DSP 实验指导书

——基于 TMS320C54x

2010.7

目录

实验说明

DSP 是一门理论与实践并重的技术，在学习了 DSP 的结构体系与基本原理以后，必须配合通过一些典型的 DSP 实验，以加深对 DSP 软、硬件的理解与掌握，同时学会 DSP 的开发工具的使用，了解 DSP 应用系统的开发环境与开发过程，从而为今后从事 DSP 的开发打下扎实的基础。

本实验指导书包含了 5 个实验，分为三种类型，供师生查阅。

1）基础性实验，熟悉和了解 DSP 的开发环境，初步学会程序的编写与 DSP 开发工具 CCS 的使用，基础实验为必做实验。

2）DSP 在片的外设与硬件接口实验，通过 DSP 的开发平台和实验系统，进一步加深对 DSP 硬件系统的理解，初步学会包括在片的外设，I/O 接口，A/D、D/A 接口、串口等的使用，教师可根据学时选做 1-2 种；

3）应用程序设计与调试，这一部分实验是综合性的，除了学会 DSP 的程序编写与调试外，要求学生综合运用数字信号处理的基本理论、MATLAB 软件，在 DSP 上实现信号处理有关算法进而了解和学会基于 DSP 的信号处理系统的开发，教师可根据学时选做。

以下各个实验程序都是基于 TMS320C5416 的实验程序，其中在数字滤波器的设计时，需要借助于 Matlab 来进行滤波器的仿真。编译所用到的 CCS 版本为 2.20.28 或者是更高版本。

1 基础实验：CCS的使用与DSP开发环境

1.1 实验目的

1. 熟悉 CCS 集成开发环境，掌握工程的生成方法；
2. 熟悉 SEED-DEC5416 实验环境；
3. 掌握 CCS 集成开发环境的调试方法；

1.2 实验内容

1. 编译与链接的设置，生成可执行的 DSP 文件；
2. 进行 DSP 程序的调试与改错；
3. 学习使用 CCS 集成开发工具的调试工具；
4. 观察实验结果；

1.3 实验程序结构说明

本实验包含的文件如下：

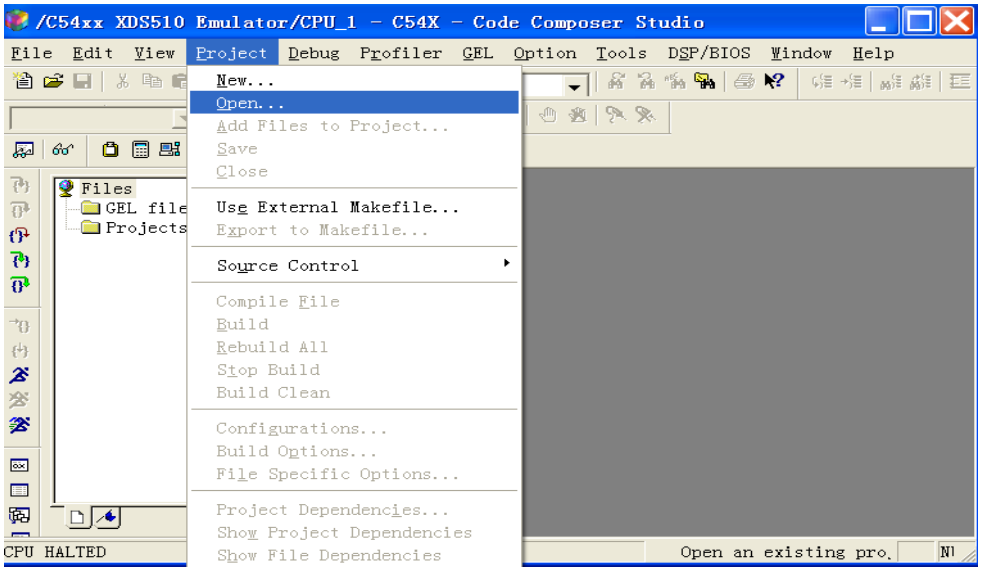
1. MATH.c 这个文件中包含了实验中关于 DSP 运算的主要函数。主要包含有：
   fixed_add(int x,int y)：　　定点加法运算；
   fixed_sub(int x,int y)：　　定点减法运算；
   fixed_mul(int x,int y)：　　定点乘法运算；
   fixed_div(int x,int y)：　　定点除法运算；
   float_add(double x,double y)：　浮点加法运算；
   float_sub(double x,double y)：　浮点减法运算；
   float_mul(double x,double y)：　浮点乘法运算；
   float_div(double x,double y)：　浮点除法运算；
   float_fixed(double x)：　　浮点转定点运算；
   fixed_float(int x)：　　定点转浮点运算；
2. math.cmd 这是 DSP 的链接文件。它的主要功能是将 DSP 的每段的程序链接到相应的 DSP 的存贮区中。
3. rts.lib 是一个库文件，主要包含了有关 C 的运行环境与相应的函数的代码。

1.4 实验步骤

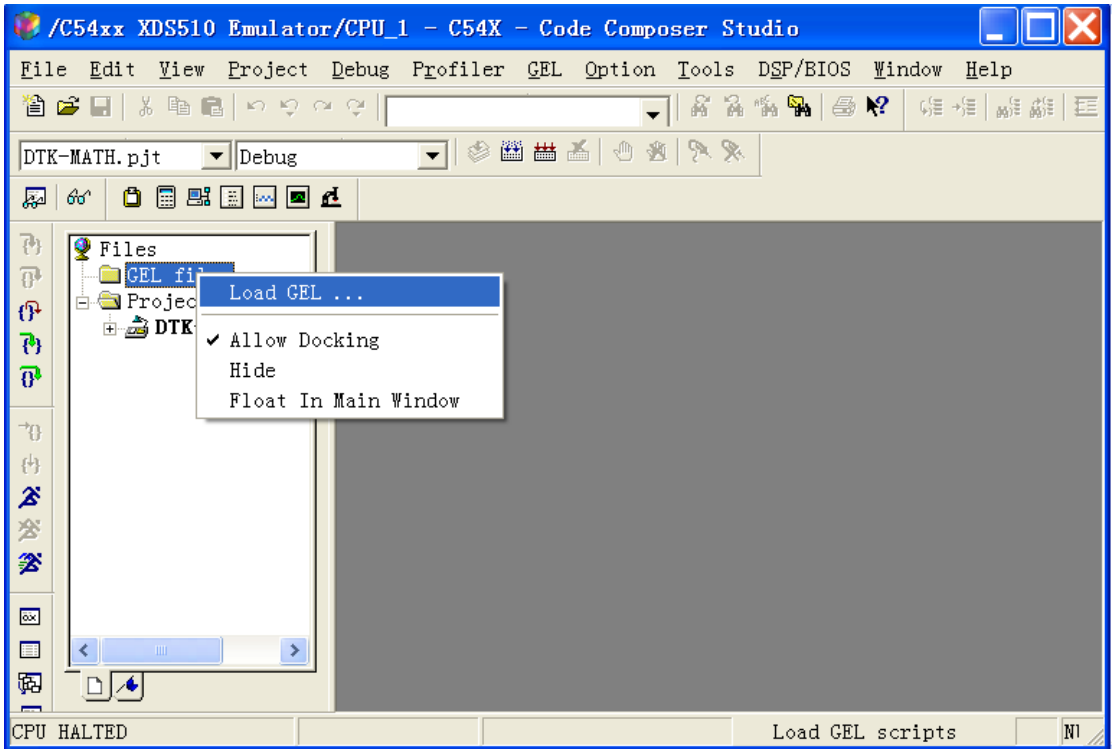
首先将光盘下 03. Examples of Program \ 04. SEED_DTK-DBD 调试实验程序目录下的 CCS-MATH 文件夹拷贝到 D：盘根目录下。

1. 将 DSP 仿真器与计算机连接好；
2. 将 DSP 仿真器的 JTAG 插头与 SEED-DEC5416 单元的 J1 相连接；
3. 启动计算机，当计算机启动后，打开 SEED-DTK_DBD 的电源。观察 SEED-DTK-IO1 单元的＋5V，＋3.3V，＋15V，－15V 的电源指示灯，SEED_DEC5416 的 D2 以及 SEED-DSK2812 的 D2 是否均亮；若有不亮的，请断开电源，检查电源。
4. 双击 

图标进入 CCS 环境。
5. 点击 Project →open 命令，在弹出的对话框中添加 DTK-MATH.pjt 文件；



6. 点击在工程视窗中右键 GEL file，在弹出的菜单中选择 Load GEL，在弹出的对话框中添加 dtk-boot.gel 文件。



7. 使用 projiect→Bulid 命令编译当前程序。使用 projiect→Bulid all 命令编译整个工程程序。

注意：

在这个实验中，为了加深对 CCS 的了解，分别在编译与链接过程中设置了的错误行。这些错误行都是在程序调试中经常遇到的。

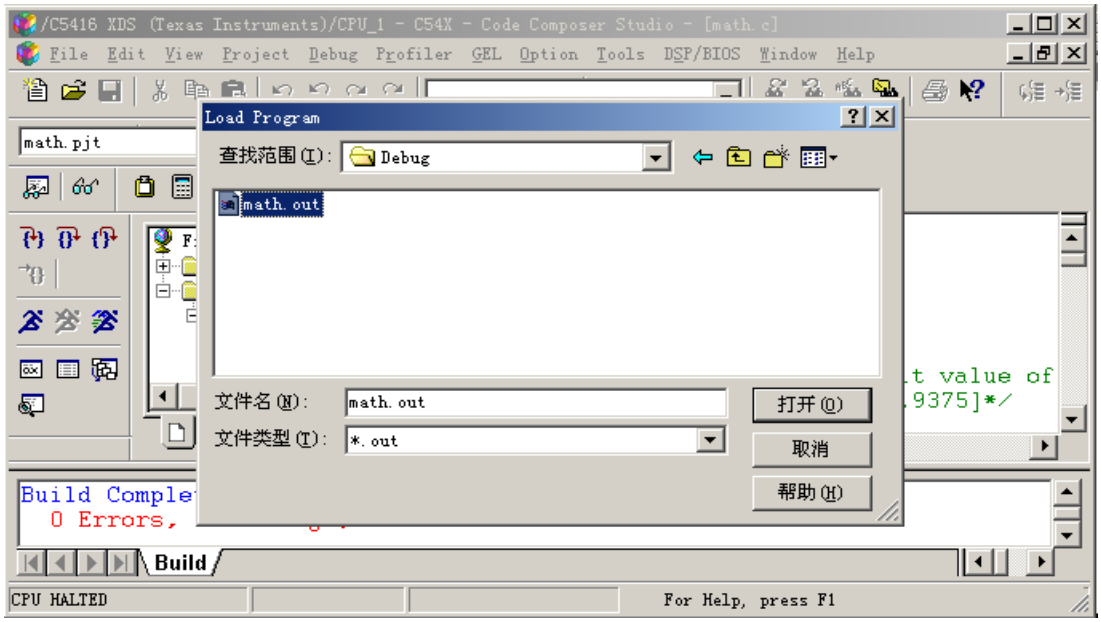
- 源程序错误：
  - 在函数 fixed_add（）中的 z 的定义未加“；”号
  - 函数 float_add（）的｛｝号缺右边而未完整
- 链接错误；
  - DSP 的空间分配重叠

    VECS:　　origin = 4B00h, length = 0120h 改为

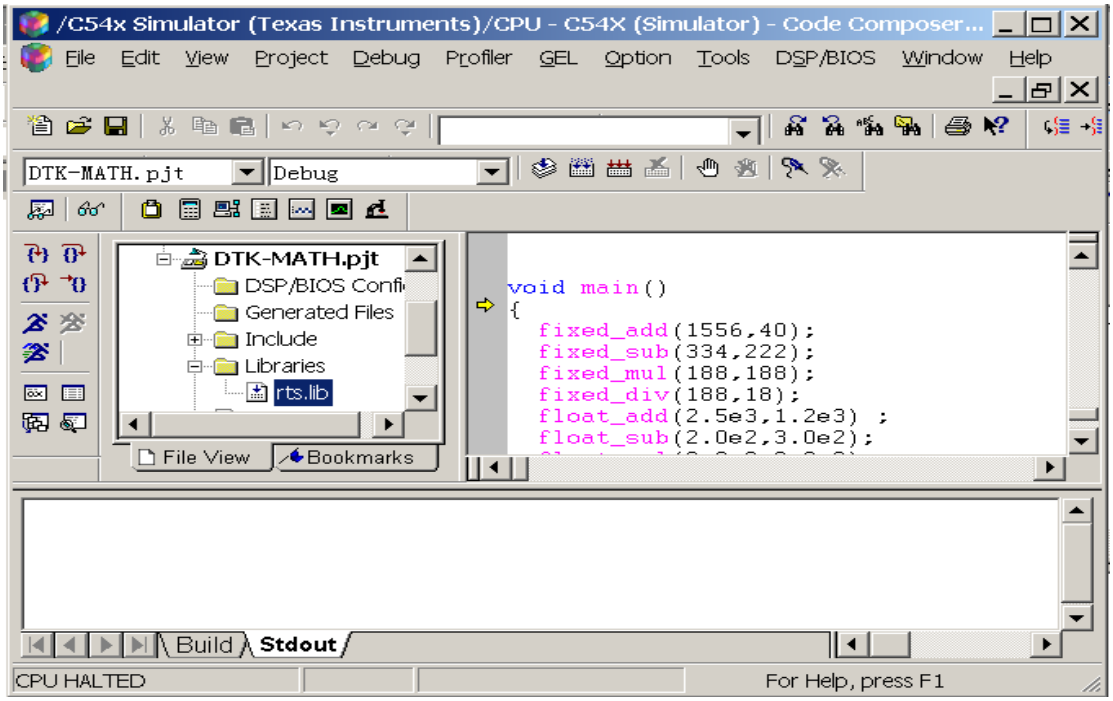
    VECS:　　origin = 4B00h, length = 0100h

