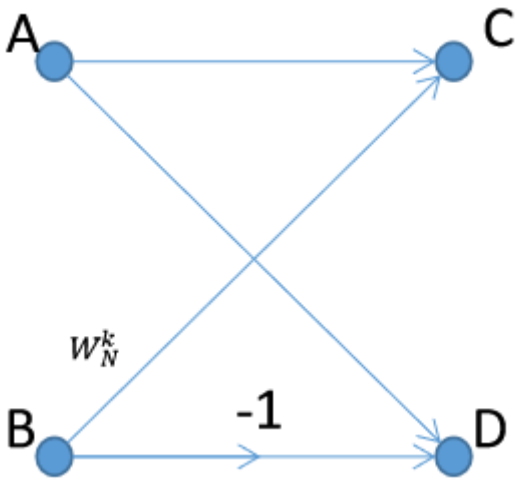
基于FFT快速优化算法在MSP430G2553单片机上实现128点快速傅里叶变换

一、项目简介及实现原理

- 项目简介
  - 基于入门级单片机MSP430G2553（主时钟16MHz，512B RAM, 16KB FLASH）（对刚刚学习单片机的童鞋来讲，这款芯片的资源已经足够，但是稍微增加一些复杂的算法或者功能就会很费劲，比如FFT算法。如果使用最直接简单的FFT，不进行优化，则最多可以运行64点FFT，主要消耗资源为RAM，512字节的RAM不允许开辟很大的数组）。
  - **FFT（Fast Fourier Transform）即快速傅里叶变换**，是**DFT（Discrete Fourier Transform，离散傅里叶变换）**的高效算法，并**不是一种新的变换**。
  - FFT是DFT的高效算法，但FFT也有一些神奇的“变种”。参考文章**《几种特殊的FFT算法》**。本项目利用其中的实数FFT算法用64点FFT程序框架实现128点FFT。
- 基本FFT原理
  - 前言：

    如果你看到了这篇文章，那么我应该默认你已经懂得了傅里叶变换是干什么的（用2553实现FFT，全网基本上就没有，但凡想实现FFT功能的也不会选2553（老人地铁手机），所以这篇文章是学长总结给为了学分而来的学弟们学妹们），并且也了解了DTFT，DFT，FFT等各种形式的傅里叶变换之间的联系（即便不理解也不影响使用本程序），此处就不进行详细阐述了。一是一篇短文章不太可能把伟大深邃的傅里叶变换讲通透；二是本文的重点是对理论转换到程序进行描述，着重于软件实现；三是笔者水平并不高，自己也并未将傅里叶变换理解的很透彻，仅仅是了解FFT的过程以及应用范畴和各种参数的含义，具体的数学推导以及整个理论的支撑建议大家从《信号与系统》-->《DSP数字信号处理》中寻找答案。
  - FFT是啥？
    简单来说，FFT就是一种快速计算DFT的方法，DFT是**有限离散傅里叶变换**，是将离散的数据进行**离散时间傅里叶变换（DTFT）**后得到的频域连续函数再进行离散化（单独的点数据，不连续，称为离散）得到的。所以DFT的输入数据和输出数据都是离散的。既然FFT只是DFT的快速算法，那么FFT的输入输出也是离散的，且计算过程比DFT快，而计算机处理模拟量（连续的）比处理离散化数据难的多，所以现在计算机普遍使用FFT进行数据处理。

    FFT后的数据有什么用呢？实际上，这个问题涉及到了很多领域，因为傅里叶变换在每一个领域几乎都可以找到它的身影。数字信号处理这门课程实际上就是在讲傅里叶变换，围绕傅里叶变换对数字信号进行处理，那么是不是可以说只要涉及到数字信号的领域都可以用傅里叶变换作为工具去分析呢？实际上，物联网，机械震动，通信传输，图像处理等很多你想到的，想不到的领域，都会使用到傅立叶变换。最直接的用处就是用作**信号的频谱分析**。
  - **FFT的计算过程**(本章节参考此篇知乎，基本框架代码来自该文章，优化部分为原创。想全面了解算法请查看此篇文章，想了解优化部分，可以略过此部分，直接跳到程序优化)
    整个FFT的计算过程一般都以图形方式表示，蝶形运算就是其中的精髓。下图即为一个简单的蝶形运算：



    表示的结果为：

$$C = A + W_N^k B$$

$$D = A - W_N^k B$$

    其中W_N^k是旋转因子，旋转因子定义为：$W_N^{kn} = e^{-j\frac{2\pi}{N}kn}$
    由该定义可以推出以下性质：
    1）周期性

