0}^{N-1} \left[x_{\text{Re}}(n) \sin\left(\frac{2\pi kn}{N}\right) \right] \end{aligned} \quad (\text{式3})$$

入力サンプルには実数しか存在しないため、2式と3式の総和の第2項は消えます。サンプル数を N とすると、2式と3式をそのまま算出するためには、 $2N^2$ 回の乗算と $2N(N-1)$ 回の加算が必要になります。したがって、256の入力サンプルでDFTを行うためには、131,072回の乗算と130,560回の加算が必要となります。そこでFFTの登場です!

FFTにはさまざまなアルゴリズムがあります。今回の計算で使ったradix-2アルゴリズムは一般的によく利用されるもので、DFTを2つの小さなDFTに分割するという作業をくり返します。そのためには、 N が2の累乗でなければなりません。Radix-2 FFTアルゴリズムに必要なステップは、図2に示すバタフライ演算として表すことができます。このバタフライ演算を見ればわかるように、radix-2アルゴリズムでは、 $(N/2)\log_2(N)$ 回の乗算と $N\log_2(N)$ 回の加算だけで計算することができます。図2で使われる W_N の値は「回転因子」と呼ばれ、アルゴリズムに入る前に計算することができます。

図2においてFFTへの入力、インデックスビットを反転した順番としてあります。したがって、 $N=8$ でradix-2 FFTアルゴリズムを計算するためには、入力データの順番を

0 (000b)、1 (001b)、2 (010b)、3 (011b)、4 (100b)、5 (101b)、6 (110b)、7 (111b)

から

0 (000b)、4 (100b)、2 (010b)、6 (110b)、1 (001b)、5 (101b)、3 (011b)、7 (111b)

へと変える必要があります。

これで、FFT出力が正しい順番になります。同じく図2から、FFTの次の段を計算するためには、個別のバタフライ演算の結果があれば良いこともわかります。演算は「順に」行うことから、新しい値が古い値を置きかえていくことになり、サンプル数 N のFFTを算出するために必要な変数の数は $2N$ だけでよくなります(1つの値が実数部分と虚数部分を持つため $2N$ 個の変数が必要になります)。

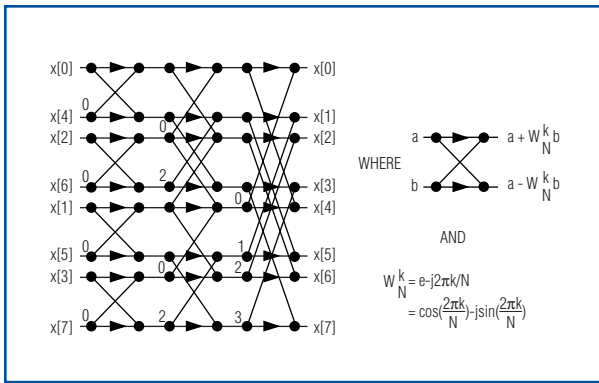


図2. バタフライ演算によって $N = 8$ のFFTを計算します。

FFT終了時、結果は複素表示ですが、式4と式5を用いると極座標に変換することができます。

$$X_{MAGN}(k) = \sqrt{X_{Re}^2(k) + X_{Im}^2(k)} \quad (\text{式4})$$

$$X_{PHASE}(k) = \arctan\left(\frac{X_{Im}(k)}{X_{Re}(k)}\right) \quad (\text{式5})$$

DSP関連の文献には、上記のDFT/FFTアルゴリズムを最適化し、高速化・小型化を進める方法がいろいろと紹介されています。中でも特に重要な(かつおそらく最も簡単な)最適化が、実数信号であるDFTの振幅が $X(N/2)$ に対して対称となることを活用したものです。

$$X(k) = X^*(N/k) \quad (\text{式6})$$

FFTのプログラミングは簡単ではありません。しかも低電力 μC には以下のようにいくつかの制約があるため、タスクがさらに難しくなります。

メモリ——今回採用した μC は2kBのRAMを持っています。FFTデータ用に2N個の16ビット変数が必要であることから、今回の μC では、Nが最大で512のFFTを行うことができることとなります。しかし、ファームウェアの他の部分でも、数バイトのRAMが消費されます。そのため、今回の演算では、Nを256までとしました。各値の実数部分と虚数部分を16ビット変数で表すため、FFTデータには1024バイトのRAMが必要となります。