在进行此实验时，只有将上述的程序错误改正后才能正确的编译与链接。产生 DTK_DBD_MATH.out。

8. 按照下图所示添加.out 文件，即使用 File→Load Program 菜单命令。.out 文件一般存放在 math 文件下的 debug 文件夹中。

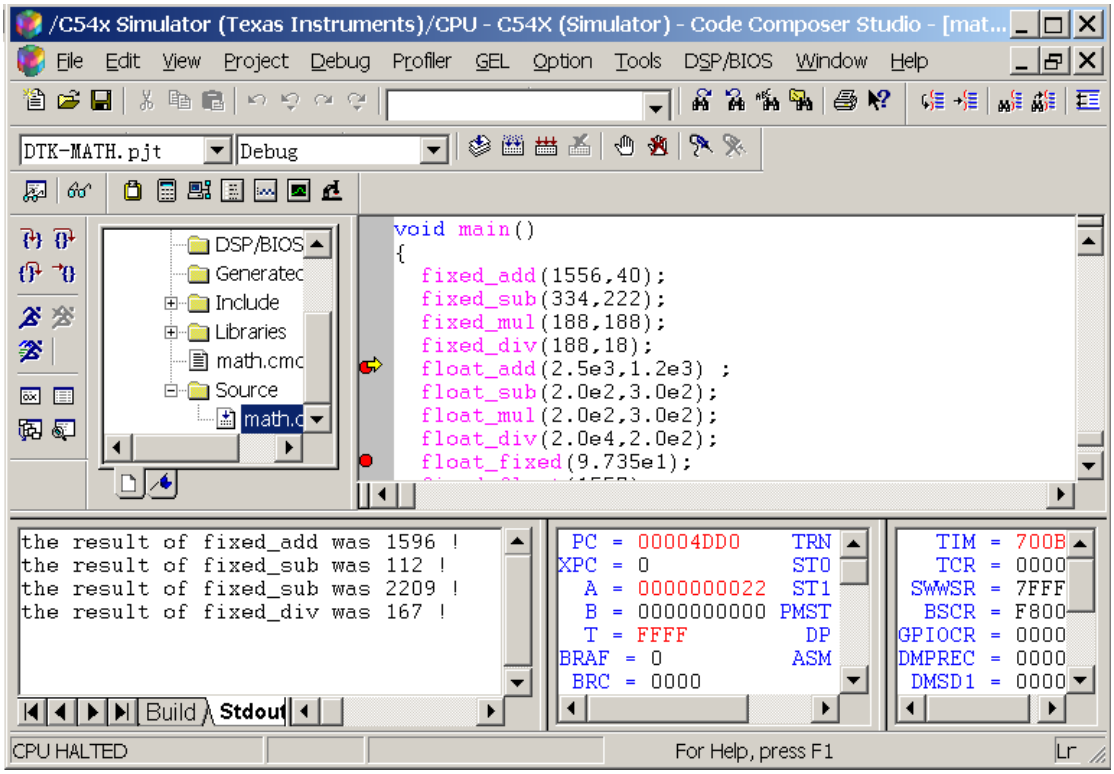


9. 点击 debug →Go Main 回到 C 程序的入口 main( )函数处。



10. 使用 F5 快捷键，运行程序，在 Stdout 观察窗中查看程序运行结果。

运行程序到第一个断点在 STDout 观察窗内看定点运算是否正确。然后再运行程序到第二个断点，观察浮点运算是否正确；再运行程序，观察浮点与定点之间的转换是否正确。



2 在片外设的使用（定时器、串行口）

2.1 实验目的

1. 了解 DSP 汇编程序的构成；
2. 了解 DSP 程序各段的含义；
3. 熟悉在汇编条件下如何编写中断服务程序；
4. 了解 DSP 的 McBSP 的工作原理和使用方法；
5. 掌握长时间间隔的定时器的处理；
6. 掌握片内外设的设置方法。

2.2 实验内容

1. DSP 的初始设置；
2. DSP 中断向量表的建立；
3. 定时中断的编写；
4. UART 的初始化；
5. MCBSP 的初始化设置；
6. MCBSP 的发送；
7. MCBSP 的接收；

2.3 实验背景知识

2.3.1 通用 TIMER 简介

TMS320VC5416 的定时器的说明：

VC5416 中有一个可编程的片上定时器，总共包含有三个可由用户设置的寄存器，并可以申请主机的中断。这三个寄存器分别为 TIM、PRD、TCR。这些寄存器与对应的存贮空间地址如下表所示：

Timer 0 Address	Timer 1 Address ('5402 only)	Register	Description
0024h	0030h	TIM	Timer register
0025h	0031h	PRD	Timer period register
0026h	0032h	TCR	Timer control register

时间寄存器（TIM）是一个 16 位的存贮器映射寄存器，它的值由周期寄存器来进行装载，并且做减一操作。

周期寄存器（PRD）是一个 16 位的存贮器映射寄存器，它是用来重装时间寄存器（TIM）寄存器的值的。

定时器控制寄存器（TCR）是一个 16 位的存贮器映射寄存器，包含了定时器的控制与状态信息。

2.3.2 McBSP 介绍

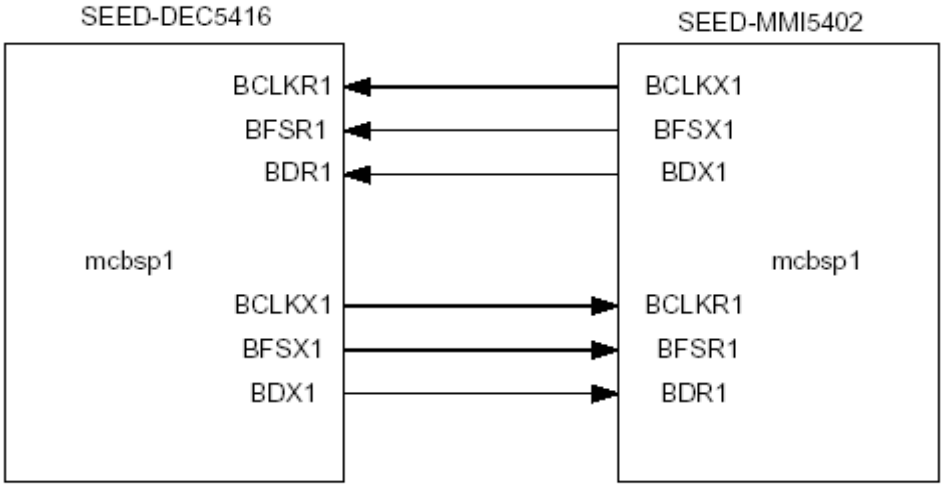
MCBSP 是 DSP 的片上外设资源。它可以与其它的 DSP、CODEC 和带有 SPI 接口的器件进行连接。在 TMS320VC5416 上共有 3 个 MCBSP（Multichannel Buffered Serial Port）。它共有三组主个管脚，包含了数据通路与控制通路。

注：CLKS 在 C5000 系列的 DSP 中没有提供，只在 C6000 中才有支持。

1. 串行同步通信的信号：FSR、CLKR、DR 和 FSX、CLKX、DX
   - 帧同步信号：FSR、FSX
   - 位-时钟：CLKR、CLKX
   - 串行数据流：DR、DX
1. 串行同步串行通信协议：
   1) 串行数据流起始时刻称为帧同步事件。帧同步事件由位-时钟采样帧同步信号给出。
   2) 串行数据流长度：串行传输的数据流位数达到设定的长度后，结束本次传输，等下一个帧同步信号达到，再发起另一次串行传输。
   3) 串行数据流传输速度：即每一个串行位的持续时间，由位-时钟决定
   4) FSR（FSX）、CLKR（CLKX）、DR（DX）三者之间的关系即如何取得帧同步事件、何时采样串行数据位流、或何时输出串行数据位流，是可以通过 MCBSP 的寄存器进行配置的。

      其中 SPCR1x 后面的寄存器是二次寻址的，其过程如下：

      首先向 SPSAx 寄存器中写入你想要操作的寄存器的子地址，然后再向 SPSDx 中写入你想要的数据，从而完成对其的操作。
2. 同步串口实验中使用 MCBSP1 作为与 SEED-MMI5402 通讯的同步串口其连接图如下：



3. 在同步串口实验中,，并而将 MCBSP1 设置成为单通道的方式使用。其设置如下：
   1) 设置 SPCR1 寄存器,禁止 SPI 模式；
   2) 设置 XCR1 寄存器，单数据相，发送数据长度为 16 位,每相 1 个数据；
   3) 设置 XCR2 寄存器，发送数据延时一个位；
   4) 设置 RCR1 寄存器，单数据相，接受数据长度为 16 位,每相 1 个数据；

5） 设置 RCR2 寄存器，单数据相,接收延时一个位；
6） 设置 PCR 寄存器，设置 BLCKR 为输入，下降沿接收数据；设置 BFSR 为输入，并且其极性为高有效；设置 BCLKX 由内部时钟产生，并且上沿发送数据；
7） 设置 SRGR1，确定分频数为 0x0FF，MCBSP2 的波待率为 625k，帧脉冲宽度为 1 个数据位，0x0F：
8） 设置 SRGR2，确定时钟来源为内部的 CPU，确定帧同步为低有效；

2.4 实验要求

能够掌据汇编语言的程序结构。能够按照设置的定时时间进行定时输出，能进入定时中断;通过串行通信发送单个字符和连续的多个字符,能进入串行中断。

2.5 实验程序功能与结构说明

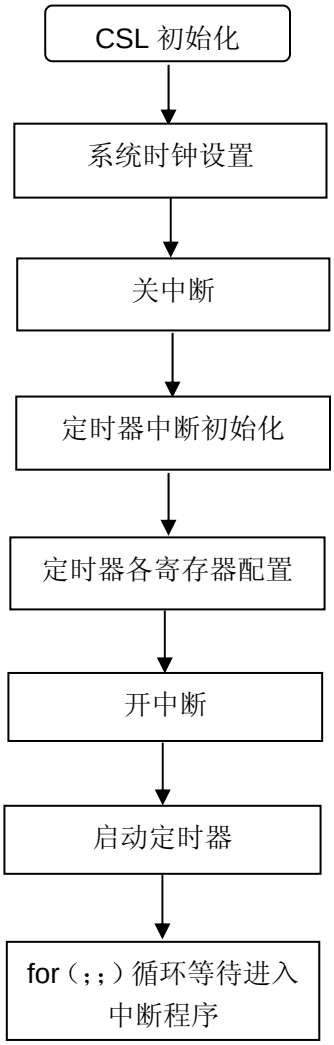
2.5.1 Timer 实验调试程序

2.5.1.1 在 Timer 实验调试程序中，主要包含以下文件：

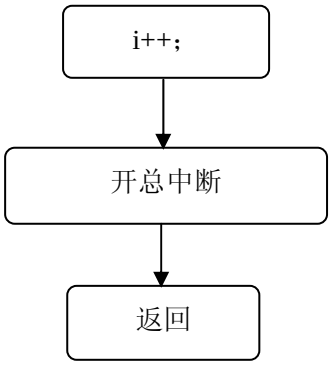
1. timer.c：这是实验的主程序，主要包含 CSL 初始化，DSP 初始化，Timer 及 Timer 中断初始化，Timer 中断程序。
2. vector.asm：包含 5416 的中断向量表。
3. dec5416.cmd：声明了系统的存贮器配置与程序各段的链接关系。

2.5.1.2 程序流程图（定时器）

主程序流程图

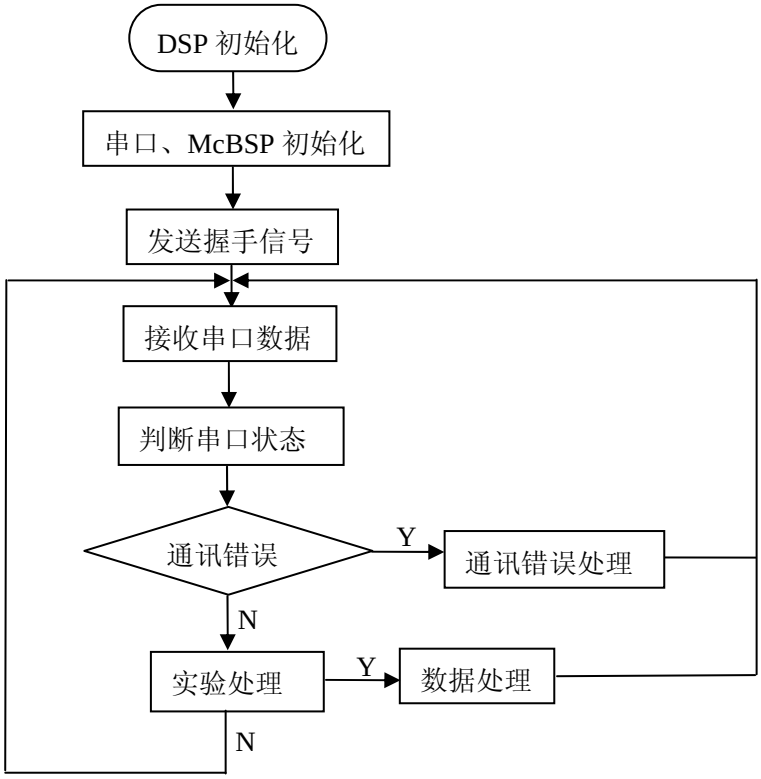


中断程序流程图

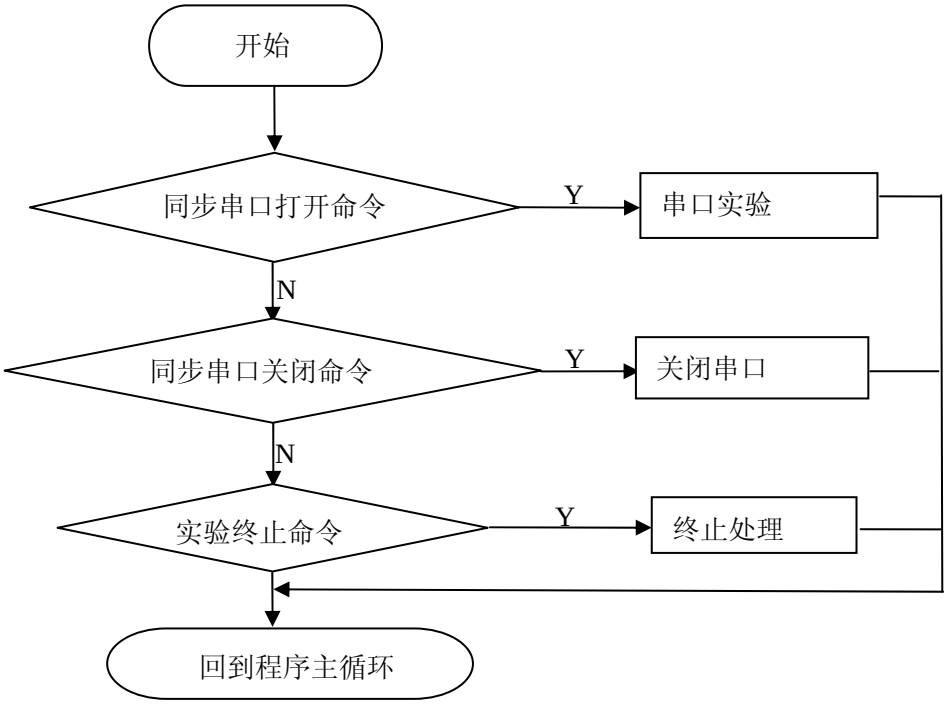


2.5.1.3　程序流程图（串行口）

主程序流程图



实验处理流程图



2.5.1.4 实验步骤（定时器）

首先将光盘下 03. Examples of Program \ 04. SEED_DTK-DBD 调试实验程序目录下的 CCS-Timer 的文件夹拷贝到 D：盘根目录下。

1. 将 DSP 仿真器与计算机连接好；
2. 将 DSP 仿真器的 JTAG 插头与 SEED-DEC5416 单元的 J1 相连接；
3. 启动计算机，当计算机启动后，打开 SEED-DTK_DBD 的电源。观察 SEED-DTK-IO1 单元的＋5V，＋3.3V，＋15V，－15V 的电源指示灯，SEED_DEC5416 的 D2 以及 SEED-DSK2812 的 D2 是否均亮；若有不亮的，请断开电源，检查电源。
4. 打开 CCS，进入 CCS 的操作环境。
5. 装入 timer.pjt 工程文件，添加 dtk-boot.gel 文件。
6. 装载程序 timer.out，进行调试。
7. 在程序的第 101 行“i=i+1;”处设置断点。
8. 运行程序，程序会停在断点处，表明已进入定时器中断。继续运行程序，程序每次都会停在断点处。实验者可根据自己的需要改变周期寄存器的值，从而控制每次进中断的时间。

2.5.1.5 实验步骤（串行口）

首先将光盘下 03. Examples of Program \ 04. SEED_DTK-DBD 调试实验程序目录下的 DTK _MCBSP 的文件夹拷贝到 D：盘根目录下。