$$W_N^{m+lN} = W_N^m，其中：W_N^{-m} = W_N^{N-m}$$

2）对称性

$$W_N^{m+\frac{N}{2}} = -W_N^m$$

3）可约性

$$W_{\frac{N}{m}}^{k} = W_N^{mk}$$

4）特殊的旋转因子

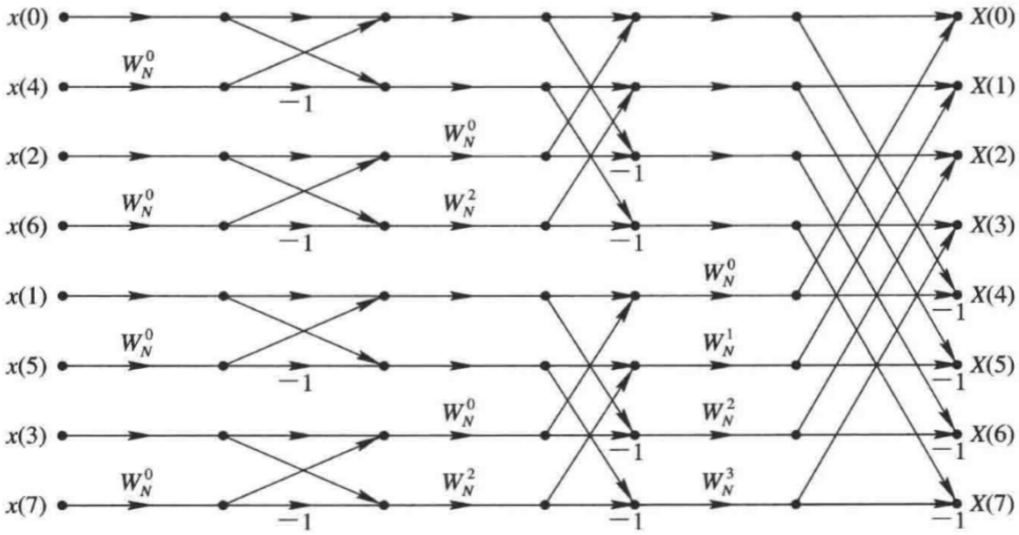
$$W_N^0 = 1$$

这些性质的运用，使得DFT的运算量大大减少，也是FFT能够高效快速运算的核心。

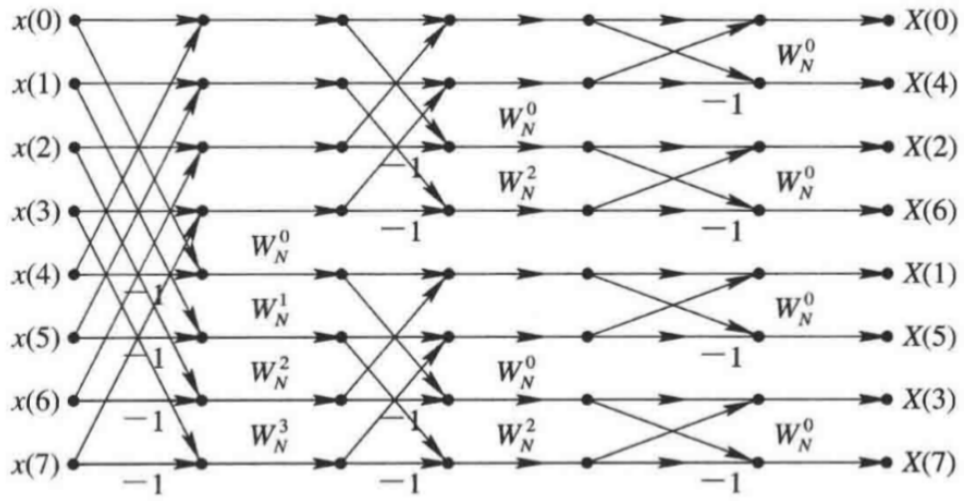
了解了蝶形运算的规则，我们还需要知道FFT的运算流程，FFT有很多种不同的抽取运算方式，大体分为时序抽取和频域抽取。

大家只要将两者抽取的图形对比一下，就可以知道，这两种方案其实是相同的，只不过把输入和输出的标号顺序交换了一下，一个是从奇偶序到顺序（基2时序抽选）一个是从顺序到奇偶序（基2频域抽选）。下图分别为时序和频域抽选的计算流程图：

基2时序抽选法



基2频域抽选法



看到上面的流程图，我们就可以来分析整个FFT的运算结构以及转化成C语言程序所需要的步骤了（仅针对基2时域抽选）：

- 1）输入序列为划分奇偶后的序列，因此，我们需要先进行码位倒序，即将原来顺序排列的数据按照新的规则分开重新排列。

- 2）进行蝶形运算的设计，这一点极为头痛，尤其是看到这个流程图之后，似乎完全没有办法用一些程序思维将其转化。最开始我也是这么认为，但好在人类的强大就是可以把每一代的智慧结晶都保存下来，供我这种不太灵光的人借鉴，指引前进的方向。于是乎，在互联网的帮助下，找到了这篇优秀的**文章**，完整的阐述了FFT蝶形运算的规律，以及如何将其糅合进C语言程序中。看完整个文章以后，发现所有的思路都归结“找规律”这三个字，人为的寻找不同的参数之间的规律，并且将参数之间的规律融合，由此形成了一个完整的算法程序。

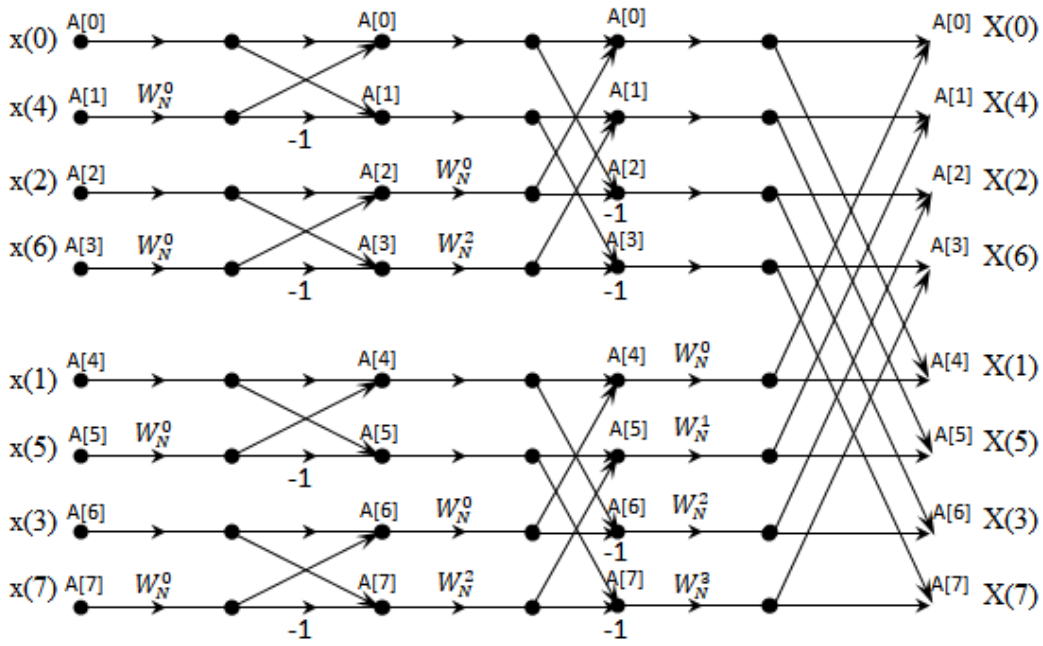
- 程序是怎样炼成的

  首先从8点FFT的运算图中寻找规律，下图是8点FFT的运算流程图，首先规定几个符号：

  1）此图$N = 8 = 2^3$，表示为8点FFT

  2）左边x(n)代表输入的数组，A[n]代表C语言储存的数组，右边的X(k)代表运算后的数组，即经过FFT变换后的数组。

  3）W_N^p是旋转因子，p为指数，p相同则旋转因子相同。



找规律：

- 根据左边的x(n)的顺序，我们知道在x(n)输入到整个蝶形运算时，顺序发生了变化，我们称之为**码位倒序**，即用二进制表示时，码位倒序后的数的二进制为原数二进制的反序，如下表所示：

顺序		反序	
十进制	二进制	二进制	十进制
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

因此，整个程序应该先将原序列数据经过码位倒序，重新排列后，再输入到FFT程序进行计算。实现码位倒序的程序比较多，最基本的可以从原理入手。

例如，设I为原的数组下标，J为码位倒序后的数组下标，从左到右遍历I的所有位，遍历的同时，将当前的位的二进制值，按照从右到左的顺序赋给J，最终完成码位倒序。也可以选择从两边向中间遍历，最后将J变成码位倒序。