速度——低電力 μC は高いMIPS/mAレーティングを持ちますが、それでも、FFTを実行するためには若干の最適化を行って命令数を減らす必要があります。幸い、今回のアプリケーションで使用したCコンパイラ(www.iar.comにあるIAR Embedded Workbench for MAXQ)には、多くの最適化のレベルや設定が用意されています。ハードウェア乗算器を上手く活用すれば、許容レベルまでプログラムを最適化することができます。

浮動小数点演算機能がない——今回の標準的な低電力 μC は、浮動小数点演算機能を持っていません。そのため、あらゆる演算を固定小数点で行うこととなります。

ファームウェアは、Q8.7法によって小数部分を表現します。したがって、ファームウェアは以下を仮定しています。

- ビット0から6が小数部を表す。
- ビット7から14が整数部を表す。
- ビット15が符号ビットを表す(2の補数)。

このような仮定があっても加算と減算には関係ありませんが、乗算では、数字がQ8.7フォーマットになるように注意する必要があります。

この記数法は、今回のFFTアルゴリズムが遭遇する可能性のある最大値を表すことが可能で、最高の精度を提供します。たとえば、今回のADCからは符号付きの8ビットサンプルが2の補数フォーマットで得られます。入力が最大振幅のDC電圧であるとする(符号付き8ビットサンプルでは127)、スペクトラム全体が $X(0)$ に含まれ、かつQ8.7法では32512に等しくなります。この数字は、符号付きの16ビット値にちょうど収まります。

ファームウェア

このセクションでは、低電力 μC でradix-2 FFTを計算するファームウェアについて解説します。ADCから読み出したサンプルは x_n_re 行列に記憶します。この行列が $x(n)$ の実数値を表します。虚数値はFFTの最初にゼロに初期化し、 x_n_im 行列に記憶します。FFT終了時、スペクトラム計算の結果が元のサンプル値を置きかえ、 x_n_re と x_n_im に記憶されます。

サンプルの収集

FFTアルゴリズムでは、サンプルは一定の周波数でサンプリングされると仮定しています。FFT用サンプルの収集は、上手くやらなければ問題が発生することがあります。たとえば、ジッタがサンプル間隔に発生するとFFT結果に誤差が混入するため、ジッタを最小限に抑える必要があります。

サンプル間隔のジッタは、ADCサンプルループに含まれる判断ステートメントによって引き起こされることがあります。今回のシステムでは符号付き8ビットサンプルをADCから読み出し、16ビット変数の行列に記憶します。リスト1に、ADCからの値の読み出しと記憶を行う関数の2種類の疑似コードアルゴリズムを示します。アルゴリズム1の方法では、サンプル値が負の場合、正の場合よりも読み出しと保存に時間がかかり、サンプル間隔にジッタが発生します。

リスト1. ADCサンプリングの2種類の疑似コードアルゴリズムを示します。1番目のアルゴリズムではサンプル間隔にジッタが発生しますが、2番目のアルゴリズムではそのような問題は起きません。

```
// ALGORITHM 1: INCONSISTENT SAMPLING
FREQUENCY - BAD!
```

```
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
```

```

doADCSampleConversion() //
Instruct ADC to sample Vin
    sample[i] = read8BitSampleFromADC() //
Read 8-bit sample from ADC
    if (sample[i] & 0x0080) // If
the 8-bit sample was negative
        sample[i] = sample[i] + 0xFF00 //
Make the 16-bit word negative
end
// ALGORITHM 2: FIXED SAMPLING FREQUENCY -
GOOD!
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
    doADCSampleConversion() //
Instruct ADC to sample Vin
    sample[i] = read8BitSampleFromADC() //
Read 8-bit sample from ADC
end
for i = 0 to (N-1)
begin
    if (sample[i] & 0x0080) // If
the 8-bit sample was negative
        sample[i] = sample[i] + 0xFF00 //
Make the 16-bit word negative
end