1.将 DSP 仿真器与计算机连接好。

2.将 DSP 仿真器的 JTAG 插头与 SEED-DEC5416 单元的 J1 相连接。

3.启动计算机，当计算机启动后，打开 SEED-DTK_DBD 的电源。观察 SEED-DTK-IO1 单元的＋5V，＋3.3V，＋15V，－15V 的电源指示灯，SEED_DEC5416 的 D2 以及 SEED-DSK2812 的 D2 是否均亮；若有不亮的，请断开电源，检查电源。

4.进入 SEED-DTK_DBD 的“实验选项”的菜单下，选择“同步串口实验”一项，进入后光标停留在“使用 CCS 选项”处，按 “Enter”键，通过“→”选择是否使用 CCS。在调试时选择使用 CCS。

5.打开 CCS，进入 CCS 的操作环境。

6.装入 DTK _MCBSP.pjt 工程文件，添加 Dtk-boot.gel 文件，进行调试。

7.装载程序 DTK _MCBSP.out。

8.运行程序，此时 LCD 显示器的下方将出现“同步串口实验装载成功！”

9.打开 MCBSP.c 文件，在第 202 行 if(MCBSP_RRDY(mcbsp1))”处设置断点。

10.在“发送模式”一项任意选择“整帧”或“单个”，在“串口状态”一项选择“开启”，这是程序将停在所设的断点处。

11.继续运行程序。

12.若“发送模式”选择的是“整帧”，则在“自动发送数据”选项选择任意字符串，此数据串将在 LCD 显示器最下方的“数据接受：”后出现，说明通讯成功。若“发送模式”选择的是“单个”，则在 LCD 显示器下方的键盘上输入任意数字，此数字将在 LCD 显示器最下方的“数据接受：”后出现，说明通讯成功。

注意：如果操作失误或者发现对实验箱的操作已不起作用，请复位实验箱，这时一定要在 CCS 下重新复位 CPU,再重复步骤 6)-步骤 12)。

2.6 思考题

理解 DSP 程序的编写过程，如何编写中断程序？

3 信号处理实验：数字滤波器（FIR 、IIR）

第一部分 FIR滤波器的设计

3.1 实验目的

(1) 了解 FIR、IIR 滤波器的原理及使用方法；
(2) 了解使用 Matlab 语言设计 FIR、II 滤波器的方法；
(3) 了解 DSP 对 FIR、II 滤波器的设计及编程方法；
(4) 熟悉对 FIR、II 滤波器的调试方法；

3.2 实验内容

本试验要求设计滤波器采样频率为 1000hz，截止频率 200hz 的高通滤波器。设计 IIR 滤波器实现上面要求。

输入信号频率为 50HZ 和 400HZ 的合成信号，目的是通过我们设计的滤波器将 50HZ 的信号滤掉，余下 400HZ 的信号成分，达到滤波的效果。

3.3 实验原理

数字滤波器的输入 x[k]和输出 y[k]之间的关系可以用如下常系数线性差分方程及其 z 变换描述：

$$y[k]=\sum_{i=0}^{N} a_i x[k-i]+\sum_{i=1}^{M} b_i y[k-i]$$

系统的转移函数为：

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{k=1}^{M} b_k z^{-k}}{1-\sum_{k=0}^{N} a_k z^{-k}}$$

设 N=M，则传输函数变为：

$$H(z)=\frac{a_0+a_1 z^{-1}+\cdots+a_n z^{-N}}{1+b_1 z^{-1}+\cdots+b_n z^{-N}}$$

转换成极零点表示为：

$$H(z)=C\prod_{j=1}^{N}\frac{z-z_j}{z-p_j}$$

式中，z_j 表示零点，p_j 表示极点，它具有 N 个零点和 N 个极点,如果任何一个极点在 Z 平面单位圆外，则系统不稳定。如果系数 b_j 全部为 0，滤波器成为非递归的 FIR 滤波器，这时系统没有极点，因此 FIR 滤波器总是稳定的。对于 IIR 滤波器，有系数量化敏感的缺点。由于系统 对序列施加的算法，是由加法、延时和常系数乘三种基本运算的组合，所以可以用不同结构的数字滤波器来实现而不影响系统总的传输函数。图 3.1 是四阶直接型 IIR 滤波器的结构。

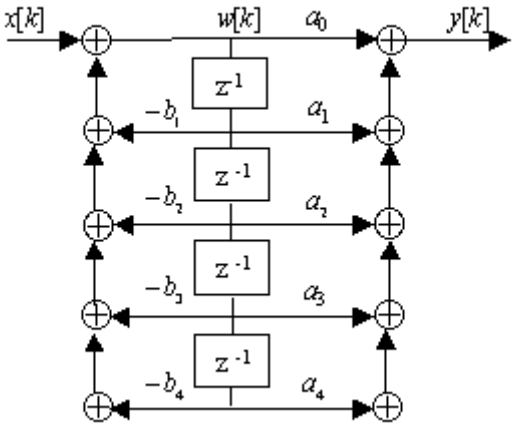


图 3.1 四阶直接型 IIR 滤波器的结构

3.4 实验程序设计

1. 滤波器的 Matlab 语言设计

编写 matlab 程序，生成 FIR 滤波器系数后，附到 DSP 汇编语言程序中。

```
fs=1000;fs2=fs/2;
 f1=50;f2=400;
 pi=3.141592653;
 i=0:1:255
 x=sin(2*pi*f1*i/fs)+sin(2*pi*f2*i/fs)
 X=fft(x,512);
```

```
 P1=X.*conj(X)/512;
subplot(2,1,1);plot(P1(1:256))


 wp=100/500;ws=300/500;
  %[n,wn]=buttord(wp,ws,1,40);%this is low pass filter
%[b,a]=butter(n,wn);
 [n,wn]=buttord(ws,wp,1,40);% this is for high pass filter
 [b,a]=butter(n,wn,'high');
 subplot(2,2,2);freqz(b,a,512,1000)
 y=filter(b,a,x);
 Y=fft(y,512);
 P=Y.*conj(Y)/512;
 %f=1000*(0:255)/512;
subplot(2,1,2);
 plot(P(1:256))
 n
 a=a*2^12
 b=b*2^12
 wn
```

下图是对应频率特性图：

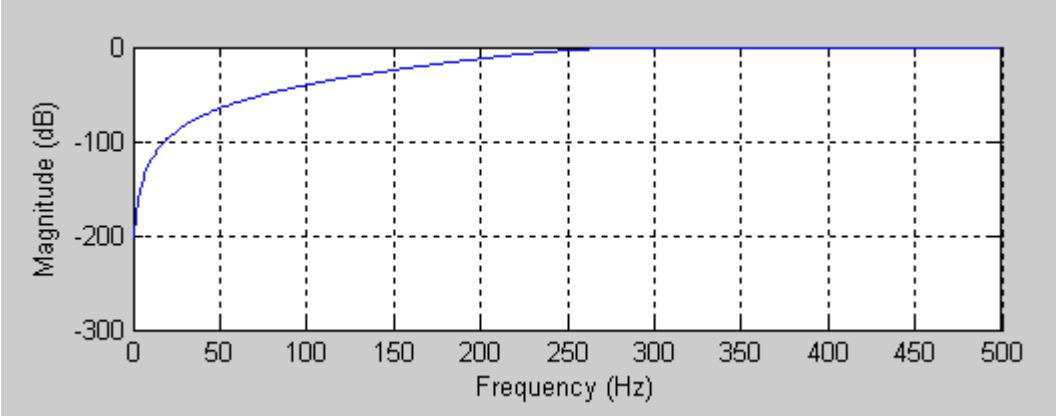


图 3.2　滤波器的频率特性图

下图是滤波后的功率谱图：

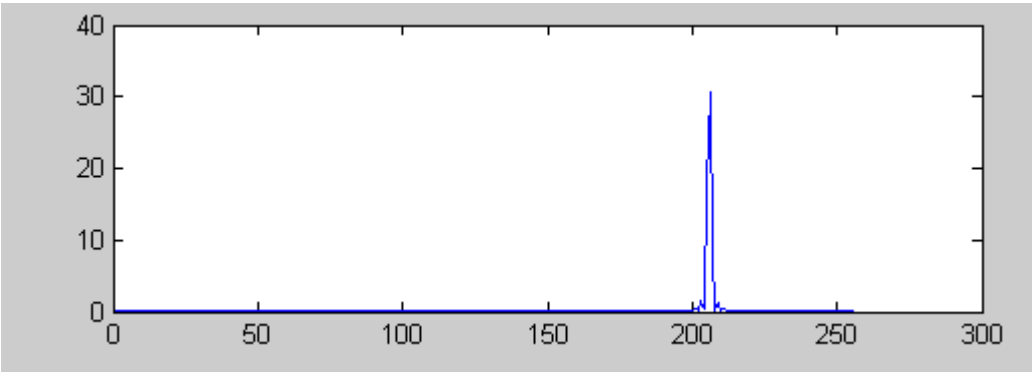


图 3.3　滤波后的信号功率谱图

2. 编写一输入信号程序，文件名为 firinput.c。输入数据为 50hz 和 400hz 的合成信号，采样率为 1000hz,256 个样点。

3. DSP 汇编语言程序设计。由汇编源文件 filter.asm,中断向量表 vectors.asm 和链接命令文件 filter.cmd 组成。

3.5　实验步骤

1. 打开 CCS，新建立一工程文件 iir.pjt。

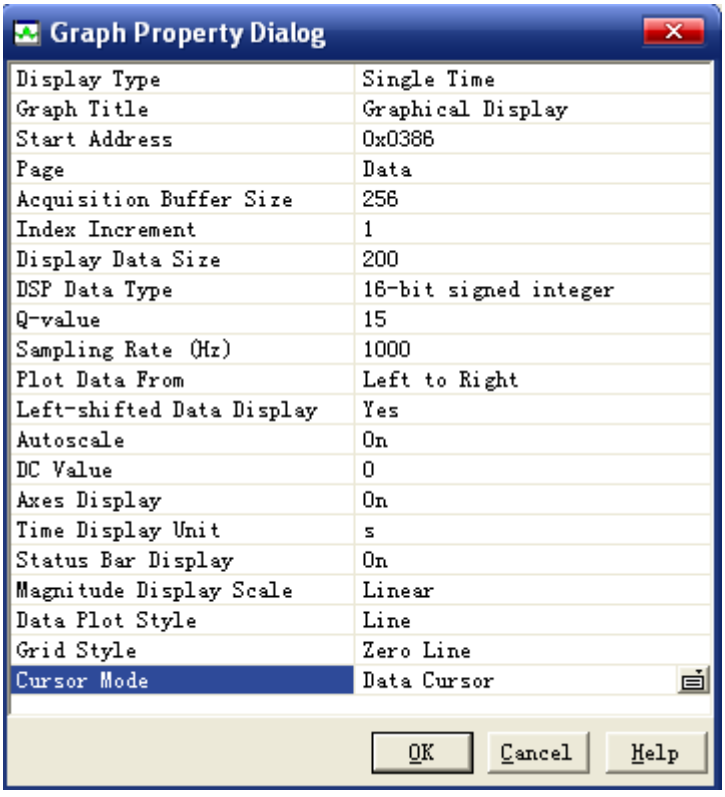
2. 将汇编源文件 filter.asm、中断向量表 vectors.asm 和链接命令文件 filter.cmd 添加到 iir.pjt 中。

3. 在 project 菜单下选择 build options 选项，选取 Linker 选项，调整为 -q -c -m".\Debug\filter.map" -o".\Debug\filter.out" -w -x。点击编译，链接图标，通过后生成 filter.out 文件和 iir.map 文件，其余选项可默认。

4. 在 file 菜单下，选择 load program 选项，将生成的 filter.out 文件装载到 DSP 中。

5. 运行程序，在 view 菜单下选择 watch window 选项来观测变量值。依次输入 inputdata 和 filterdata 来观测输入输出变量值，这两个变量分别为滤波前的输入数据和滤波后输出数据的首地址。

