使用 CCS 进行 DSP 编程（一）

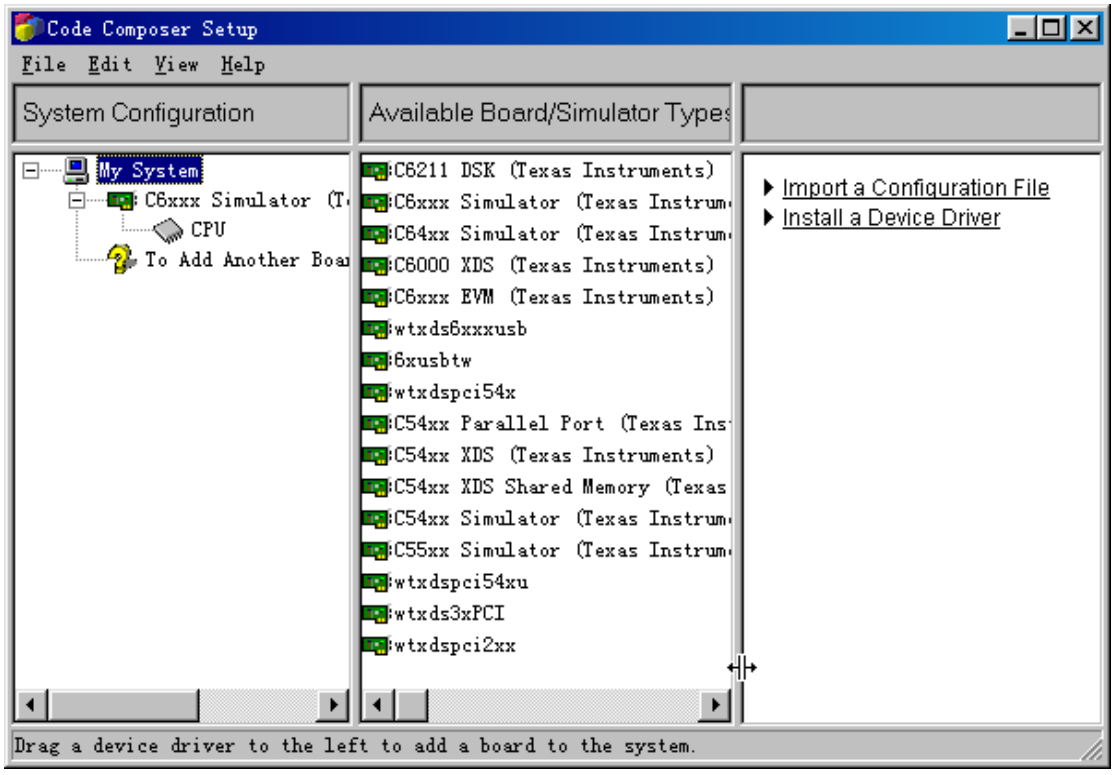
——CCS 编程入门

pacificxu

TI 公司提供了高效的 C 编译器和集成开发环境 Code Composer Studio，学习‘ C6X 的编程应该从学习 CCS 的使用开始。

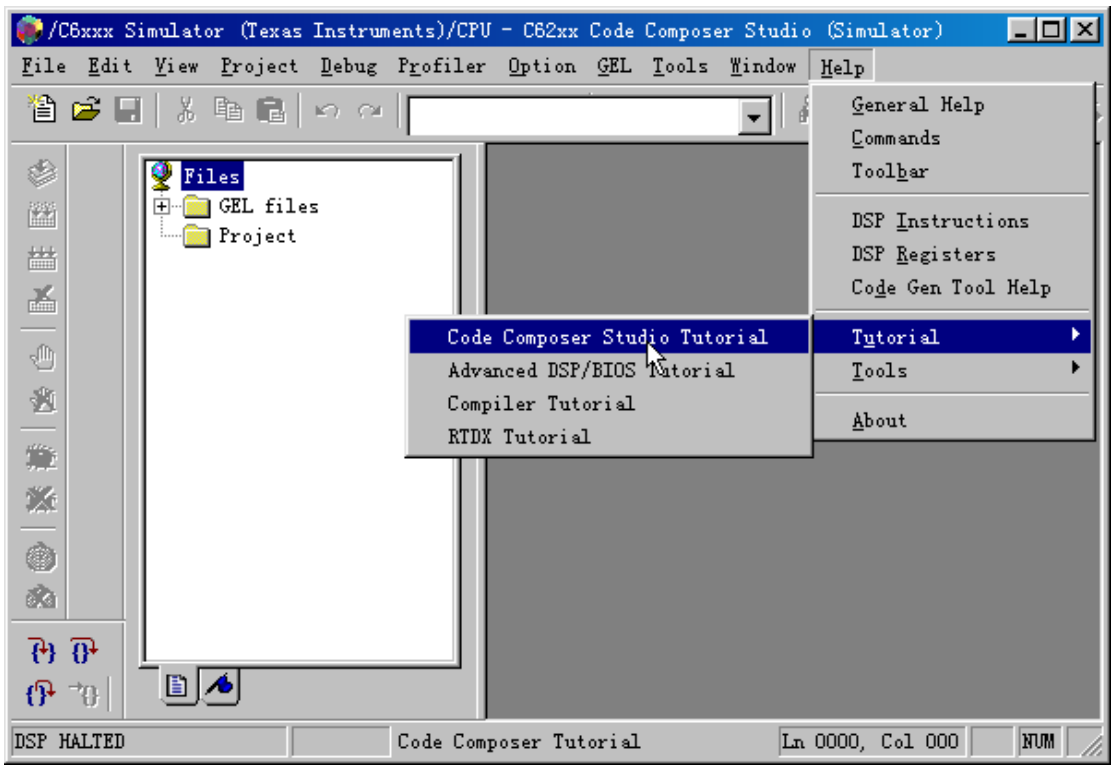
首先安装 CCS，CCS 的安装有详细的说明，并配有简短的 Quick Time 的多媒体介绍，对于没有购买 CCS 的用户，可以从 TI 处得到 30 天的试用版（没有硬件仿真功能）。

使用 CCS 前需要对 CCS 进行设置，以 Simulator 为例，运行 Setup CCS C6000 1.20，安装 Device Driver，对于有硬件支持的仿真器，可以选择配套的 CCS 驱动，设置完成的画面如下图所示：用户的界面大致相同。