这里不使用这两种方案，因为涉及到的位操作对于C语言而言效率不高，写起来也比较繁琐。

还有一种方案：

原数组下标I是按照顺序排列的，那么下一个数组下标一定是是上一个数组下标加1得到的，即I满足从右到左的进位规则。例如：1011001的下一个数一定是1011010。那么在经过码位倒序后，就会满足从左到右的进位规则，例如，1011001的码位倒序为1001101，其下一个数是1011010，码位倒序为0101101。可以看出0101101是1001101按照从左到右加1得到的（将最左边看做低位，最右边看做高位，进行加法运算）。
我们采用此种方案进行码位倒序的程序设计，开始程序之前，我们需要知道一下几点：

- 1.第一个下标和最后一个下标的反序都是自身，比如0000,1111反序都是自身；第二个下标和第N/2个下标互为反序，比如00001和10000（N=32）；
- 2.无论反序后的下标是如何排列的，反序前后的同一位置对应的下标只有三种情况，相等，小于，或者大于，所以我们只需要在大于或者小于的情况下互换数据，就能实现所有数据仅交换一次，此处选择原下标小于倒序下标时进行互换。
- 3.程序中i代表原下标，j代表倒序下标。由于FFT实际上是复数运算，因此我们用dataR和dataI分别储存实部和虚部。
- N一定为2^n

了解上述规律后，我们开始程序设计，为了遍历所有的元素，我们需要从0下标遍历到N-1下标，可能有的同学会问，根据上边的表格，遍历一半的不就可以把所有的数据都交换了吗？而且还省下了时间，不过这仅仅是N=8 的情况，举个例子当N=128时，原下标为1101111，表示十进制为111，其反序为1111011，表示123，所以遍历一半并不能完全走遍所有情况。所以我们应该从1到N-2（0和N-1的反序是自身）。框架如下：

```
for(i=1;i<=N-2;i++)
{
    if(i<j)//交换数据
    {
        Temp=dataR[i];//实部
        dataR[i]=dataR[j];
        dataR[j]=Temp;

        Temp=dataI[i];//虚部
        dataI[i]=dataI[j];
        dataI[j]=Temp;
    }
    /*
      反序运算得到J
      */
}
```

接下来，我们考虑如何得到反序J，从左到右的进位规律可以这么理解，从左到右加1后，如果最高位为1，那么该位变为0，并向后继续查找，如果仍为1，该位也变为0，直到下一位是0，把0变为1，停止查找；如果最高位为0，则直接变为1，停止查找。由于我们是从1开始，那么1的反序一定是N/2，故J的初始值设为N/2，程序如下：

```
j=N/2;
for(i=1;i<=N-2;i++)
{
    if(i<j)//交换数据
    {
        Temp=dataR[i];//实部
        dataR[i]=dataR[j];
        dataR[j]=Temp;

        Temp=dataI[i];//虚部
        dataI[i]=dataI[j];
        dataI[j]=Temp;
    }
    //开始下一个反序J的计算
    k=N/2;//从最高位开始判断
    while(1)//从高到低位依次判断，直到该位为0时跳出循环
    {
        if(j<k)//该位为0
        {
            j=j+k;//该位置1
            break;
        }
        else//该位为1
        {
            j=j-k;//该位置0
            k=k/2;//下一位
        }
    }
    _nop();//MSP430系列内置空操作函数，常用于debug
}
```

自此，码位倒序的程序我们就设计好了。

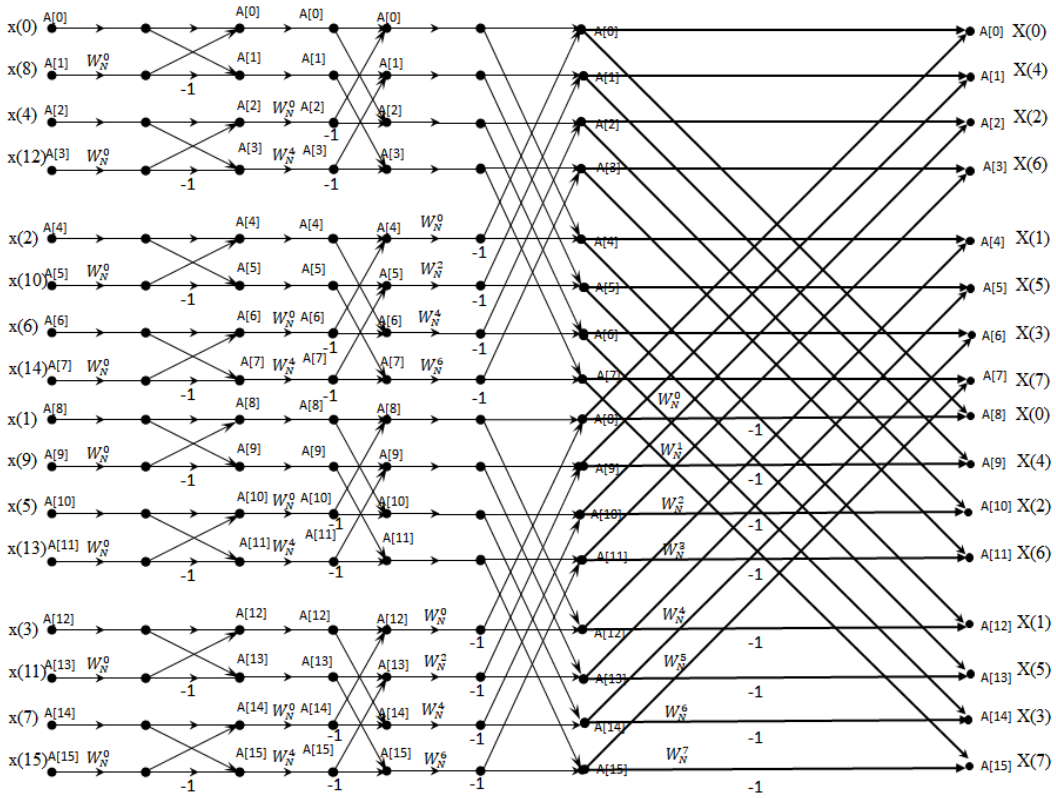
接下来，就是蝶形运算的设计，蝶形运算相比于码位倒序规律更多，而且需要从公式中推导，但无论如何，只要有规律，就一定能用程序设计出来，稍稍复杂一些而已。因此，我们使用N=8和N=16两种蝶形图来寻找运算流程的规律。