6. 可以在 view 菜单下选择 graph/time frequence，弹出如下对话框。



按照要求，设置好相应的参数，来观测输入和输出数据的波形。

7. 具体调试执行程序时，可使用断点，单步执行等方式。

3.6 程序运行结果

根据提供的配置文件，其中滤波前的数据首地址放在数据存储空间的地址 384H 处，滤波后的数据首地址放在数据存储空间的地址 264H。

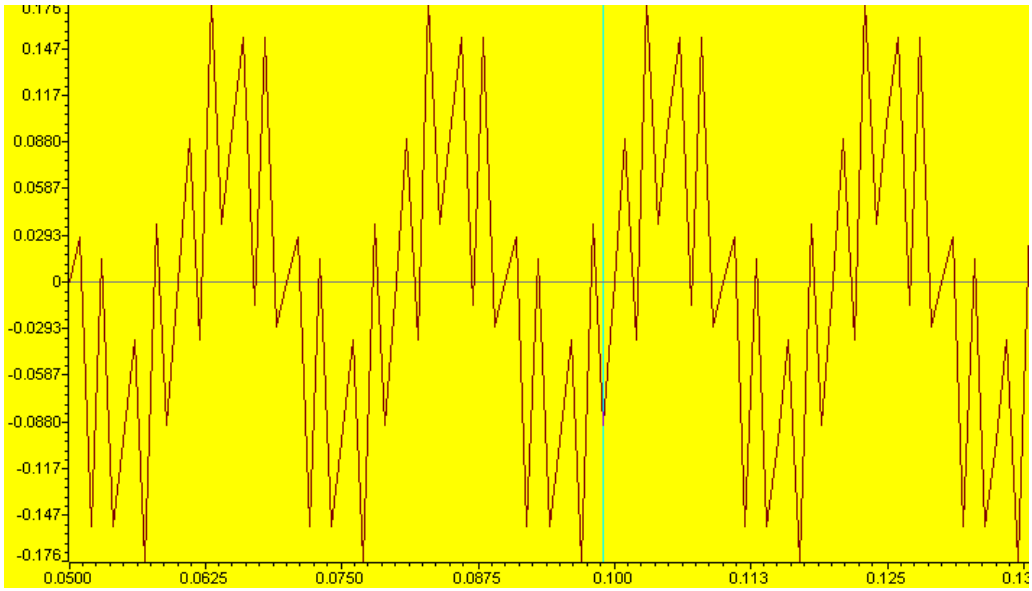


图.3.4 滤波前 CCS 中的数据时域波形

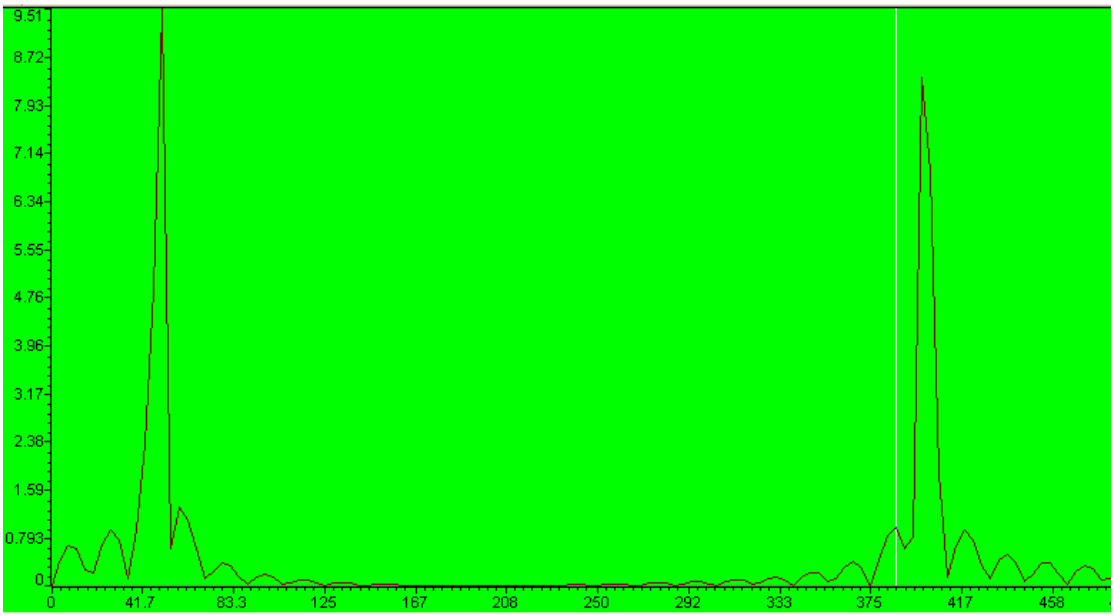


图 3.5 滤波前 CCS 中的数据频域波形

图 3.6 滤波后 CCS 中的数据时域波形

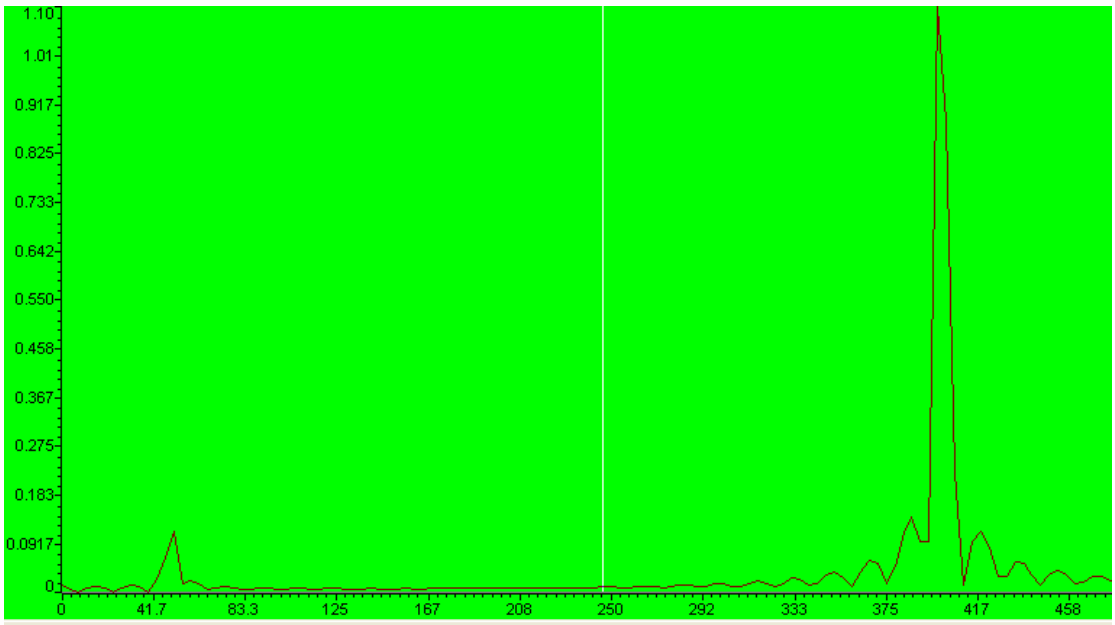


图 3.7 滤波后 CCS 中的数据频域波形

3.7 思考题

1. 为什么要对 matlab 程序生成的系数进行调整，即将浮点数转换成整数？

2. 试改变输入的信号（保证一个频率在通带范围内，一个在通带范围外），进行相应的数据调整，然后在 CCS 底下观测看输入数据波形。

3. 可进行滤波器系数的调整再进行相应滤波，然后在 CCS 底下看输出数据波形情况有何变化。

3 信号处理实验：数字滤波器（FIR 、IIR）

第二部分 IIR滤波器的设计

3.1 实验目的

（1）了解 FIR 滤波器的原理及使用方法；
（2）了解使用 Matlab 语言设计 FIRR 滤波器的方法；
（3）了解 DSP 对 FIR 滤波器的设计及编程方法；
（4）熟悉对 FIR 滤波器的调试方法；

3.2 实验内容

本试验要求设计滤波器采样频率为 1000hz，截止频率 300hz 的低通滤波器。设计 FIR 滤波器实现上面要求。

输入信号频率为 40HZ 和 480HZ 的合成信号，目的是通过我们设计的滤波器将 480HZ 的信号滤掉，余下 40HZ 的信号成分，达到滤波的效果。

3.3 实验原理

一个线性位移不变系统的输出序列 $y(n)$和输入序列 $x(n)$之间的关系，应满足常系数线性差分方程：

$$y(n)=\sum_{i=0}^{N-1}b_i x(n-i)-\sum_{i=1}^{M}a_i y(n-i) \qquad n\geq 0$$

$x(n)$: 输入序列，$y(n)$: 输出序列，ai、bi : 滤波器系数，　　N: 滤波器的阶数。

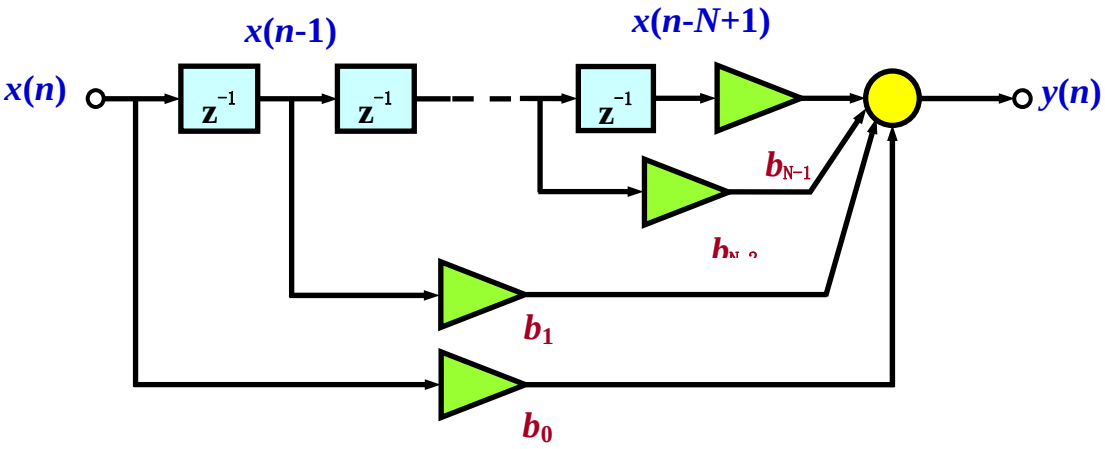
在式上式中，若所有的 ai 均为 0，则得 FIR 滤波器的差分方程：

$$y(n)=\sum_{i=0}^{N-1}b_i x(n-i)$$

对上式进行 z 变换，可得 FIR 滤波器的传递函数：

$$H(z)=\frac{Y(z)}{X(z)}=\sum_{i=0}^{N-1}b_i z^{-i}$$

FIR 滤波器的结构



FIR 滤波器的单位冲激响应 $h(n)$为有限长序列。

若 $h(n)$为实数，且满足偶对称或奇对称的条件，则 FIR 滤波器具有线性相位特性。

偶对称：$h(n)= h(N-1-n)$；

奇对称：$h(n)= -h(N-1-n)$。

偶对称线性相位 FIR 滤波器的差分方程：

$$y(n)=\sum_{i=0}^{\frac{N}{2}-1} b_i[x(n-i)+x(n-N+1+i)]$$

N——偶数

在数字滤波器中，FIR 滤波器具有如下几个主要特点：

① FIR 滤波器无反馈回路，是一种无条件稳定系统；

② FIR 滤波器可以设计成具有线性相位特性。

本实验程序设计的就是一种偶对称的线性相位滤波器。

程序算法实现采用循环缓冲区法。

算法原理：

① 在数据存储器中开辟一个 N 个单元的缓冲区(滑窗)，用来存放最新的 N 个输入样本；

② 从最新样本开始取数；

③ 读完最老样本后，输入最新样本来代替最老样本，而其他数据位置不变；

④ 用 BK 寄存器对缓冲区进行间接寻址，使缓冲区地址首尾相邻。

3.4 实验程序

1. 滤波器的 Matlab 语言设计

编写 matlab 程序，生成 FIR 滤波器系数后，附到 DSP 汇编语言程序中。

Matlab 程序如下：

```
f=[0 0.6 0.6 1];
m=[1 1 0 0]
b=firls(36,f,m)
freqz(b,1,512)
b=b*2^15
```

频率响应图如下：

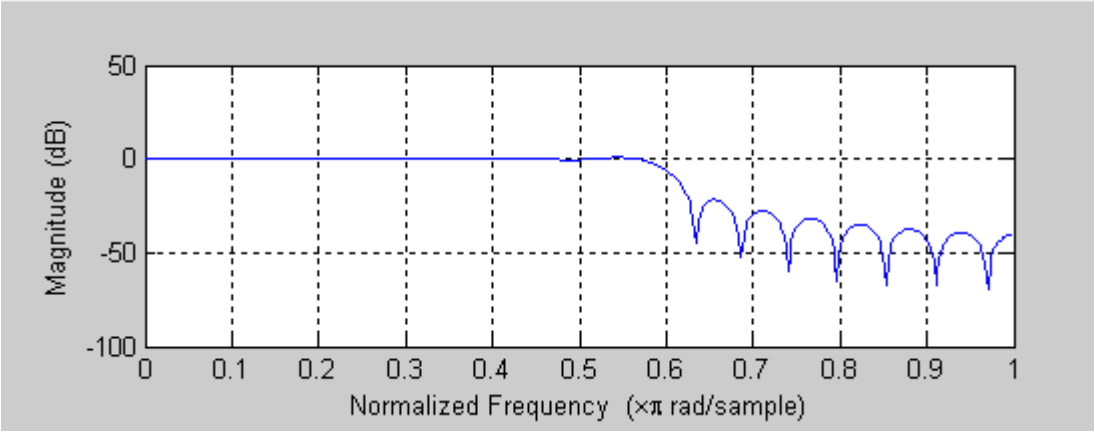


图 1　滤波器的频率特性图

相位特性图如下：

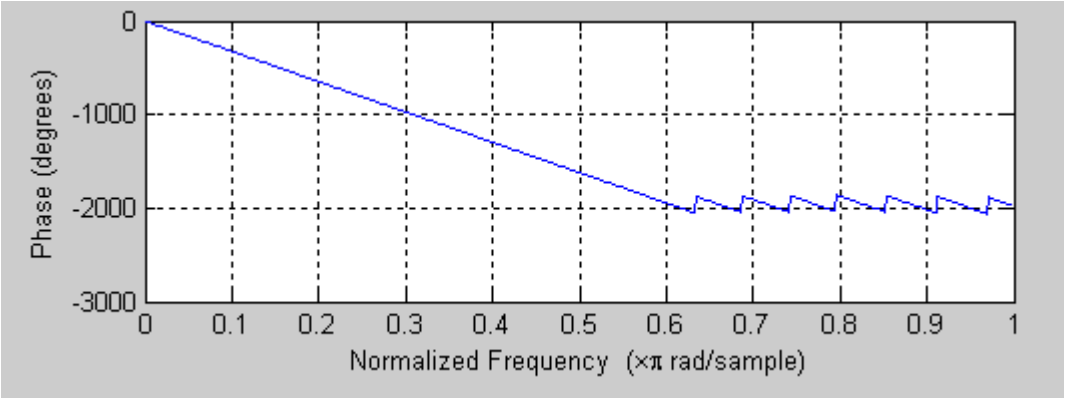


图 2　滤波器的相位特性图

2. 编写一输入信号程序，文件名为 firinput.c。输入数据为 40hz 和 480hz 的合成信号，采样率为 1000hz,256 个样点。

3. DSP 汇编语言程序设计。由汇编源文件 fir.asm,中断向量表 vectoes.asm 和链接命令文件 fir.cmd 组成。

3.5　实验步骤

1. 打开 CCS，新建立一工程文件 fir.pjt。

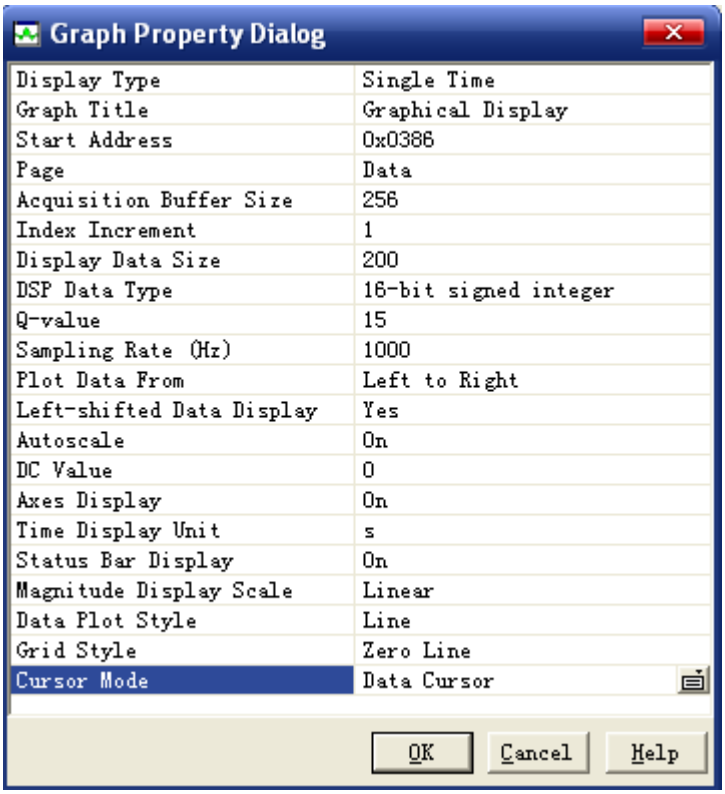
2.将汇编源文件 fir.asm、中断向量表 vectors.asm 和链接命令文件 fir.cmd 添加到 fir.pjt 中。

3. 在 project 菜单下选择 build options 选项，选取 Linker 选项，调整为 -q -c -m".\Debug\fir.map" -o".\Debug\fir.out" -w -x。点击编译，链接图标，通过后生成 fir.out 文件和 firr.map 文件，其余选项可默认。

4. 在 file 菜单下，选择 load program 选项，将生成的 fir.out 文件装载到 DSP 中。

5. 运行程序，在 view 菜单下选择 watch window 选项来观测变量值。依次输入 input 和 output 来观测输入输出变量值，这两个变量分别为滤波前的输入数据和滤波后输出数据的首地址。