```

三角関数ルックアップテーブル

このFFTアルゴリズムでは、コサインやサインの値を計算する代わりにルックアップテーブル(LUT)を用います。サインとコサインのLUTとなる宣言をリスト2に示します。実際のファームウェアには、これらのLUTを自動的に生成するプログラムのソースコードもコメントとして記載されています。回転因子のインデックスが0から(N/2)-1までとなるため、LUTはいずれも、N/2個の要素を持ちます(図2参照)。

リスト2. コサイン関数とサイン関数のLUT。

```

const int cosLUT[N/2] = {+128,+127,+127, ...
,-127,-127,-127};
const int sinLUT[N/2] = {+0 ,+3 , +6, ...
,+9 , +6, +3};

```

これらのLUTを記憶した行列は定数として宣言しているため、コンパイラはこれらをデータ空間ではなくコード空間に記憶するように強制します。なお、LUTの値もQ8.7法に従って表記しなければならないため、実際のコサインおよびサインの値に 2^7 をかけた値となります。

ビット反転

ビット反転の順番(Nは既知)は、ランタイムに計算する、ルックアップテーブルを使ってインデックス化する、展開

したループとして直接書き込むと、いろいろな方法が考えられます。どの方法を選んでも、ソースコードのサイズと実行速度という面でトレードオフがあります。今回のFFTアプリケーションでは、展開ループによってビット反転を行うことにしました。こうするとソースコードは長くなりますが、実行速度は速くなります。この展開ループのコードをリスト3に示します。アプリケーションファームウェアには、この展開ループを自動的に生成するプログラムのソースコードもコメントとして記載されています。

リスト3. N = 256の展開ループによってビット反転を行います。

```

i=x_n_re[ 1]; x_n_re[ 1]=x_n_re[128];
x_n_re[128]=i;

i=x_n_re[ 2]; x_n_re[ 2]=x_n_re[ 64];
x_n_re[ 64]=i;

i=x_n_re[ 3]; x_n_re[ 3]=x_n_re[192];
x_n_re[192]=i;

i=x_n_re[ 4]; x_n_re[ 4]=x_n_re[ 32];
x_n_re[ 32]=i;

...

i=x_n_re[207]; x_n_re[207]=x_n_re[243];
x_n_re[243]=i;

i=x_n_re[215]; x_n_re[215]=x_n_re[235];
x_n_re[235]=i;

i=x_n_re[223]; x_n_re[223]=x_n_re[251];
x_n_re[251]=i;

i=x_n_re[239]; x_n_re[239]=x_n_re[247];
x_n_re[247]=i;

```

Radix-2 FFTアルゴリズム

ビット反転によってサンプルの並び順を変更したら、FFT計算に入ります。今回のradix-2 FFT用ファームウェアでは、図2に示すバタフライ演算を3つのメインループで実行します。外側のループでは、FFT演算の $\log_2(N)$ 段をカウントします。内側のループでは、各段におけるバタフライ演算を実行します。

このFFTアルゴリズムの核心となるのは、各バタフライ演算を実行する、短いコードブロックです。このブロックをリスト4に示しますが、残念ながら、今回実行するものの中でこの部分だけは、ポータブルでないファームウェアとなっています。MUL_1マクロとMUL_2マクロは μC のハードウェア乗算器を使って乗算を1命令サイクルで実行するものです。これらのマクロはMAXQ2000特有のものであり、その内容は、実際のファームウェアでは細かく検討可能です。

リスト4. バタフライ演算をCで行います。

```

/* (1) Macro MUL_1(A,B,C): C=A*B (result
in Q8.7)*/
/* (2) Macro MUL_2(A,C) : C=A*last_B (result
in Q8.7)*/
MUL_1(cosLUT[tf],x_n_re[b],resultMulReCos);