首先我们要知道蝶形运算的程序实现需要哪些变量：

1）两个数据输入的间隔B

2）旋转因子W：第L级旋转指数P、第L级旋转因子的种类、旋转因子的增量

其中L表示FFT的级数（相同的蝶形运算称作一级，下图一共有4级），$L=1，2，3...M$。M表示FFT点数的幂，即若$N=8=2^3=2^M$，则M=3。旋转因子表示为W_N^p，p表示旋转因子的指数。



根据蝶形图，我们可以得到如下规律：

1.第L级运算中，每个蝶形运算的输入数据的间隔B，$B=2^{L-1}$。L=1，每个蝶形运算输入数据相隔1个，$B=2^0=1$；L=2，每个蝶形运算输入数据相隔2个，$B=2^1=2$；L=3，每个蝶形运算输入数据相隔4个，$B=2^2=4$；

2.第L级运算中，旋转因子种类数目，L=1，只有一种（W_N^0）；L=2，有2种（W_N^0,W_N^4）；L=3，有4种（W_N^0,W_N^2,W_N^4,W_N^6）；故第L级，有2^{L-1}种旋转因子。

3.第L级旋转因子的增量$k=2^{M-L}$

4.第L级中同一旋转因子之间的间隔2^L

5.第L级同种蝶形运算的运算次数2^{M-L}

- 程序设计

首先我们应该完成M级运算，故首先需要有M次循环，其次，对于每一级蝶形运算，我们知道一共需要计算2^{M-L}次，但并不是所有的运算都用同样的旋转因子，因此需要先对每一种旋转因子计算。

```
for(L=1; L<=M;L++)  //FFT蝶形级数L从1到M
{
    /*第L级的运算*/
    //先计算同一蝶形运算的数据输入间隔 B = 2^(L-1);旋转因子种类也等于B
    B = Pow2M[L-1];//Pow2M是一个存有2^M的值的数组，通过查询方式实现2的幂次计算
    for(j=0; j<B;j++)
    {
        /*同种蝶形运算，包括旋转因子*/
        //计算旋转指数p的增量k=2^(M-L)
        k = Pow2M[M-L];//7级时，k=1
        //计算旋转指数p，增量为k时，则p=j*k（相当于加法，每一种比前一种多k）
        p=j*k;
        for(i = 0; i <= k - 1; i++)//开始计算，同种蝶形运算的次数也为k
        {
            /*进行蝶形运算*/
            //数组下标定为r
            r = j + 2 * B * i;//蝶形运算的第一个输入为j，加上增量2*B；i*2*B避免累加
            TEMP=A[r];
            A[r]=A[r]+A[r+B]W^p_N;
```

```
                A[r+B]=TEMP-A[r+B]W^p_N;//W^P_N表示旋转因子，P为指数
                _nop();
            }
        }
    }
```

以上就为FFT运算的框架，但是具体的蝶形运算仍未给出，因为还要将旋转因子的运算进行拆解，让计算机也能够运算。需要说明的是：**上述程序运算蝶形运算，将计算出的值直接重新赋值给原数组，这是FFT中节省运算空间的做法，因为FFT运算过后就不需要再使用已经运算过的数据，因此可以这样做而不影响结果。**

将蝶形运算的数学公式推导成计算可以计算的式子需要一定的DSP基础和数学功底，为了简单化问题，仅对推导思路做一个大体的解释：

$$蝶形运算=\begin{cases}A[r]_L=A[r]_{L-1}+A[r+B]_{L-1}W_N^P\\A[r+B]_L=A[r]_{L-1}-A[r+B]_{L-1}W_N^P\end{cases}\quad（下标L表示第L级）$$

该公式最大的问题在于旋转因子是e为底的指数，首先使用欧拉公式将其化解为三角函数形式的复数形式

$$W_N^p=e^{-j\frac{2\pi}{N}p}=cos(\frac{2\pi}{N}p)-i*sin(\frac{2\pi}{N}p)$$

其中i表示虚数单位，则蝶形公式最后都可以用实部和虚部的复数形式表示，即

$$蝶形运算=\begin{cases}A[r]_L=A[r]_{L-1}+A[r+B]_{L-1}W_N^P\\A[r]_{L-1}=X_R[r]+iX_I[r]//R下标为实部，I下标为虚部\\\\A[r+B]_L=A[r]_{L-1}-A[r+B]_{L-1}W_N^P\\A[r+B]_{L-1}=X_R[r+B]+iX_I[r+B]\\\\W_N^p=e^{-j\frac{2\pi}{N}p}=cos(\frac{2\pi}{N}p)-i*sin(\frac{2\pi}{N}p)\\\\A[r+B]_{L-1}W_N^P=T_R+iT_I//同上\end{cases}$$

以上述公式转换为核心，可得到：

$$\left\{T_R+iT_I=A[r+B]_{L-1}W_N^P=(X[r+B]_R+iX[r+B]_I)(cos(\frac{2\pi}{N}p)-i*sin(\frac{2\pi}{N}p))\right\}$$

$$=X[r+B]_R*cos(\frac{2\pi}{N}p)+X[r+B]_I*sin(\frac{2\pi}{N}p)+i*(X[r+B]_I*cos(\frac{2\pi}{N}p)-X[r+B]_R*sin(\frac{2\pi}{N}p))$$

$$=>\begin{cases}T_R=X[r+B]_R*cos(\frac{2\pi}{N}p)+X[r+B]_I*sin(\frac{2\pi}{N}p)\cdots\cdots(1)\\\\T_I=X[r+B]_I*cos(\frac{2\pi}{N}p)-X[r+B]_R*sin(\frac{2\pi}{N}p)\cdots\cdots(2)\end{cases}$$

将式

$$\begin{cases}A[r]_{L-1}=X_R[r]+iX_I[r]\\A[r+B]_{L-1}W_N^P=T_R+iT_I//同上\end{cases}$$

带入

$$\begin{cases}A[r]_L=A[r]_{L-1}+A[r+B]_{L-1}W_N^P\\\\A[r+B]_L=A[r]_{L-1}-A[r+B]_{L-1}W_N^P\end{cases}$$

得到：

$$\begin{cases}A[r]_L=X_R[r]+iX_I[r]+(T_R+iT_I)//R下标为实部，I下标为虚部\\\\A[r+B]_L=X_R[r]+iX_I[r]-(T_R+iT_I)\end{cases}$$

为了方便表示第L级的运算结果，我们仍然将$A[r]_L$的实部虚部分开表示，令

$$\begin{cases}A[r]_L=A_R[r]+iA_I[r]\\\\A[r+B]_L=A_R[r+B]+iA_I[r+B]\end{cases}$$

那么有

$$\begin{cases}A_R[r]=X_R[r]+T_R\cdots\cdots（3）\\\\A_I[r]=X_I[r]+T_I\cdots\cdots（4）\\\\A_R[r+B]=X_R[r]-T_R\cdots\cdots（5）\\\\A_I[r+B]=X_I[r]-T_I\cdots\cdots（6）\end{cases}$$