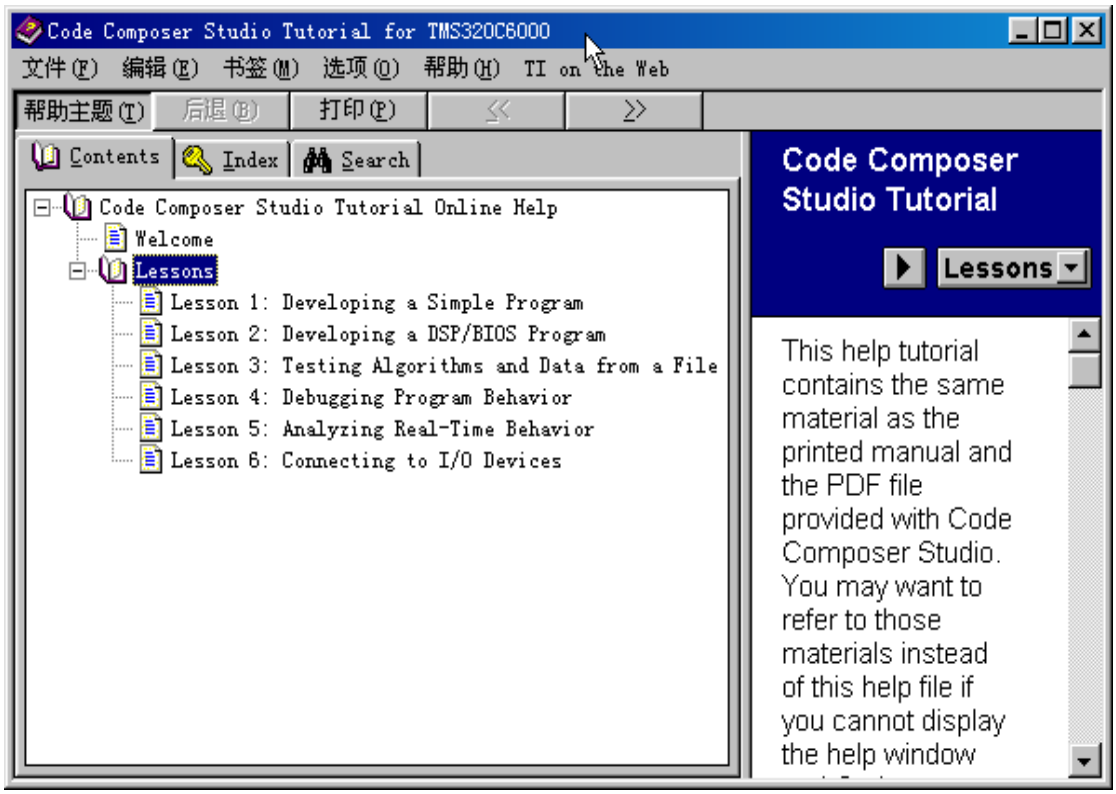


接下来就可以运行 CCS 了，**CCS 提供了比较好的例子，对于初学者，仔细学习这些例子，会起到事半功倍的效果**。在 CCS 的 Help 菜单的 Tutorial 子菜单下，给出了四个教程，分别是：Code Composer Studio Tutorial、Advanced DSP/BIOS Tutorial、Compiler Tutorial 和 RTDX Tutorial，用户可以从简单的 CCS 功能