```

```

MUL_2(sinLUT[tf],resultMulReSin);
MUL_1(cosLUT[tf],x_n_im[b],resultMulImCos);
MUL_2(sinLUT[tf],resultMulImSin);
x_n_re[b] = x_n_re[a]-
resultMulReCos+resultMulImSin;
x_n_im[b] = x_n_im[a]-resultMulReSin-
resultMulImCos;
x_n_re[a] = x_n_re[a]+resultMulReCos-
resultMulImSin;
x_n_im[a] =
x_n_im[a]+resultMulReSin+resultMulImCos;

```

複素座標から極座標への変換

V_N スペクトラムの振幅を求めるには、複素表現による $X(k)$ を極座標に変換する必要があります。今回のファームウェアでは、リスト5に示すようにこの変換を行います。FFTの計算結果はファームウェアにとって不要となるため、それを振幅値で置きかえます。

リスト5. FFTの結果を複素表記から極座標に変換します。

```

const unsigned char magnLUT[16][16] =
{
{0x00,0x10,0x20, ... ,0xd0,0xe0,0xf0},
{0x10,0x16,0x23, ... ,0xd0,0xe0,0xf0},
...
{0xe0,0xe0,0xe2, ... ,0xff,0xff,0xff},
{0xf0,0xf0,0xf2, ... ,0xff,0xff,0xff}
};

...
...

/* Compute x_n_re=abs(x_n_re) and
x_n_im=abs(x_n_im) */

...
...

x_n_re[0] = magnLUT[x_n_re[0]>>11][0];

for(i=1; i<N_DIV_2; i++)
x_n_re[i] =
magnLUT[x_n_re[i]>>11][x_n_im[i]>>11];

x_n_re[N_DIV_2] =
magnLUT[x_n_re[N_DIV_2]>>11][0];

```

式4を実際に計算をするのではなく、2次元のLUTを使って振幅を求めます。最初のインデックスにはスペクトラムの実数成分の4つの最上位ビット(MSB)を用い、2番目のインデックスにはスペクトラムの虚数成分の4MSBを用います。4MSBは、符号付き16ビット値を右に11回シフトして取得します。スペクトラムの実数成分と虚数成分は

絶対値を取ってからインデックスとして使用するため、符号ビットはゼロとなります。

式6から、スペクトラムの振幅は $X(N/2)$ に対して対称となることから、最初の $(N/2) + 1$ 個のスペクトラム値についてのみ極座標への変換を行います。また、入力サンプルが実数であれば $X(0)$ と $X(N/2)$ の虚数部分がゼロになることも明らかです。このため、これら2カ所の振幅は個別計算してあります。なお、実際のファームウェアには、 $X(k)$ の振幅用のLUTを自動的に生成するプログラムのソースコードもコメントとして記載されています。

ハミングウィンドウとハンニグウィンドウ

このプロジェクトのファームウェアには、ハミングウィンドウあるいはハンニグウィンドウを入力サンプルに適用するためのLUT (Q8.7フォーマット)も用意してあります。ウィンドウ関数を適用すると、時間領域における $x(n)$ の切り捨てから生じるスペクトルリークを低下させることができます。ハミングウィンドウ関数は式7、ハンニグウィンドウ関数は式8のとおりです。

$$h(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right) \quad (\text{式7})$$

$$h(n) = 0.5 \left[1 - \cos\left(\frac{2\pi n}{N-1}\right) \right] \quad (\text{式8})$$

リスト6は、これらの関数を実行するためのコードです。実際のファームウェアには、やはり、これらのウィンドウ関数用のLUTを自動的に生成するプログラムのソースコードもコメントとして記載されています。

リスト6. ハミングウィンドウ関数とハンニグウィンドウ関数のLUT。

```

const char hammingLUT[N] = {+10, +10, +10, ...
,+10, +10, +10};
const char hannLUT[N] = { +0, +0, +0, ...
, +0, +0, +0};

...
...

for(i=0; i<256; i++)
{
#ifdef WINDOWING_HAMMING

MUL_1(x_n_re[i],hammingLUT[i],x_n_re[i]); //
x(n)*=hamming(n);

#endif

#ifdef WINDOWING_HANN

MUL_1(x_n_re[i],hannLUT[i],x_n_re[i]); //
x(n)*=hann(n);

#endif
}