通过以上式（1）到式（6），我们就可以通过编程的方式来计算蝶形运算了。

$$\begin{cases} T_R = X[r+B]_R * cos(\frac{2\pi}{N}p) + X[r+B]_I * sin(\frac{2\pi}{N}p) \cdots\cdots (1) \\ T_I = X[r+B]_I * cos(\frac{2\pi}{N}p) - X[r+B]_R * sin(\frac{2\pi}{N}p) \cdots\cdots (2) \\ A_R[r] = X_R[r] + T_R \cdots\cdots （3） \\ A_I[r] = X_I[r] + T_I \cdots\cdots （4） \\ A_R[r+B] = X_R[r] - T_R \cdots\cdots （5） \\ A_I[r+B] = X_I[r] - T_I \cdots\cdots （6） \end{cases}$$

对于该式中的A[r]数组，是复数形式的，而C语言并没有处理复数对应的数据结构，故需要用两个数组分别对应其实部和虚部，所以FFT运算部分的整体程序如下：

```
for (L = 1; L <= M; L++)  //FFT蝶形级数L从1到M
{
    /*第L级的运算*/
    //先计算同一蝶形运算的数据输入间隔 B = 2^(L-1);旋转因子种类也等于B
    B = Pow2M[L - 1];//Pow2M是一个存有2^M的值的数组，通过查询方式实现2的幂次计算
    for (j = 0; j < B; j++)
    {
        /*同种蝶形运算，包括旋转因子*/
        //计算旋转指数p的增量k=2^(M-L)
        k = Pow2M[M - L];//7级时，k=1
        //计算旋转指数p，增量为k时，则p=j*k（相当于加法，每一种比前一种多k）
        p = j * k;
        for (i = 0; i <= k - 1; i++)//开始计算，同种蝶形运算的次数也为k
        {
            /*进行蝶形运算*/
            //数组下标定为r
            r = j + 2 * B * i;//蝶形运算的第一个输入为j，加上增量2*B；i*2*B避免累加
            Tr = dataR[r + B] * cos_fft64(p) + dataI[r + B] * sin_fft64(p);//此处的cos与sin函数的运算与大家常用的C语言库函数不同，原因是G2553单片机的RAM有限，为了不加重处理器的负担，仍然使用查表法计算。而且该数值也并不是cos(p)的值，而是cos(2.0*PI*p/N)的值。
            Ti = dataI[r + B] * cos_fft64(p) - dataR[r + B] * sin_fft64(p);

            dataR[r + B] = (dataR[r] - Tr);
            dataI[r + B] = (dataI[r] - Ti);
            dataR[r] = (dataR[r] + Tr);
            dataI[r] = (dataI[r] + Ti);
            _nop();
        }
    }
}
```

此处解释一下可能的符号混淆，$X_R[r]$和$X_I[r]$实际上就是当前未经过运算的数组的实部与虚部，从前面的定义即可看出，所有式子（2）至式（6）都是用当前的数组的实数与虚数在进行运算，算出的结果赋值给了A[r]数组的实部与虚部，而又因之前我们提到过的FFT每一次运算结果后均不会再次用到对应的存储空间的值，故直接将结果赋给对应的数组，也即

```
dataR[r] = (dataR[r] + Tr);   dataI[r] = (dataI[r] + Ti);
```

这一代码的来源。

二、实际程序优化过程

- 通过上面的基础理论，我们已经得到了最基本的FFT计算程序，并且我也提到过，G2553的性能对于FFT来说太过紧张，我们必须进行一些“大刀阔斧”的创新与优化，才能使得MSP430G2553运行点数更高的FFT，有的同学可能会问，运行64点不就够了嘛，何必这么费劲的运行128点，甚至256点？首先，运行128点的FFT分辨率会更高，尤其是在用LCD屏幕来显示图形的时候；其次，不优化运行64位FFT几乎将单片机的资源占用完，若还要运行其他程序，就会出错；优化过的程序性能更好，也可以与其他程序同时运行，何乐不为？并且写出一个不健壮的程序，会造成更多未知的bug。不断的对已有的知识进行创新与优化才是人类在科学道路上进步的阶梯。

  下面先总结一下一共有哪几方面需要更改：

  1.在目前的程序中，所有浮点数的运行一定要以float运行。因为在编译过程中会默认以double类型运行，耗费巨量的flash空间（储存程序）以及运行时间。（如果你发现你写的程序中，编译器编译出了占用大量字节的非用户程序，那很可能是某一些程序中用到了double精度的运算）举例：

```
sum=a*b*1.5;//会被编译器默认调用double精度的运算函数运算
sum=a*b*1.5f;//人为指定用float型
//调用一些函数时也可以查找有没有以float类型运行的
sqrtf(Tr * Tr + Ti * Ti);//以float精度运行的开根号函数
```

2.sin与cos函数需要先用matlab或excel计算出对应的值（并非简单的sin(x)的值），编译时会将常量存入flash，而不占用RAM空间，节省RAM资源，加快运算速度。

3.以上述FFT算法为核心，加入**实数FFT快速算法**，节省RAM。

- 接下来正式介绍128点FFT的程序：

1.算法理论

实数FFT算法：基本的FFT算法是通过实部虚部分开计算，得出结论，但实际应用中很多都是实数，并没有虚部，但在计算中必须要开辟出虚数数组，使得算法得以进行，以及后续计算。而实数FFT可以缩减一半的空间，以64点的基本FFT来计算128点的实数FFT。

推导过程分为两部分，**第一**：用N点的FFT同时实现两个的N点实数序列的DFT。**第二**：将一个2N点的实数序列拆分成两个N点序列，利用前面提到的算法分别计算出两个序列的FFT，最后组合，实现N点FFT程序实现2N点实数序列的DFT。

不给出理论推导，仅给出转换公式：

（1）若有两个实数序列$x(n)$和$y(n)$，则可以将其组合为新序列：$a(n)=x(n)+i*y(n)$其中i为虚数单位
输入到基础的FFT程序中可得到$a(n)$的DFT结果设为$A(k)$，可推得$A(k)=X(k)+i*Y(k)$，其中$X(k)$和$Y(k)$为$x(n)$和$y(n)$的DFT结果
又推得$A(N-k)=X^*(k)+i*Y^*(k), A^*(N-k)=X(k)-i*Y(k)$其中*表示共轭。（实际上这是一条性质，FFT的共轭对称）则用这两式可以得出：

$$X(k)=\frac{1}{2}[A_R(k)+A_R(N-k)]+\frac{i}{2}[A_I(k)-A_I(N-k)]\ldots\ldots（7）$$
$$Y(k)=\frac{1}{2}[A_I(k)+A_I(N-k)]+\frac{i}{2}[A_R(N-k)-A_R(k)]\ldots\ldots（8）$$
$$k=0,1,2,3\ldots N-1$$