开始，如创建一个工程文件 Project，到创建一个完善的用户程序一步一步的进行。



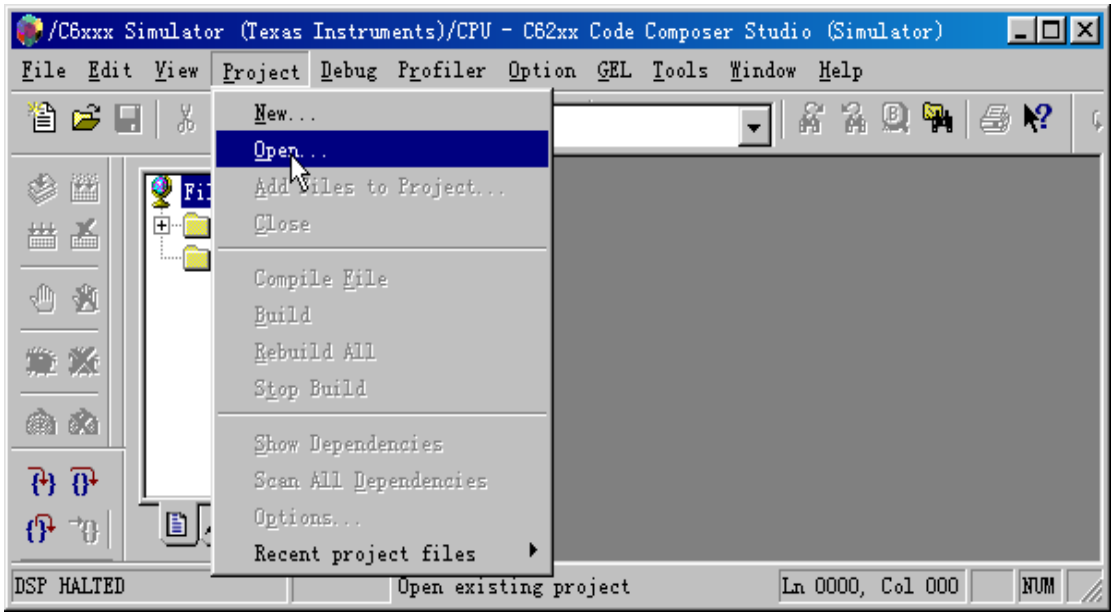
下面是 Code Composer Studio Tutorial 的例子：



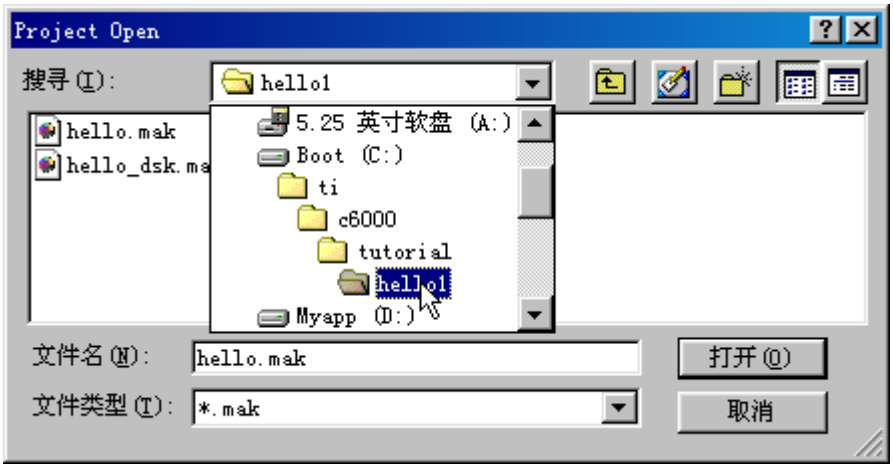
分别从生成一个简单的“ Hello World ”程序，到使用 DSP/BIOS 功能，到程序的调试，实时分析，I/O 操作等分 6 课来讲解，可以领略 TI 的 CCS 的强大功能。