6. 可以在 view 菜单下选择 graph/time frequence，弹出如下对话框。



按照要求，设置好相应的参数，来观测输入和输出数据的波形。

7．具体调试执行程序时，可使用断点，单步执行等方式。

3.6 实验结果

程序运行结果如下：

根据提供的配置文件，其中滤波前的数据首地址放在数据存储空间的地址 60H 处，滤波后的数据首地址放在数据存储空间的地址 4000H。

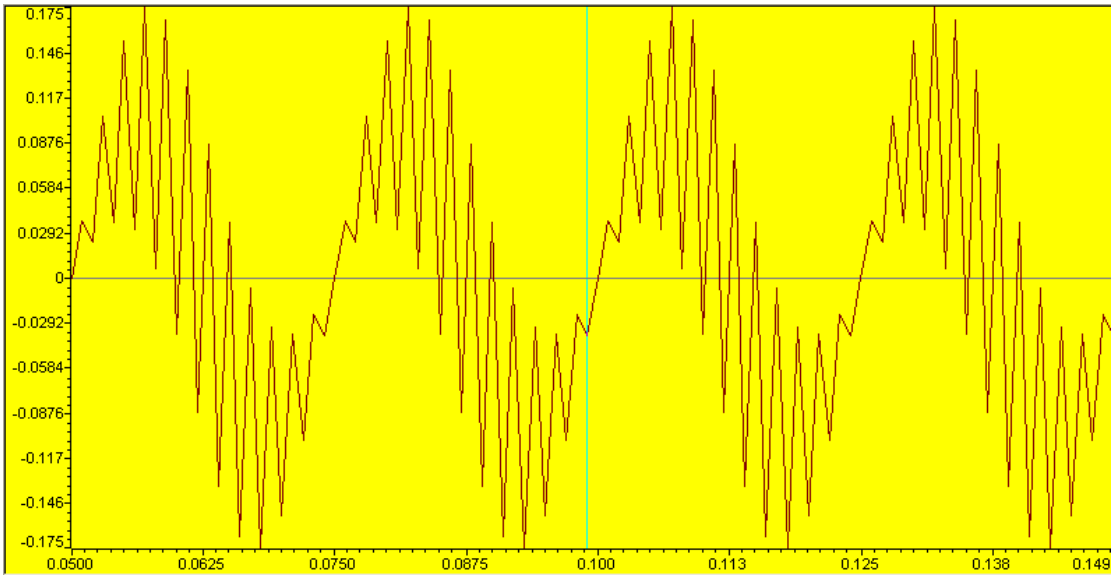


图 3 滤波前 CCS 中的数据时域波形

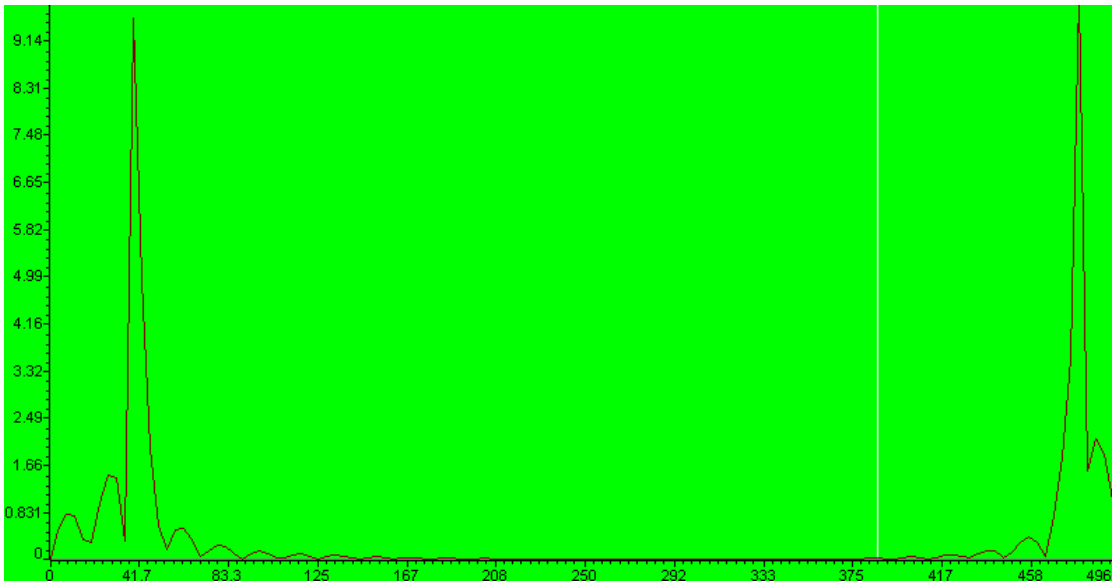


图 4 滤波前 CCS 中的数据频域波形

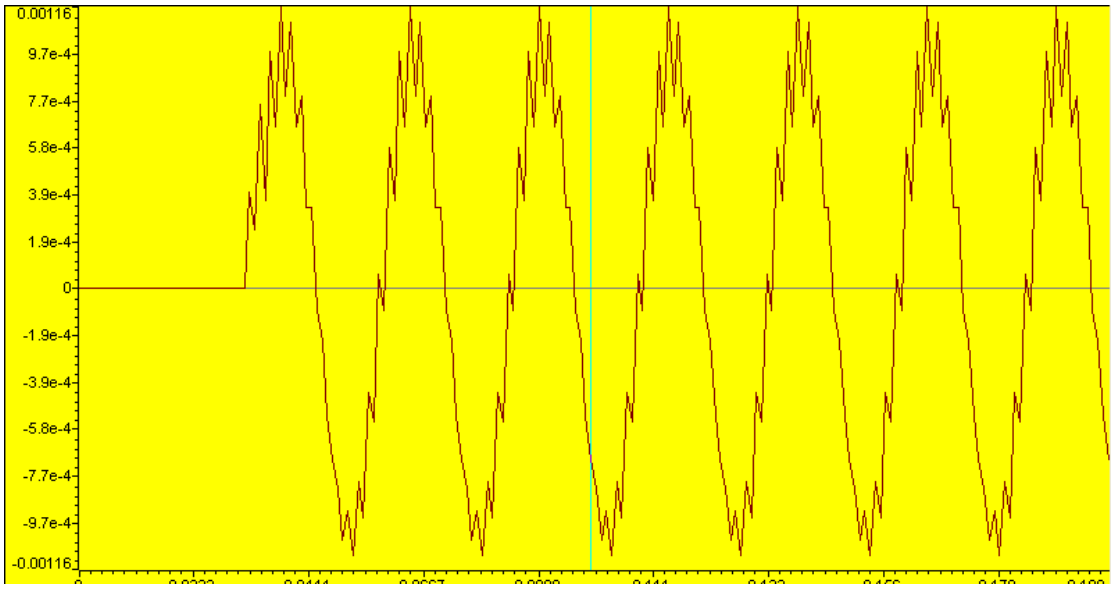


图 5 滤波后 CCS 中的数据时域波形

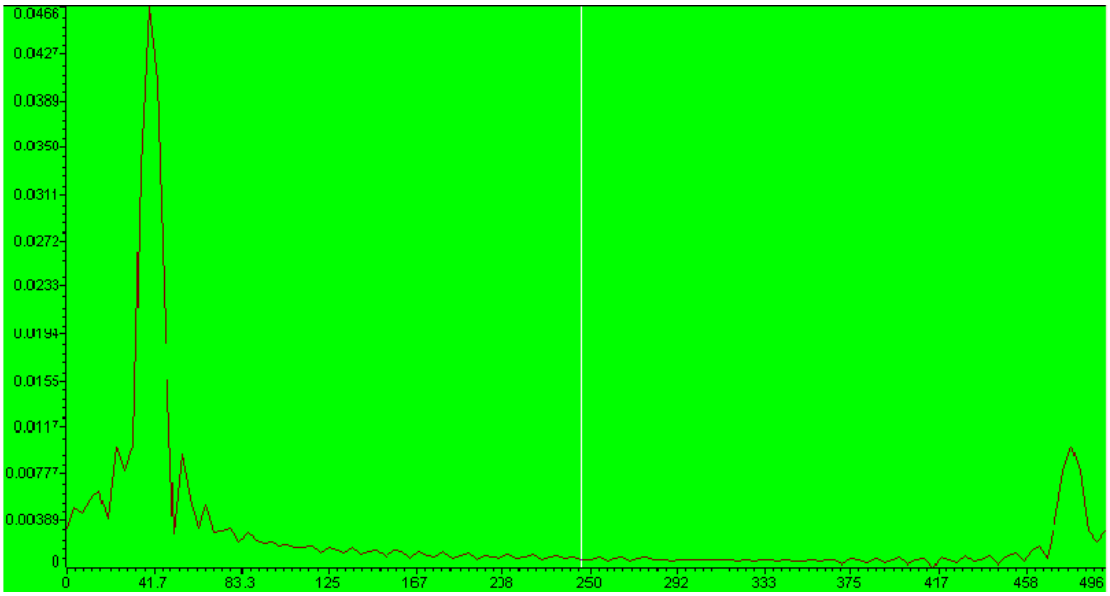


图 6 滤波后 CCS 中的数据频域波形

3.7 思考题

1. 为什么要对 matlab 程序生成的系数进行调整，即将浮点数转换成整数？
2. 试改变输入的信号（保证一个频率在通带范围内，一个在通带范围外），进行相应的数据调整，然后在 CCS 底下观测看输入数据波形。
3. 可进行滤波器系数的调整再进行相应滤波，然后在 CCS 底下看输出数据波形情况有何变化。

4 综合实验（语音数据采集、处理）

4.1. 实验目的：

1. 了解CODEC 工作的基本原理，了解编码与解码的过程；
2. 理解DSP 的MCBSP 的工作原理，了解SPI 方式；
3. 熟悉DSP 与CODEC（TLV320AIC23B）的控制与数据传输的过程。
4. 熟悉宏函数实现对MCBSP 的设置；
5. 熟悉FIR、IIR滤波器的原理及使用方法；
6. 掌握使用Matlab语言设计FIR、IIR滤波器的方法；
7. 掌握利用DSP实现数字滤波器的方法；

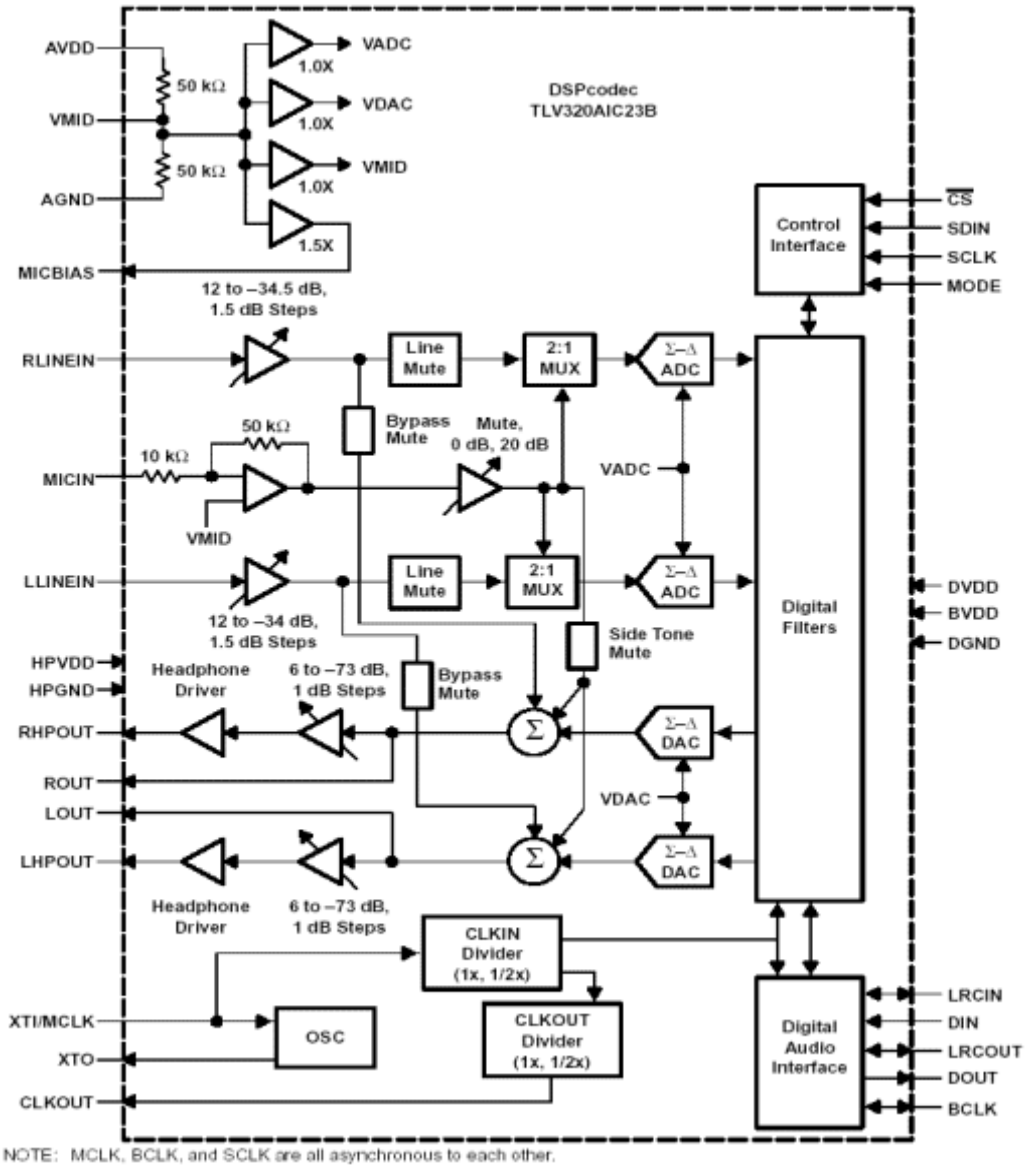
4.2. 实验内容：

1. DSP 的初始设置；
2. DSP 中断项量表的建立；
3. 异步通讯；
4. CODEC 的输入与输出；
5. 对音频信号进行数字滤波的方法

4.3. 实验背景知识：

LV320AIC23B 的介绍：

TLV320AIC23B（以下简称AIC23）是TI 推出的一款高性能的立体声音频Codec 芯片，内置耳机输出放大器，支持MIC 和LINE IN 两种输入方式（二选一），且对输入和输出都具有可编程增益调节。AIC23 的模数转换（ADCs）和数模转换（DACs）部件高度集成在芯片内部，采用了先进的Sigma-delta 过采样技术，可以在8K 到96K 的频率范围内提供16bit、20bit、24bit 和32bit 的采样，ADC 和DAC 的输出信噪比分别可以达到90dB 和100dB。与此同时，AIC23 还具有很低的能耗，回放模式下功率仅为23mW，省电模式下更是小于15uW。AIC23 的管脚和内部结构框图如下：



1． 音频接口：

主要连接为 ：

BCLK： 数字音频接口时钟信号（bit 时钟），AIC23 工作在主模式，该时钟由AIC23 产生；

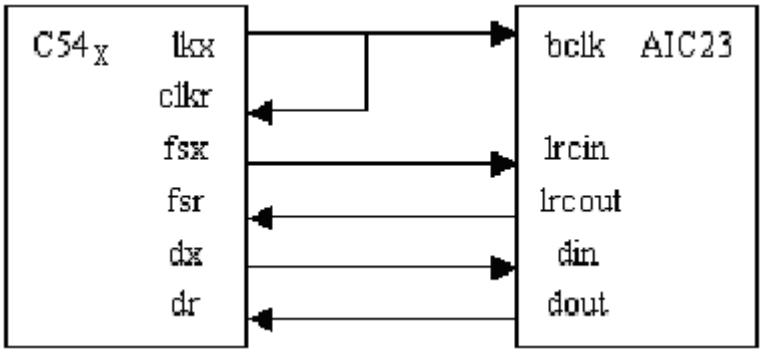
LRCIN： 数字音频接口DAC 方向的帧信号；

LRCOUT： 数字音频接口ADC 方向的帧信号；

DIN： 数字音频接口DAC 方向的数据输入；

DOUT： 数字音频接口ADC 方向的数据输出；

这部分可以和DSP 的McBSP（Multi-channel buffered serial port，多通道缓存串口）无缝连接，唯一要注意的地方是McBSP 的接收时钟和AIC23 的BCLK 都由AIC23来提供，连接示意图如下：