故可用N点FFT得到两个N点实序列的FFT。

（2）若有一2N点的实序列$x(n)$，对其进行奇偶抽取，得到如下两序列：

$$\begin{cases} g(n)=x(2n) \\ h(n)=x(2n+1) \qquad n=0,1,2,\ldots,N-1 \end{cases}$$

此时利用上一条算法，将两个序列用N点FFT运算可得：

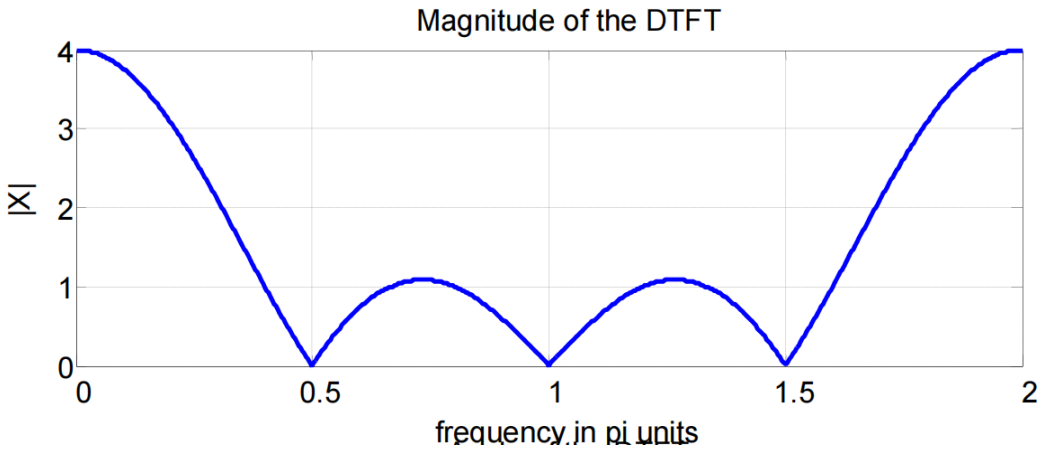
$$G(k)=\frac{1}{2}[A_R(k)+A_R(N-k)]+\frac{i}{2}[A_I(k)-A_I(N-k)]\ldots\ldots（9）$$
$$H(k)=\frac{1}{2}[A_I(k)+A_I(N-k)]+\frac{i}{2}[A_R(N-k)-A_R(k)]\ldots\ldots（10）$$
$$k=0,1,2\ldots N-1$$

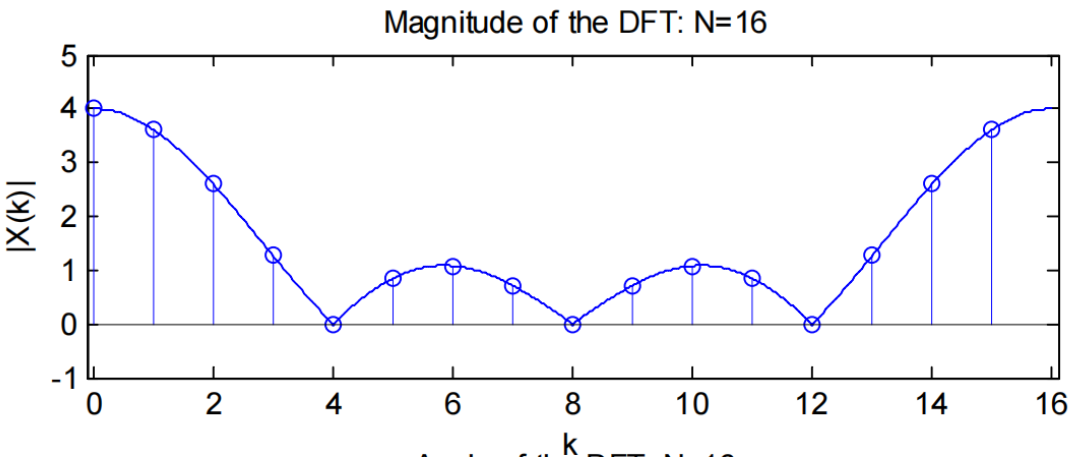
又根据蝶形运算的特点有：

$$\begin{cases} X(k)=G(k)+W_{2N}^{k}H(k) \\ X(N+k)=G(k)-W_{2N}^{k}H(k) \quad k=0,1,2\ldots\ldots N-1 \end{cases}\ldots\ldots（11）$$

有了如上公式我们就可以进行实数FFT程序的编写了，但以上公式其实是不完整的。有的同学可能注意到，k的取值有从0开始，而第N点并没有计算，应该怎么求？这就要深入细致的去理解FFT计算结果的共轭对称性与DTFT的对应关系了。

FFT的对称性是关于$\frac{N}{2}$点共轭对称，也就是说从0到N-1点中是以$\frac{N}{2}$为中心共轭对称，那么1与N-1共轭对称，0应该与N对称，但实际上我们只取N点的FFT结果，也就是0到N-1，第N点实际上是没有计算数值的。若利用傅里叶变换的平移特性，我们能发现0与N点值应该相同。在DTFT中，值是连续的，我们察觉不到第N点的值的变化，将其离散化以后（FFT的结果）由于只取N点，那么第N点的值就被舍去，与下图对照可以更直观的理解，这是离散傅里叶变换中的一个小坑，希望能引起大家注意！





所以，第N点其实就是第0点，则将$A_R(0)=A_R[N], A_I(0)=A_I(N)$带入可以得到：

$$X(0)=\frac{1}{2}[A_R(0)+A_R(0)]+\frac{i}{2}[A_I(0)-A_I(0)]=A_R(0).....（12）$$

$$Y(0)=\frac{1}{2}[A_I(0)+A_I(0)]+\frac{i}{2}[A_R(0)-A_R(0)]=A_I(0).....（13）$$

以上就是实数FFT的理论公式，在此总结一下，为了将计算所需的空间压缩，我们将N长的数组划分为两个$\frac{N}{2}$长的数组，并且组合为一个复数数组，带入FFT程序计算$\frac{N}{2}$点FFT，再由计算结果导出两个数组分别进行FFT的结果，最后再将两个数组的FFT结果进行一次蝶形变换，就得到了最终的实数FFT结果。