```

結果の検証

FFTアプリケーションの結果をテストするため、ファームウェアは、 $X(k)$ の振幅を μC のUARTポートからパソコンにアップロードします。パソコンのシリアルポートからこの振幅値を読むために作られたFFT Graphというソフトウェアを使って、算出したスペクトラムをリアルタイムに表示させます(このソフトウェアも、このプロジェクトのファームウェアに含まれています)。図3が、 μC を使い、200kspsで入力電圧をサンプリングし、計算した結果をFFT Graphで表示させたものです。入力信号は以下の4種類です。

- a) 4.3V DC信号
- b) 50kHzのサイン波
- c) 70kHzのサイン波
- d) 6.25kHzの方形波

さらなる発展

興味を持たれたら、今回のFFT演算の最適化や構成にじっくりと時間をかけてみられたらいいでしょう。今回のアーティクルではradix-2アルゴリズムを使用しましたが、必要な加算回数や乗算回数をもっと劇的に減らしてくれるアルゴリズムも存在します。また、FFTの高速化を実現することができる最適化手法も、このアーティクルで取りあげた以外に各種存在します。たとえば、入力サンプルが実数であれば入力サンプルの虚数部分は常にゼロとなるため、スペクトラムの前半だけが意味を持ちます。

この情報を活用してFFTの初段と最終段を最適化すれば高速化が可能ですが、プログラム領域の消費量は増えます。

このアーティクルで紹介したアルゴリズムは、低電力 μC 用にかかれたFFTアルゴリズムの良い出発点であると言えます。このアプリケーションのファームウェアには細かいコメントが記入してありますので、詳しい情報や実行の詳細についてはそちらをご覧ください。

参考文献

Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics Computation*, Vol. 19, pp 297-301, 1965.

Lemieux, Joe, "Fixed-point math in C," *Embedded Systems Programming*, October 2003.

Proakis, John G. and Dimitris G. Manolakis, *Digital Signal Processing Principles, Algorithms, and Applications*, 3rd Edition, Prentice Hall, 1996.

Smith, Steven W., *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd Edition, California Technical Publishing, 1999.

*Embedded Systems Design*の2005年10月号にも、同様のアーティクルが掲載されています。

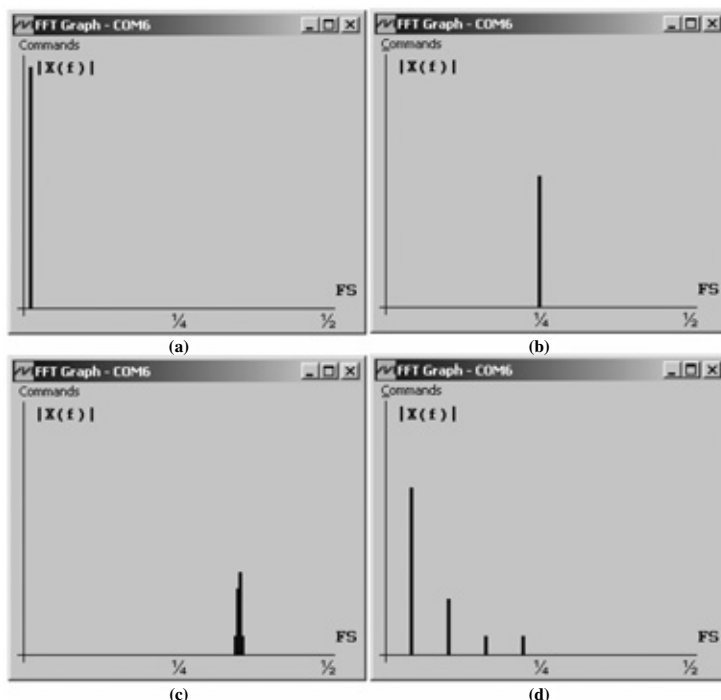


図3. FFT Graphを用いて低電力 μC で算出したスペクトラムを表示します。