2. 配置接口：

主要管脚为

SDIN： 配置数据输入

SCLK： 配置时钟

DSP 通过该部分配置AIC23 的内部寄存器，每个word 的前7bit 为寄存器地址，后9bit 为寄存器内容。具体方法和寄存器具体内容见后。

3. 其他：

主要管脚为

MCLK： 芯片时钟输入(12M)

VMID： 半压输入，通常由一个10U 和一个0.1U 电容并联接地

MODE： 芯片工作模式选择，Master CS－片选信号（配置时有效）

CLKOUT： 时钟输出，可以为MCLK 或者MCLK/2（详见寄存器配置）

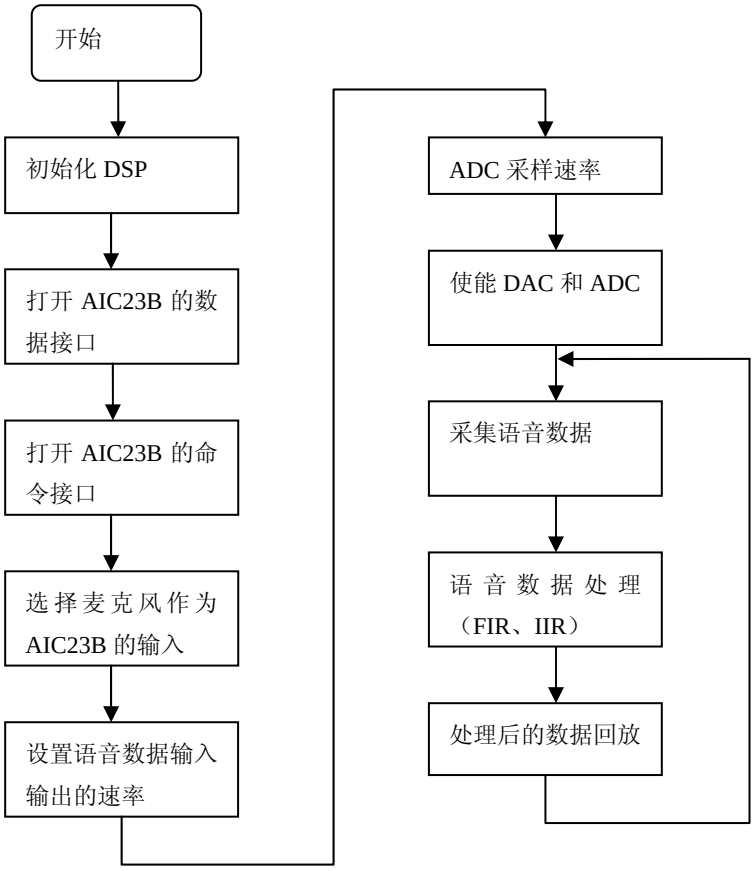
4. 与AIC23 配置接口的连接

AIC23 的配置采用SPI 模式。通常比较简单的办法是利用DSP 的一个McBSP 用SPI 模式跟AIC23 连接。

4.4. 实验要求

通过音频输入与输出实验，了解对VC5416 的MCBSP 的设置；通过对音频信号进行数字滤波，掌握语音处理的基本方法。

4.5 实验的软件流程图



4.6 实验步骤：

1) 将DSP 仿真器与计算机连接好；
2) 将DSP 仿真器的JTAG 插头与SEED-DEC54xx 单元的J8 相连接；
3) 启动计算机，当计算机启动后，打开SEED-DTK5416 的电源。观察DTK-IO单元的＋5V、＋3.3V、＋15V、－15V 的电源指示灯是否均亮；若有不亮的，请断开电源，检查电源；
4)打开CCS，进入CCS 的操作环境；
5) 编写语音采集和数字滤波程序，并进行调试；
6）装载运行，通过VIEW-graph功能查看采集得到的原始波形和滤波以后的波形；
7）用耳机分别听原声和滤波以后的声音，分辨它们的区别。

4.7 实验结果对照

如果程序无误，操作正确的话，会得到如下的时域和频域图。

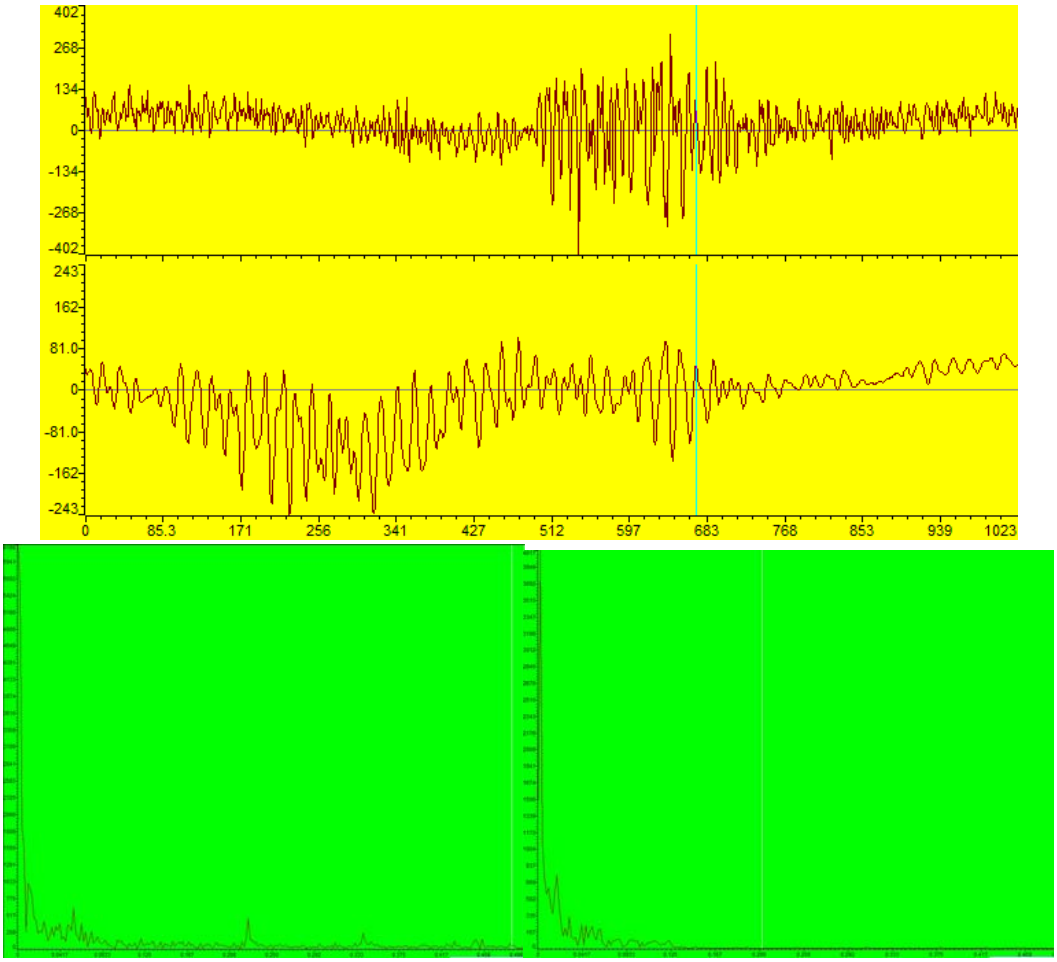


图 4 语音滤波前后的时域和频域图

4.8 思考题

如何改变滤波器的截止频率、通频带等参数。

5 信号处理实验：快速傅立叶变换(FFT)

5.1 实验目的

(5) 了解 FFT 的原理；
(6) 了解使用 Matlab 语言实现 FFT 的方法；
(7) 了解在 DSP 中 FFT 的设计及编程方法；
(8) 熟悉对 FFT 的调试方法；

5.2 实验内容

本试验要求使用 FFT 变换求一个时域信号的频域特性，并从这个频域特性求出该信号的频率值。使用 Matlab 语言实现对 FFT 算法的仿真，然后使用 DSP 汇编语言实现对 FFT 的 DSP 编程。

5.3 实验原理

对于有限长离散数字信号{x[n]}，0≤x≤N-1，它的频谱离散数学值{X（K）}可由离散傅氏变换（DFT）求得。DFT 定义为：

$$X(K)=\sum_{N=0}^{N-1}x[n]e^{-j(2\pi/N)nk} \qquad k=0,1,\cdots,N-1$$

也可以方便的把它改写成如下形式：

$$X(K)=\sum_{N=0}^{N-1}x[n]W_N^{nk}$$

式中 W_N（有时简写为 W）代表 $e^{-j(2\pi/N)nk}$ 。不难看出，W^{nk} 是周期性的，且周期为 N，即

$$W_N^{(n+mN)(k+lN)}=W_N^{nk} \qquad m,l=0,\pm1,\pm2,\cdots\cdots$$

W^{nk} 的周期性是 DFT 的关键之一。为了强调起见，常用表达式 W_N 取代 W 以便明确地给出 W^{nk} 的周期为 N。

由 DFT 的定义可以看出，在 x[n]为复数序列的情况下，完全可以直接运算 N 点 DFT 需要 $(N-1)^2$ 次复数乘法和 N（N-1）次复数加法。因此，对于一些相当大的 N 值（如 1024 点）来说，直接计算它的 DFT 所需要的计算量是很大的。一个优化的实数 FFT 算法是一个组合以后的算法。原始的 2N 个点的实输入序列组合成一个 N 点的复序列，然后对复序列进行 N 点的 FFT 运算，最后再由 N 点复数输出拆散成 2N 点的复数序列，这 2N 点的复数序列与原始的 2N 点的实数输入序列

的 DFT 输出一致。

FFT 的基本思想在于：将原来的 N 点序列分成两个较短的序列，这些序列的 DFT 可很简单地组合起来得到原序列的 DFT。例如，若 N 为偶数，将原有的 N 点序列分成两个（N/2）点序列，那么计算 N 点 DFT 将只需要约[（N/2）^{2}*2]=N^2/2 次复数乘法。即比直接计算少作一半乘法。因子（N/2）^{2} 表示直接计算（N/2）点 DFT 所需要的乘法次数，而乘数 2 代表必须完成两个 DFT。上述处理方法可以反复使用，即（N/2）点的 DFT 计算也可以化成两个（N/4）点 DFT（假定 N/2 为偶数），从而又少作一半乘法。使用这种方法，在组合输入和拆散输出的操作中，FFT 的运算量减半。这样，利用实数 FFT 算法来计算实输入序列的 DFT 的速度几乎是一般复 FFT 算法的两倍。

假定序列{x[n]}的点数 N 是 2 的幂，按上述处理方法，定义两个分别为 x[n]的偶数项和基数项的（N/2）点序列 $x_1[n]$和 $x_2[n]$，即：

$x_1[n]=x[2n]$　　n=0，1，…，（N/2）-1

$x_2[n]=x[2n+1]$　　n=0，1，…，（N/2）-1

{x[n]}的 N 点 DFT 可写成：

$$X(K)=\sum_{N=0}^{N-1}x[n]W_N^{nk} \quad + \quad \sum_{N=0}^{N-1}x[n]W_N^{nk}$$

（n 为偶数）　　（n 为奇数）

$$= \sum_{N=0}^{N/2-1}x[2n]W_N^{2nk} \quad + \quad \sum_{N=0}^{N/2-1}x[2n+1]W_N^{(2n+1)k}$$

因考虑到W_N^k可写成：

$$W_N^k \quad = \quad [e^{-j(2\pi)/N}]^2 \quad = \quad e^{-j(2\pi)/(N/2)} \quad = \quad W_{N/2}$$

故 X（k）可写为：

$$X(K)=\sum_{N=0}^{N/2-1}x_1[n]W_{N/2}^{nk} \quad + \quad W_N^k\sum_{N=0}^{N/2-1}x_2[n]W_{N/2}^{nk}$$

$$= X_1(K)+W_N^kX_2(K)$$

式中$X_1(K)$和$X_2(K)$是$x_1(n)$和$x_2(n)$的（N/2）点 DFT。上式表明，N 点 DFT X（k）

可分解为按上式的规则加以组合的两个（N/2）点 DFT。

5.4　FFT的高级编程

FFT 算法的 Matlab 语言编程十分简单，直接使用 Matlab 下的 FFT 函数就可以了，可以参考 FFT 函数的帮助文件，使得 Matlab 编程简单方便，这里不再做详细的描述，下面介绍 C 语言对 FFT 的编程，了解 FFT 实现的方法。

在本程序中采用地址表查找，其中的数据是用 C 语言计算出来，然后输出到文件中拷上去的。其中 FFT 的算法验证的输入是用 C 语言生成正弦波和方波的数

据放在FFT_INPUT中的，其中的C语言程序如下：

```
＃include<stdio.h>
# include<stdib.h>
int signal(int x)(方波子程序)
 {int t=x%8;
if(t<=3)return 1;
else return －1;
}
void main0
{
int i,j,k;
FILE*fp;
Int mm[256];
for (i=0; i<=255;i++)
mm[i]=(sin(3.141593536*i/4+3.1414/16)*(-32768)/2); 正弦波程序
//mm[i]=signal(i)*(-32768)/2;
k=0;
for(i=0;i<=255;i++)
   {
fp=fopen(“sinsin.txt”, “a”);
if(k%8=0)
fprintf(fp, “\n  .word”);
fprintf(fp, “0%xh,” mm[i]);
fclose(fp);
k++;
}}
```

在程序中由于要用到位到序地址计算，如果用汇编则必须使得所分的首地址为整数，为避免这种情况，采用查表的方法进行位地址的计算，而该表也是由C语言生成的其程序如下：

```
#include<stdio.h>
#include<stdib.h>
#include<math.h>
void main0
 {
    int num[256],i,m=0,n,k,q;
    FILE*fp;
       for(i=0;i<=255;i++)
       {num[i]=I;
        printf(“%d,”,num[i]);
        }
    for(i=0;i<=255;i++)
    {  q=num[i];
        for(k=0;k<=7;k++)
        {
```

```
            n=q%2;
            q=q/2;
            m=m+n*pow(2,(7-k));
           }
           num[i]=m;
           m=0’
      }
      k=0;
      for(i=0;i<=255;i++)
        {
          fp=fopen(“fft_real.txt”, “a”);
          if(k%8=0)
          fprintf(fp, “\n  .word”);
          fprintf(fp, “000%xh,”,num[i]);
          fclose(fp);
          k++;
         }
}
```

在 FFT 的主程序中，是按照每一趟的蝶形来进行运算的，因此，在每一趟的蝶形中要用到 SIN,COS 的值，这些值也是用 C 语言生成的

```
#include<stdio.h>
#include<stdib.h>
#include<math.h>
#define pi 3.1415926536
maim()
{
     int i,j=0,k,m,n;
     FILE*fp;
     static float a0,a1,b1,b2;
     static int w[512][2];
     for((i=0;i<8;i++)
    {for(m=0;m<pow(2,i);m++)
      {w[m][0]=(cos(2*pi*m/pow(2,i+1))*32767);
       w[m][1]=( sin(2*pi*m/pow(2,i+1))*(-32767);
       fp=fopen(“xossin.txt”, “a”);
       if(j%8=0)fprintf(fp, “\n  .word ”);
       j=j+2;
       fprintf(fp, “0%xh,0%xh,”,w[m][0], w[m][1]);
       fclose(fp)
       }
     }
}
```