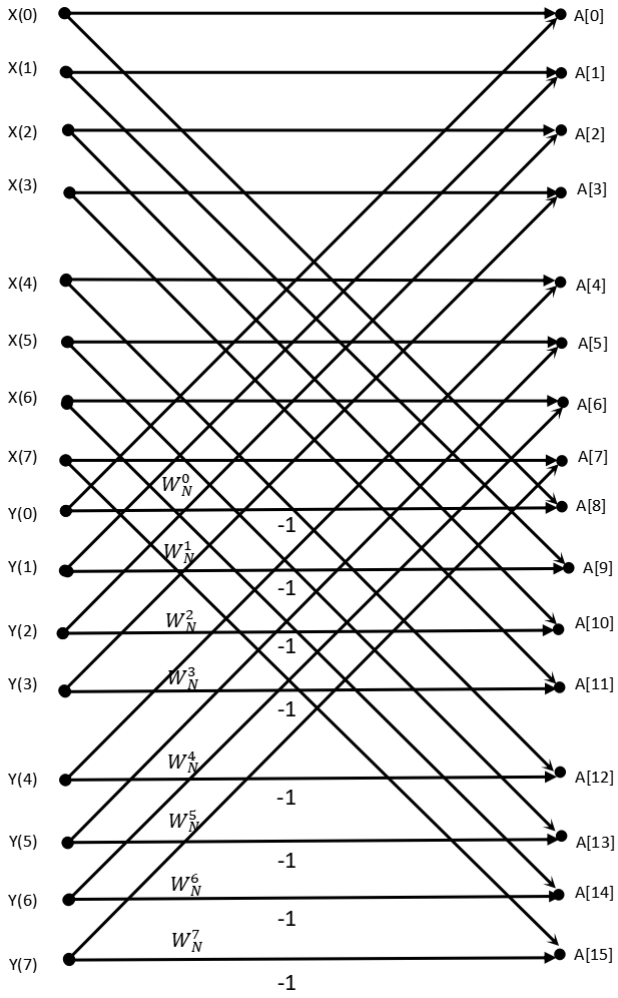
2.程序计算

下面给出的程序是在MSP430G2553单片机上实际运行的程序，为了便于理解，直接使用注释的方式对程序解释。

```
/*此函数将长度为2*FFTN的数组进行奇偶分组，即将偶序下标的数据分为一组，寄序下标的数据又分为一组*/
/*此函数不能够放于FFTR程序中，因为置换数组的过程中会占用128字节的RAM资源，若放于同一函数中，排序所占用的资源不会被释放，造成程序崩溃跑飞，程序不能正常运行*/
/*数据类型为int，因为FFT用于计算采集波形的频谱，ADC采集的数据为int，存放于adcbuff数组中，该数组为128位，占用256字节RAM。由于整个项目以该数组为核心，故256字节将常驻RAM，不会被释放*/

/*参数含义：FFTRN和FFTN为宏定义，值分别为128和64*/
void FFTR_SEQ()//分奇偶，重复利用储存空间
{
    unsigned char i;
    int temp[FFTN];//开辟128字节缓存空间，一个int占用2字节，共64个
    for (i = 0; i < FFTN; i++)
    {
        temp[i] = adcbuff[2*i+1];//抽取其中奇序部分，放入缓存数组
    }
    for (i = 0; i < FFTN; i++)
    {
        adcbuff[i]= adcbuff[2 * i];//将偶序部分放入数组的前64个空间
    }
    for (i = 0; i < FFTN; i++)
    {
        adcbuff[FFTN+i] = temp[i];//将奇序部分放入数组的后64个空间
    }
}
```

```
/*实数FFT程序，调用此函数前应该先调用一次FFTR_SEQ()函数*/
/*由于RAM资源只有512字节，adcbuff已经占用了256，因此，我们只能用adcbuff来储存运算结果，这样会降低运算结果的精度（float类型被截短为int），不过经过我的实际应用，最后的运算结果不会受太大影响，作为频谱图的相对分析是完全可行的*/
void FFTR()
{
    unsigned int k;
    float Tr, Ti;
    int *dataR;
    dataR = (int *)adcbuff;//函数指针指向数组的首地址，也是偶序数组的首地址
    int *dataI;
    dataI= (int *)&(adcbuff[64]);//函数指针指向第64个数组单元，也是奇序数组的首地址
    FFT(dataR, dataI);//调用FFT程序，将组合而成的复数数组进行64点FFT运算

    //先求X1(k)和X2(k)，即从复数FFT的结果求dataR和dataI的FFT结果，再将X1和X2蝶形组合
    float x1R, x2R, x1I, x2I, x1I1, x2I1;
    /*此处为程序核心，由公式（7）（8）我们知道，需要用到64位复数结果计算X1和X2数组，数据的利用顺序为从两头向中间靠拢，并且，在第32位时出现数据重合，此后将重复利用，因此，我们必须要在数据重复之前，将结果运算完毕，否则，就会出现数据在被利用之前就已经覆盖的情况*/
```

```
    for (k = 0; k <= FFTRN / 4; k++)//32个循环，计算64点,在32次以后，由于重复利用数组，会出现数据覆盖，损失dataR，dataI的前32位，因此需要在32次内将后32位计算完成
    {
        if (k == 0)//由之前的推导可知，0点的值需要单独计算
        {
            x1R = dataR[k];//公式12
            x1I = 0;
            x2R = dataI[k];//公式13
            x2I =  0;
            Tr = x1R + x2R;/*此处Tr并不是公式中的Tr，此处将公式（1）（3）联合，即=x1R+x2R*COS(0)+x2R*SIN(0)=x1R+x2R;需要注意的是，公式（1）（3）中的序号r与r+B代表的其实是输入蝶形运算的两个数据顺序，只要顺序一致就可以用此公式计算。*/
            Ti = x1I + x2I;/*此处Tr并不是公式中的Tr，此处将公式（1）（3）联合，即=x1R+x2R*COS(0)+x2R*SIN(0)=x1R+x2R;需要注意的是，公式（1）（3）中的序号r与r+B代表的其实是输入蝶形运算的两个数据顺序，只要顺序一致就可以用此公式计算。*/
            adcbuff[k] = sqrtf(Tr * Tr + Ti * Ti);//直接计算功率

            Tr = x1R - x2R;//根据蝶形运算的规律，公式（5）（6）
            Ti = x1I - x2I;//根据蝶形运算的规律，公式（5）（6）
            adcbuff[FFTRN/2] = sqrtf(Tr * Tr + Ti * Ti);
        }
        else
        {
            //前32位，针对X(K)和Y(K)
            x1R = (dataR[k] + dataR[FFTRN / 2 - k]) / 2.0f;//公式7
            x1I = (dataI[k] - dataI[FFTRN / 2 - k]) / 2.0f;
            x2R = (dataI[k] + dataI[FFTRN / 2 - k]) / 2.0f;//公式8
            x2I = (dataR[FFTRN / 2 - k] - dataR[k]) / 2.0f;

            //后32位，针对X(K)和Y(K)
            //x1R1 = (dataR[FFTRN / 2 - k] + dataR[k]) / 2.0;//计算结果一样，用x1R代替
            x1I1 = (dataI[FFTRN / 2 - k] - dataI[k]) / 2.0f;//公式7，K= FFTRN/4 + k，带入公式
            //x2R1 = (dataI[FFTRN / 2 - k] + dataI[k]) / 2.0;
            x2I1 = (dataR[k] - dataR[FFTRN / 2 - k]) / 2.0f;//公式8

            //以上公式计算X(k) Y(k)，以下公式便是根据蝶形图计算得出，公式依据与FFT复数程序一致，只是标号不同，
            //为了便于理解，给出此蝶形运算的图示（代码下方）
            Tr = x2R * cos_fft128(k) + x2I * sin_fft128(k);
            Ti = x2I * cos_fft128(k) - x2R * sin_fft128(k);
            Tr = x1R + Tr;
            Ti = x1I + Ti;
            adcbuff[k] = sqrtf(Tr * Tr + Ti * Ti);

            Tr = x1R - Tr;
            Ti = x1I - Ti;
            adcbuff[FFTRN/2+k] = sqrtf(Tr * Tr + Ti * Ti);

            //因子中cos关于FFTRN / 2 并不对称！！
            Tr = x2R * cos_fft128(FFTRN / 2 -k) + x2I1 * sin_fft128(FFTRN / 2 -k);
            Ti = x2I1 * cos_fft128(FFTRN / 2 -k) - x2R * sin_fft128(FFTRN / 2 -k);
            Tr = x1R + Tr;
            Ti = x1I1 + Ti;
            adcbuff[FFTRN/2-k] = sqrtf(Tr * Tr + Ti * Ti);

            Tr = x1R - Tr;
            Ti = x1I1 - Ti;
            adcbuff[FFTRN-k] = sqrtf(Tr * Tr + Ti * Ti);

            _nop();
        }
    }

    for (k = 0; k < FFTRN; k++)
    {
        adcbuff[k] = adcbuff[k]/FFTRN;//缩小一下数据范围
    }
}
```