其中得到值按照用到的顺序排列，使用时只要顺序取出就可以了。

FFT 的主体部分又分为 5 各小部分，其中的顺序如下：1.除掉波形总的直流

部分 2.将要进行 FFT 变换的数据按照位到序的顺序装入到蝶形运算区 3.进行蝶形运算 4.进行功率谱计算 5.求波形的频率

其程序的解释见程序注释。

5.5 FFT的DSP编程

从上面FFT实现的过程可以看出，其实现步骤主要有以下四步：

（1） 将输入序列压缩和位倒序。

（2） N 点的复数 FFT。

（3） 奇数号部分和偶数号部分分离。

（4） 产生最后的输出数据。

最初，将原始输入的 2N 点的实序列 a(n)，存储在 4N 大小的数据处理缓冲区的下半部分。

```
      .asg    AR1,FFT_TWID_P
      LD     #FFT_DP,DP
      STM    #SYSTEM_STACK,SP
    rfft_task:
      STM    #FFT_ORIGIN,AR3          ; to data
      STM    #data_input,AR4
      RPT     #K_FFT_SIZE*2-1
                                      ; K_FFT_SIZE＝128
      MVDD   *AR4+,*AR3+
```

0C00h	
0C01h	
……	
0CFEh	
0CFFh	
0D00h	a(0)
0D01h	a(1)
……	……
0DFFEh	a(254)
0DFFFh	a(255)

然后通过子程序调用，有以下几步计算

①去除原始数据的直流分量。

由于采样的需要，信号发生器输出的为叠加了直流分量的正弦信号，所以有此一步。

```
    .asg AR3,FFT_INPUT
    .def  data_ave
data_ave:
    STM #FFT_ORIGIN,FFT_INPUT       ;  AR3 → 1 st original input
    LD  #0,A
    RPT #K_FFT_SIZE*2-1
```

```
    ADD *FFT_INPUT+,A                    ; 累加原始数据到 A
    NOP
    LD  A,#-1-K_LOGN                     ; A 右移 8 位，即 A/256，输入数
据取平均
    STM #K_FFT_SIZE*2-1,BRC
    STM #FFT_ORIGIN,FFT_INPUT        ; AR3 → 1 st original input
    RPTB ave_end-1
    SUB *FFT_INPUT,A,B                   ; B=A-(*FFT_INPUT)
    NEG B                            ; B=(*FFT_INPUT)-A，即将原始数据去除
直流分量
    STL B,-1,*FFT_INPUT+                 ; 除以 2 以防止溢出后存入原址
ave_end:
    NOP
    RET
```

②将输入序列位倒序存储，以使算法结束后的输出序列为自然顺序。

首先，将 2N 点的实输入序列拷贝到标记为 FFT_ORIGIN 的连续内存空间中，理解为一个 N 点的复数序列 d(n)。时序列中索引为偶数的作为 d(n)的实部，索引为奇数的作为 d(n)的虚部。这一部就是压缩。然后，将得到的复数序列位倒序存储到数据处理缓冲区 fft_data。

0C00h	R(0)=a(0)
0C01h	I(0)=a(1)
0C02h	R(64)=a(128)
0C03h	I(64)= a(129)
……	……
0CFCh	R(63)=a(126)
0CFDh	I(63)=a(127)
0CFEh	R(127)=a(254)
0CFFh	I(127)=a(255)

0D00h	a(0)
0D01h	a(1)
0D02h	a(2)
0D03h	a(3)
……	……
0DFCh	a(252)
0DFDh	a(253)
0DFEh	a(254)
0DFFh	a(255)

```
; Bit Reversal Routine
    .asg AR2,REORDERED_DATA
    .asg AR3,ORIGINAL_INPUT
    .asg AR7,DATA_PROC_BUF
    .def  bit_rev
bit_rev:
    SSBX FRCT                                ; FRCT=1；准备小数乘
法
    STM #FFT_ORIGIN,ORIGINAL_INPUT           ; AR3 -> 1 st original
input
    STM #fft_data,DATA_PROC_BUF              ;  AR7  ->  data
processing buffer
    MVMM DATA_PROC_BUF,REORDERED_DATA    ; AR2 -> 1st bit-reversed
data
    ; DATA_PROC_BUF,REORDERED_DATA 同指向 fft_data
```

```
    ; ORIGINAL_INPUT 指向 FFT_ORIGIN
     STM #K_FFT_SIZE-1,BRC
    RPTBD bit_rev_end-1
    STM #K_FFT_SIZE,AR0                          ; AR0 = 1/2 size of circ
buffer
    MVDD *ORIGINAL_INPUT+,*REORDERED_DATA+
    MVDD *ORIGINAL_INPUT-,*REORDERED_DATA+
    MAR *ORIGINAL_INPUT+0B                         ; 位倒序
bit_rev_end:
    RET                                          ; return to Real FFT main
module
```

③一个 N 点的复数序列存储在数据处理缓冲区。

对 d (n)作 fft 变换，结果得到 D(k)=f{d(n)}=R(k)+jI(k)。

0C00h	R(0)
0C01h	I(0)
0C02h	R(1)
0C03h	I(1)
……	……
0CFCh	R(126)
0CFDh	I(126)
0CFEh	R(127)
0CFFh	I(127)

0D00h	a(0)
0D01h	a(1)
0D02h	a(2)
0D03h	a(3)
……	……
0DFCh	a(252)
0DFDh	a(253)
0DFEh	a(254)
0DFFh	a(255)

```
; 256-Point Real FFT Routine
    .asg AR1,GROUP_COUNTER
    .asg AR2,PX
    .asg AR3,QX
    .asg AR4,WR
    .asg AR5,WI
    .asg AR6,BUTTERFLY_COUNTER
    .asg AR7,DATA_PROC_BUF                          ; for Stages 1 & 2
    .asg AR7,STAGE_COUNTER                          ; for the remaining
stages
    .def  fft
fft:
;                                Stage                                  1
------------------------------------------------------------------------
--------------------------------------
    STM #K_ZERO_BK,BK       ;BK=0 so that *ARn+0% = *ARn+0
                               ; 循环缓冲器大小为 0
    LD #-1,ASM                  ; outputs div by 2 at
each stage
    MVMM DATA_PROC_BUF,PX                       ; PX -> PR
    LD *PX,A                                    ; A := PR
```

```
    STM #fft_data+K_DATA_IDX_1,QX                ; QX -> QR
    STM #K_FFT_SIZE/2-1,BRC
    RPTBD stage1end-1
    STM #K_DATA_IDX_1+1,AR0
    SUB *QX,16,A,B                               ; B := PR-QR
    ADD *QX,16,A                                ; A := PR+QR
    STH A,ASM,*PX+                              ; PR' := (PR+QR)/2
    ST B,*QX+                                   ; QR' := (PR-QR)/2
    ||LD *PX,A                                   ; A := PI
    SUB *QX,16,A,B                               ; B := PI-QI
    ADD *QX,16,A                                ; A := PI+QI
    STH A,ASM,*PX+0                             ; PI' := (PI+QI)/2
    ST B,*QX+0%                                 ; QI' := (PI-QI)/2
    ||LD *PX,A                                   ; A := next PR
stage1end:

                                                ; Stage 2
    MVMM DATA_PROC_BUF,PX                       ; PX -> PR
    STM #fft_data+K_DATA_IDX_2,QX                ; QX -> QR
    STM #K_FFT_SIZE/4-1,BRC
    LD *PX,A                                    ; A := PR
    RPTBD stage2end-1
    STM #K_DATA_IDX_2+1,AR0

                                                ; 1st butterfly
    SUB *QX,16,A,B                               ; B := PR-QR
    ADD *QX,16,A                                ; A := PR+QR
    STH A,ASM,*PX+                              ; PR' := (PR+QR)/2
    ST B,*QX+                                   ; QR' := (PR-QR)/2
    ||LD *PX,A                                   ; A := PI
    SUB *QX,16,A,B                               ; B := PI-QI
    ADD *QX,16,A                                ; A := PI+QI
    STH A,ASM,*PX+                              ; PI' := (PI+QI)/2
    STH B,ASM,*QX+                              ; QI' := (PI-QI)/2
                                                ; 2nd butterfly
    MAR *QX+
    ADD *PX,*QX,A                                ; A := PR+QI
    SUB *PX,*QX-,B                               ; B := PR-QI
    STH A,ASM,*PX+                              ; PR' := (PR+QI)/2
    SUB *PX,*QX,A                                ; A := PI-QR
    ST B,*QX                                    ; QR' := (PR-QI)/2
    ||LD *QX+,B                                  ; B := QR
    ST A, *PX                                   ; PI' := (PI-QR)/2
    ||ADD *PX+0%,A                               ; A := PI+QR
    ST A,*QX+0%                                 ; QI' := (PI+QR)/2
    ||LD *PX,A                                   ; A := PR
```

```
stage2end:
                                        ; Stage 3 thru Stage
                                      logN-1
    STM #K_TWID_TBL_SIZE,BK             ; BK = twiddle table size
always
    ST #K_TWID_IDX_3,d_twid_idx             ; init index of
twiddle table
    STM #K_TWID_IDX_3,AR0               ; AR0 = index of twiddle
table
    STM #cosine,WR                          ; init WR pointer
    STM #sine,WI                        ; init WI pointer
    STM #K_LOGN-2-1,STAGE_COUNTER       ; init stage counter
    ST #K_FFT_SIZE/8-1,d_grps_cnt           ; init group counter
    STM #K_FLY_COUNT_3-1,BUTTERFLY_COUNTER  ; init butterfly counter
    ST #K_DATA_IDX_3,d_data_idx             ; init index for input
data
stage:
    STM #fft_data,PX                        ; PX -> PR
    LD d_data_idx, A
    ADD *(PX),A
    STLM A,QX                           ; QX -> QR
    MVDK d_grps_cnt,GROUP_COUNTER       ; AR1 contains group
counter
;A circular buffer of size R must start on a N-bit boundary (that is, the
N LSBs
;of the base address of the circular buffer must be 0), where N is the
smallest
;integer that satisfies 2 N > R. The value R must be loaded into BK.
group:
    MVMD BUTTERFLY_COUNTER,BRC  ; # of butterflies in each grp
    RPTBD butterflyend-1
    LD *WR,T                    ; T := WR
    MPY *QX+,A                  ; A := QR*WR || QX->QI
    MACR *WI+0%,*QX-,A          ; A := QR*WR+QI*WI  || QX->QR
    ADD *PX,16,A,B                  ; B := (QR*WR+QI*WI)+PR
    ST B,*PX                    ; PR' :=((QR*WR+QI*WI)+PR)/2
    ||SUB *PX+,B                ; B := PR-(QR*WR+QI*WI) || PX->PI
    ST B,*QX                    ;QR' :=(PR-(QR*WR+QI*WI))/2
    ||MPY *QX+,A                ; A := QR*WI [T=WI] || QX->QI
    MASR *QX,*WR+0%,A           ; A := QR*WI-QI*WR
    ADD *PX,16,A,B                  ; B := (QR*WI-QI*WR)+PI
    ST B,*QX+                   ; QI' :=((QR*WI-QI*WR)+PI)/2   ||
QX->QR
    ||SUB *PX,B                     ; B := PI-(QR*WI-QI*WR)
```

```
    LD *WR,T                        ; T := WR
    ST B,*PX+                       ; PI' := (PI-(QR*WI-QI*WR))/2   ||
PX->PR
    ||MPY *QX+,A                    ; A := QR*WR || QX->QI

butterflyend:
                                    ; Update pointers for next group
    PSHM AR0                        ; preserve AR0
    MVDK d_data_idx,AR0
    MAR *PX+0                       ; increment PX for next group
    MAR *QX+0                     ; increment QX for next group
    BANZD group,*GROUP_COUNTER-
    POPM AR0                        ; restore AR0
    MAR *QX-
                                    ; Update counters and indices for
                                  next stage
    LD d_data_idx,A
    SUB #1,A,B                          ; B = A-1
    STLM B,BUTTERFLY_COUNTER      ; BUTTERFLY_COUNTER = #flies-1
    STL A,1,d_data_idx                  ; double the index of data
    LD d_grps_cnt,A
    STL A,ASM,d_grps_cnt                ; 1/2 the offset to next group
    LD d_twid_idx,A
    STL A,ASM,d_twid_idx                ; 1/2 the index of twiddle table
    BANZD stage,*STAGE_COUNTER-
    MVDK d_twid_idx,AR0             ; AR0 = index of twiddle table
fft_end:
    RET                             ; return to Real FFT main module
```

④将 fft 的计算结果分离为 RP（偶实部），RM（奇实部），IP（偶虚部），IM（奇虚部）。

RP(k)=RP(N-k)=0.5*(R(k)+R(N-k))
RM(k)=-RM(N-k)=0.5*(R(k)-R(N-k))
IP(k)=IP(N-k)=0.5*(I(k)+I(N-k))
IM(k)=-IM(N-k)=0.5*(I(k)-I(N-k))
RP(0)=R(0)
IP(0)=I(0)
RM(0)=IM(0)=RM(N/2)=IM(N/2)=0
RP(N/2)=R(N/2)
IP(N/2)=I(N/2)

0C00h	RP(0)=R(0)
0C01h	IP(0)=I(0)
0C02h	RP(1)
0C03h	IP(1)

……	……
0CFCh	RP(126)
0CFDh	IP(126)
0CFEh	RP(127)

0CFFh	IP(127)
0D00h	a(0)
0D01h	a(1)
0D02h	IM(127)
0D03h	RM(127)

……	……
0DFCh	IM(2)
0DFDh	RM(2)
0DFEh	IM(1)
0DFFh	RM(1)

```
;======================================================================
;Unpack 256-Point Real FFT Output
    .def  unpack
unpack:
; Compute intermediate values RP, RM, IP, IM
    .asg AR2,XP_k
    .asg AR3,XP_Nminusk
    .asg AR6,XM_k
    .asg AR7,XM_Nminusk

    STM #fft_data+2,XP_k                          ; AR2 -> R[k] (temp
RP[k])
    STM #fft_data+2*K_FFT_SIZE-2,XP_Nminusk       ;   AR3   ->   R[N-k]
(tempRP[N-k])
    STM #fft_data+2*K_FFT_SIZE+3,XM_Nminusk       ; AR7 -> temp RM[N-k]
    STM #fft_data+4*K_FFT_SIZE-1,XM_k             ; AR6 -> temp RM[k]
    STM #-2+K_FFT_SIZE/2,BRC
    RPTBD phase3end-1
    STM #3,AR0
    ADD *XP_k,*XP_Nminusk,A                    ;   A   :=   R[k]+R[N-k]
=2*RP[k]
    SUB *XP_k,*XP_Nminusk,B                    ;   B   :=   R[k]-R[N-k]
=2*RM[k]
    STH A,ASM,*XP_k+                            ; store RP[k] at AR[k]
    STH A,ASM,*XP_Nminusk+      ; store RP[N-k]=RP[k] at AR[N-k]
    STH B,ASM,*XM_k-            ; store RM[k] at AI[2N-k]
    NEG B                       ; B := R[N-k]-R[k] =2*RM[N-k]
    STH B,ASM,*XM_Nminusk-      ; store RM[N-k] at AI[N+k]
    ADD *XP_k,*XP_Nminusk,A     ; A := I[k]+I[N-k] =2*IP[k]
    SUB *XP_k,*XP_Nminusk,B     ; B := I[k]-I[N-k] =2*IM[k]
    STH A,ASM,*XP_k+            ; store IP[k] at AI[k]
    STH A,ASM,*XP_Nminusk-0     ; store IP[N-k]=IP[k] at AI[N-k]
    STH B,ASM,*XM_k-            ; store IM[k] at AR[2N-k]
    NEG B                       ; B := I[N-k]-I[k] =2*IM[N-k]
    STH B,ASM,*XM_Nminusk+0     ; store IM[N-k] at AR[N+k]
phase3end:
    ST #0,*XM_k-                ; RM[N/2]=0
    ST #0,*XM_k                 ; IM[N/2]=0
```

进而得到原始序列的傅利叶变换：

AR(k)=AR(2N－k)=RP(k)+cos(k　/N)*IP(k)－sin(k　/N)*RM(k)

AI(k)=－AI(2N－k)=IM(k)－cos(k　/N)*RM(k)－sin(k　/N)*IP(k)

AR(0)=RP(0)+IP(0)

AI(0)=IM(0)－RM(0)

AR(N)=R(0)－I(0)

AI(N)=0

这里：

A(k)=A(2N－k)=AR(k)+j AI(k)=F{a(n)}

0C00h	AR(0)
0C01h	AI(0)
0C02h	AR(1)
0C03h	AI(1)
……	……
0CFEh	AR(127)
0CFFh	AI(127)

0D00h	AR(128)
0D01h	AI(128)
……	……
0DFCh	AR(254)
0DFDh	AI(254)
0DFEh	AR(255)
0DFFh	AI(255)

```
; Compute AR[0],AI[0], AR[N], AI[N]
    .asg AR2,AX_k
    .asg AR4,IP_0
    .asg AR5,AX_N
    STM #fft_data,AX_k                  ; AR2 -> AR[0] (tempRP[0])
    STM #fft_data+1,IP_0                ; AR4 -> AI[0] (tempIP[0])
    STM #fft_data+2*K_FFT_SIZE+1,AX_N   ; AR5 -> AI[N]
    ADD *AX_k,*IP_0,A                   ; A := RP[0]+IP[0]
    SUB *AX_k,*IP_0,B                   ; B := RP[0]-IP[0]
    STH A,ASM,*AX_k+                ; AR[0] = (RP[0]+IP[0])/2
    ST #0,*AX_k                     ; AI[0] = 0
    MVDD *AX_k+,*AX_N-              ; AI[N] = 0
    STH B,ASM,*AX_N                 ; AR[N] = (RP[0]-IP[0])/2
; Compute final output values AR[k], AI[k]
    .asg AR3,AX_2Nminusk
    .asg AR4,COS
    .asg AR5,SIN
    STM #fft_data+4*K_FFT_SIZE-1,AX_2Nminusk      ;       AR3       ->
AI[2N-1](temp RM[1])
    STM #cosine+K_TWID_TBL_SIZE/K_FFT_SIZE,COS    ; AR4 -> cos(k*pi/N)
    STM #sine+K_TWID_TBL_SIZE/K_FFT_SIZE,SIN  ; AR5 -> sin(k*pi/N)
    STM #K_FFT_SIZE-2,BRC
    RPTBD phase4end-1
    STM #K_TWID_TBL_SIZE/K_FFT_SIZE,AR0       ; index of twiddle tables
    LD *AX_k+,16,A                                ;    A    :=    RP[k]
||AR2->IP[k]
```

```
    MACR *COS,*AX_k,A                              ;
A :=A+cos(k*pi/N)*IP[k]
    MASR *SIN,*AX_2Nminusk-,A          ; A := A-sin(k*pi/N)*RM[k]   ||
AR3->IM[k]
    LD *AX_2Nminusk+,16,B              ; B := IM[k] ||AR3->RM[k]
    MASR *SIN+0%,*AX_k-,B              ; B := B-sin(k*pi/N)*IP[k]   ||
AR2->RP[k]
    MASR *COS+0%,*AX_2Nminusk,B      ; B := B-cos(k*pi/N)*RM[k]
    STH A,ASM,*AX_k+                   ; AR[k] = A/2
    STH B,ASM,*AX_k+                   ; AI[k] = B/2
    NEG B                               ; B := -B
    STH B,ASM,*AX_2Nminusk-           ; AI[2N-k] = -AI[k]= B/2
    STH A,ASM,*AX_2Nminusk-           ; AR[2N-k] = AR[k] = A/2
phase4end:
    RET                    ; returntoRealFFTmain module
```

⑤计算功率谱密度

$P[k]=AR(k)^2+ AI(k)^2$

```
;=======================================================
;Compute the Power Spectrum of the Complex Output of the 256-Point Real
FFT
    .asg AR2,AX
    .asg AR3,OUTPUT_BUF
    .def  power

power:
    STM #data_output,OUTPUT_BUF        ; AR3 points to output buffer
    STM #K_FFT_SIZE*2-1,BRC
    RPTBD power_end-1
    STM #fft_data,AX                    ; AR2 points to AR[0]
    SQUR *AX+,A                         ; A := ARˆ2
    SQURA *AX+,A                        ; A := ARˆ2 + AIˆ2
    STH A,*OUTPUT_BUF+
power_end:
    RET                    ; return to main program
```

⑥计算输入信号主频

$f_{输入}= Kf_s/2N$

其中 P（K）为功率谱的最大值，f_s为采样频率，2N 为样本点数

```
    .def find                              .asg AR5,MAXFREQ
    .asg AR2,POWER                    find:
    .asg AR3,STORE                        STM #data_output,POWER
    .asg AR4,INDEX                        STM #data_find,STORE
```

```
    STM #data_index,INDEX
    STM #data_freq,MAXFREQ
    LD  #0,A
    LD  #0,B
    STL A,*STORE
    STM #K_FFT_SIZE-1,BRC
    RPTB find_end-1
    SUB *POWER,*STORE,A
    BC  next,ALEQ
    MVDD *POWER,*STORE
    STL  B,*INDEX
next:
    MAR *POWER+
    ADD #1,B
find_end:
     RSBX FRCT
    MPY
*INDEX,#K_SAMPLE_RATE,A     ;A*Sa
mple rate
    LD
A,#-1-K_LOGN                ;A/25
6
    STL A,*MAXFREQ
    RET
```

5.6 实验步骤

1. 打开 CCS，新建立一工程文件 FFT.pjt。

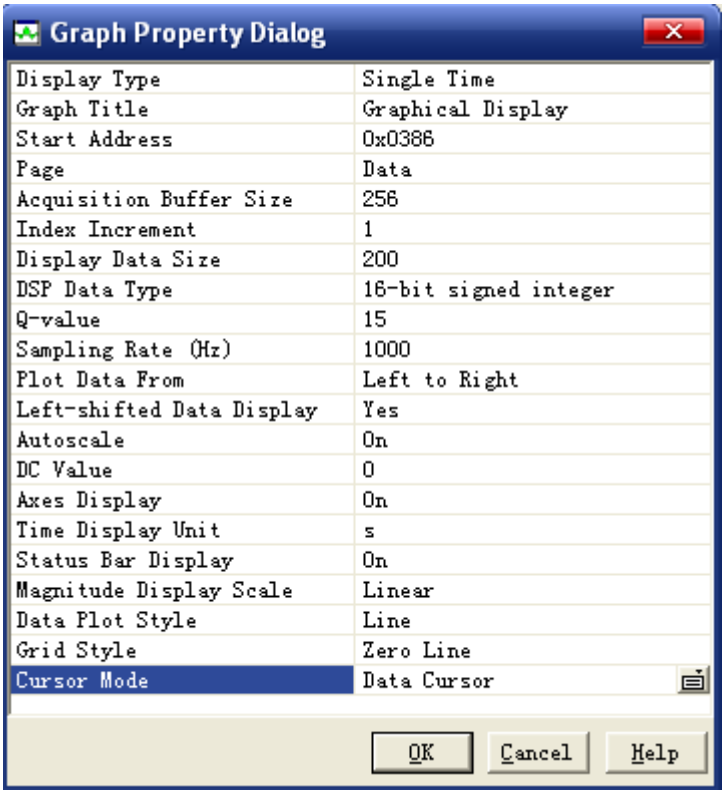
2.将汇编源文件 fft.asm、中断向量表 vectors.asm 和链接命令文件 fft.cmd 添加到 fft.pjt 中。

3. 在 project 菜单下选择 build options 选项，选取 Linker 选项，调整为 -q -c -m".\Debug\fft.map" -o".\Debug\fft.out" -w -x。点击编译，链接图标，通过后生成 fft.out 文件和 fft.map 文件，其余选项可默认。

4. 在 file 菜单下，选择 load program 选项，将生成的 fft.out 文件装载到 DSP 中。

5. 运行程序，在 view 菜单下选择 watch window 选项来观测变量值。

6. 可以在 view 菜单下选择 graph/time frequence，弹出如下对话框。



按照要求，设置好相应的参数，来观测信号输入和经 FFT 变换后输出数据的波形。

7. 具体调试执行程序时，可使用断点，单步执行等方式。

5.7 实验结果

1. 程序运行起始地址为 3000H，输入的数据在数据空间地址为 1400H，长度为 400H，输出的功率谱在数据空间地址 1800H，长度为 400H。
2. 程序运行前的输入数据的时域图和频域图如下：

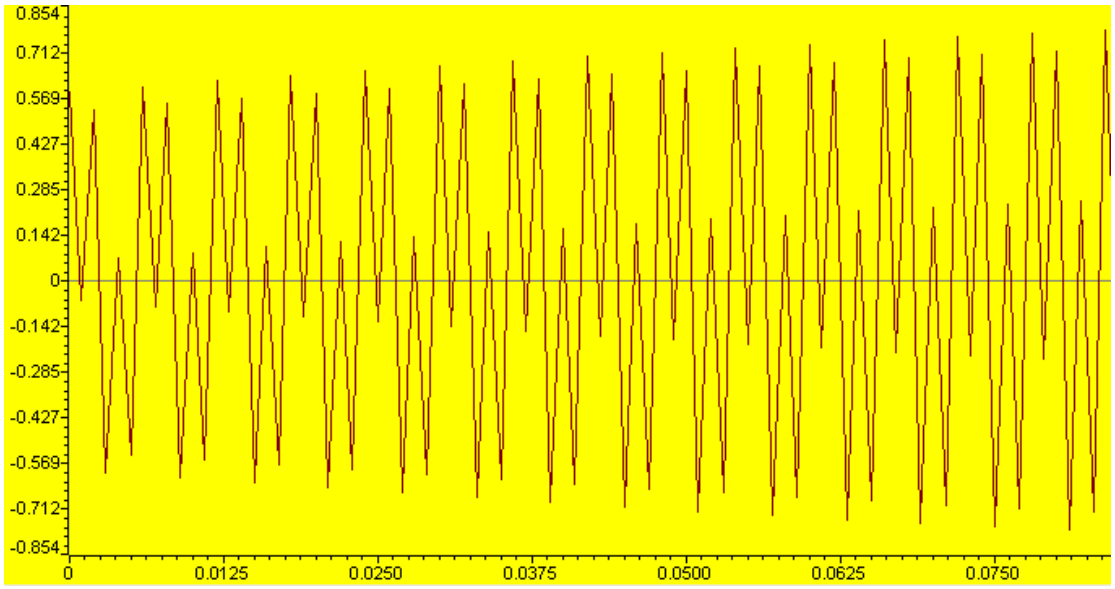


图 1 输入数据的时域图

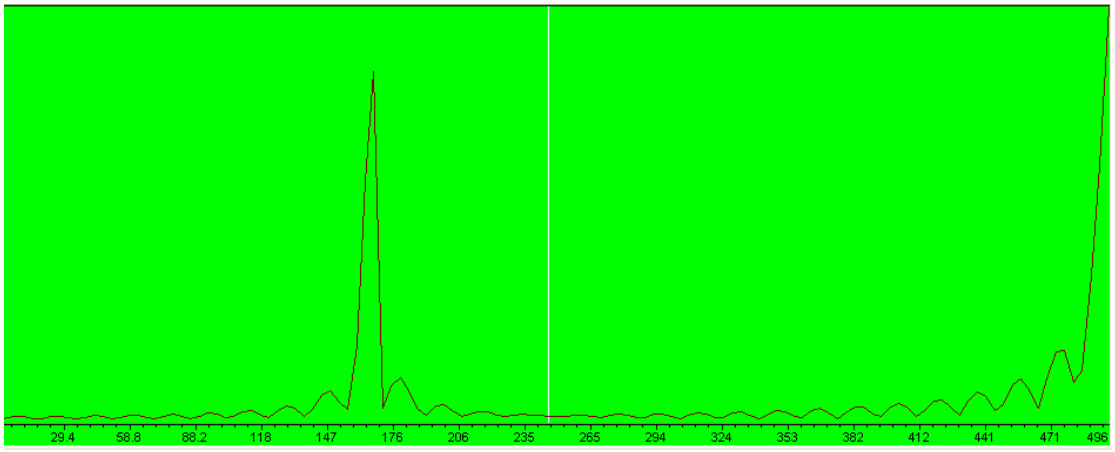


图 2 输入数据的频域图

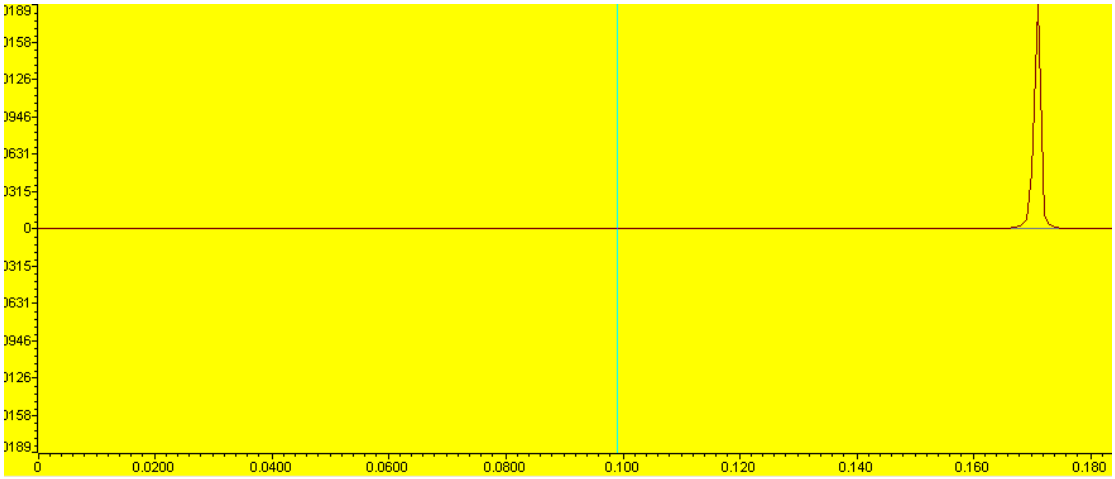


图 3 经 FFT 变换后输出数据的功率谱图

图 1 中的横坐标表示点数;图 2 中的横坐标表示频率,图中设置为 500Hz。图 1 和图 2 在调入程序到 CCS 中，在程序运行之前就可以得到。

运行程序，可以得到图 3，图 3 为程序运行后计算出输入信号的功率谱图，直接在 CCS 中描点画图就可以得到图 3。

5.8 思考题

从图 2 中可以看到，输入信号的频率成分为 170Hz 和 500hz，而图 3 中频率成分为 170Hz,这是由于 CCS 内部的 FFT 变换已经将频率的高一半去除。这是如何引起的?