一次蝶形运算图示

以上，便是优化程序的整体代码，其中cos_fft等函数及数组定义如下：

```
/*
 * FFT.C
 *
 *  Created on: 2021年1月7日
 *      Author: 24172
 */
#define M 6
#define point 128
#define FFTRN 128
#define FFTN 64
#define PI 3.1415926f
const unsigned int Pow2M[8]={1,2,4,8,16,32,64,128};//为了加快运行速度，将2的M次幂储存下来，通过查表获取值。unsigned int类型为16位2进制，最高储存数为65535
const float sin_fft_64[]={
                          0, 0.09801714, 0.19509032, 0.29028468, 0.38268343, 0.47139674, 0.55557023, 0.63439328, 0.70710678, 0.77301045,
                          0.83146961, 0.88192126, 0.92387953, 0.95694034, 0.98078528, 0.99518473, 1
};
/*const float cos_fft_64[]={
                          1, 0.99518473, 0.98078528, 0.95694034, 0.92387953, 0.88192126, 0.83146961, 0.77301045, 0.70710678, 0.63439328,
                          0.55557023, 0.47139674, 0.38268343, 0.29028468, 0.19509032, 0.09801714, 0

};*/
const float sin_fft_128[]={0,    0.049068,   0.098017,   0.14673,    0.19509,    0.24298,
 0.290285,   0.33689,    0.382683,   0.427555,
                           0.471397,   0.514103,   0.55557,    0.595699,   0.634393,   0.671559,
0.707107,   0.740951,   0.77301,    0.803208,
                           0.83147,    0.857729,   0.881921,   0.903989,   0.92388,    0.941544,
0.95694,    0.970031,   0.980785,   0.989177,
                           0.995185,   0.998795,   1
};
/*const float cos_fft_128[]={
```

```
                1,  0.998795,   0.995185,   0.989177,   0.980785,   0.970031,   0.95694,    0.941544,
0.92388,    0.903989,   0.881921,
                0.857729,   0.83147,    0.803208,   0.77301,    0.740951,   0.707107,   0.671559,
0.634393,   0.595699,   0.55557,
                0.514103,   0.471397,   0.427555,   0.382683,   0.33689,    0.290285,   0.24298,
0.19509,    0.14673,    0.098017,
                0.049068,   0
};*/
//减少使用储存空间，反复利用已有的数据，实现查表
float sin_fft64(unsigned char i)
{
    if(i<=16)
        return sin_fft_64[i];
    else if(i<=32)
        return sin_fft_64[32-i];
    else if(i<=48)
        return -sin_fft_64[i-32];
    else if(i<=64)
        return -sin_fft_64[64-i];
    else
        return 0;
}

float cos_fft64(unsigned char i)
{
    if(i<=16)
        return sin_fft_64[16-i];//cos_fft_64[i];
    else if(i<=32)
        return -sin_fft_64[i-16];//-cos_fft_64[32-i];
    else if(i<=48)
        return -sin_fft_64[48-i];//-cos_fft_64[i-32];
    else if(i<=64)
        return sin_fft_64[i-48];//cos_fft_64[64-i];
    else
        return 0;
}


float sin_fft128(unsigned char i)
{
    if(i<=32)
        return sin_fft_128[i];
    else if(i<=64)
        return sin_fft_128[64-i];
    else if(i<=96)
        return -sin_fft_128[i-64];
    else if(i<=128)
        return -sin_fft_128[128-i];
    else
        return 0;
}

float cos_fft128(unsigned char i)
{
    if(i<=32)
        return sin_fft_128[32-i];//cos_fft_128[i];
    else if(i<=64)
        return -sin_fft_128[i-32];//-cos_fft_128[64-i];
    else if(i<=96)
        return -sin_fft_128[96-i];//-cos_fft_128[i-64];
    else if(i<=128)
        return sin_fft_128[i-96];//cos_fft_128[128-i];
    else
        return 0;
}
```

到此为止，这篇文章就要结束了，算是对自己当初探究FFT时候的总结，也为后来者提供一点帮助。后续也会在这个仓库随缘更新一些经过实践的、基于MSP430G2553开发板平台、关于采集波形，波形数据处理，LCD显示的一些代码，供大家参考学习，若大家发现了相关问题，也可以提出来。