下面以“ Hello World ”程序为例讲一下 CCS 的使用。

首先打开一个 Project 文件



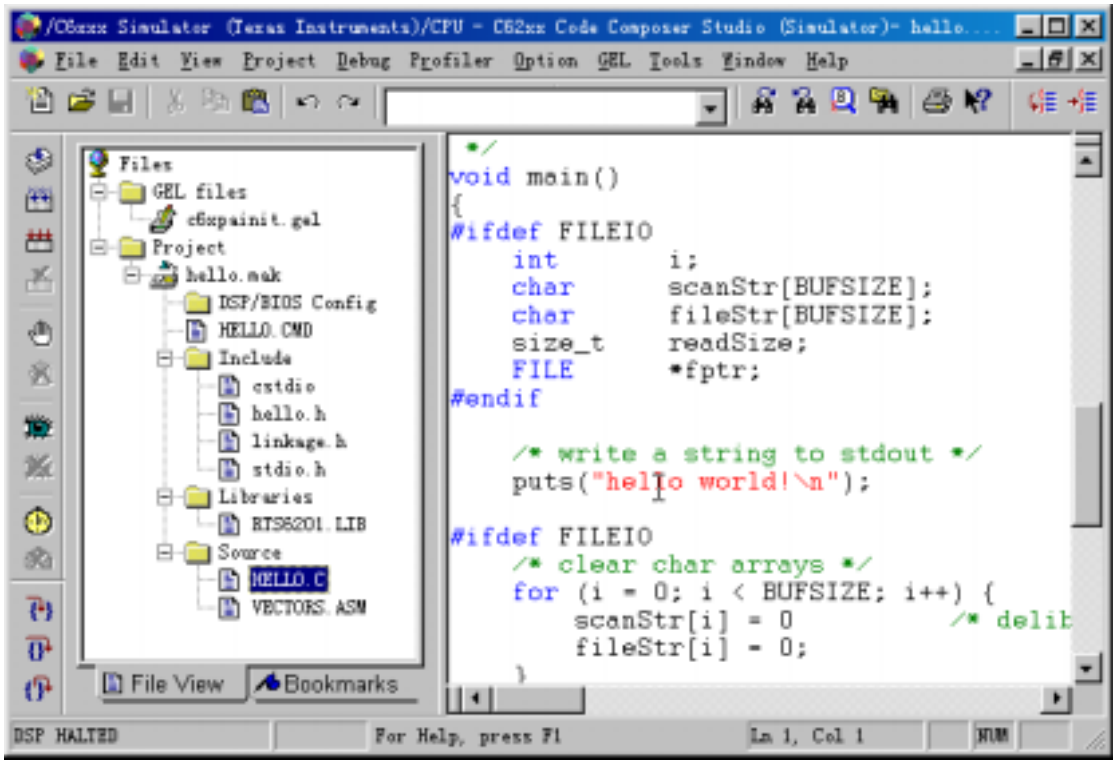
这些文件的路径如下图所示：



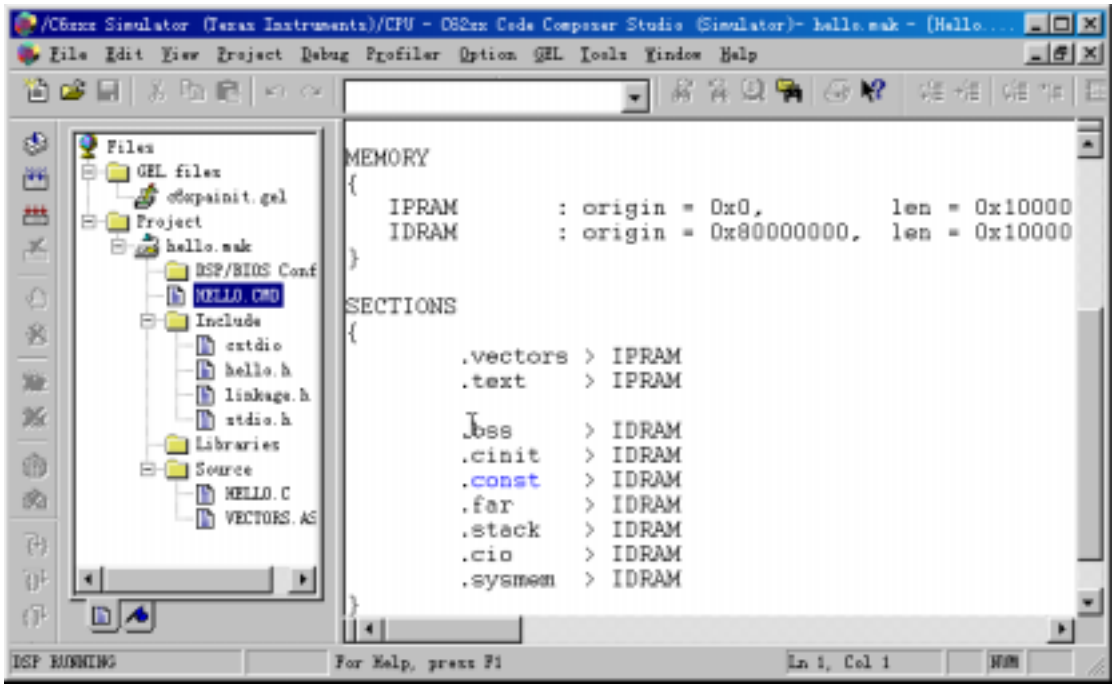
打开 hello.mak，会看到如下图所示的界面。将 File View 栏中的“ + ”号都打开，会看到整个项目工程中的所有资源。

其中*.c 文件和*.h 文件与普通的 C 语言编程中是一致的（TI 编译器支持 ANSI C 标准）。需要指出的是三个文件：HELLO.CMD、RTS6201.LIB、VECTORS.ASM。HELLO.CMD 文件给出了程序空间和数据空间的设置、及编译后各程序段在程序或数据空间的具体位置。RTS6201.LIB 文件为 DSP 运行时库，VECTORS.ASM 为中断

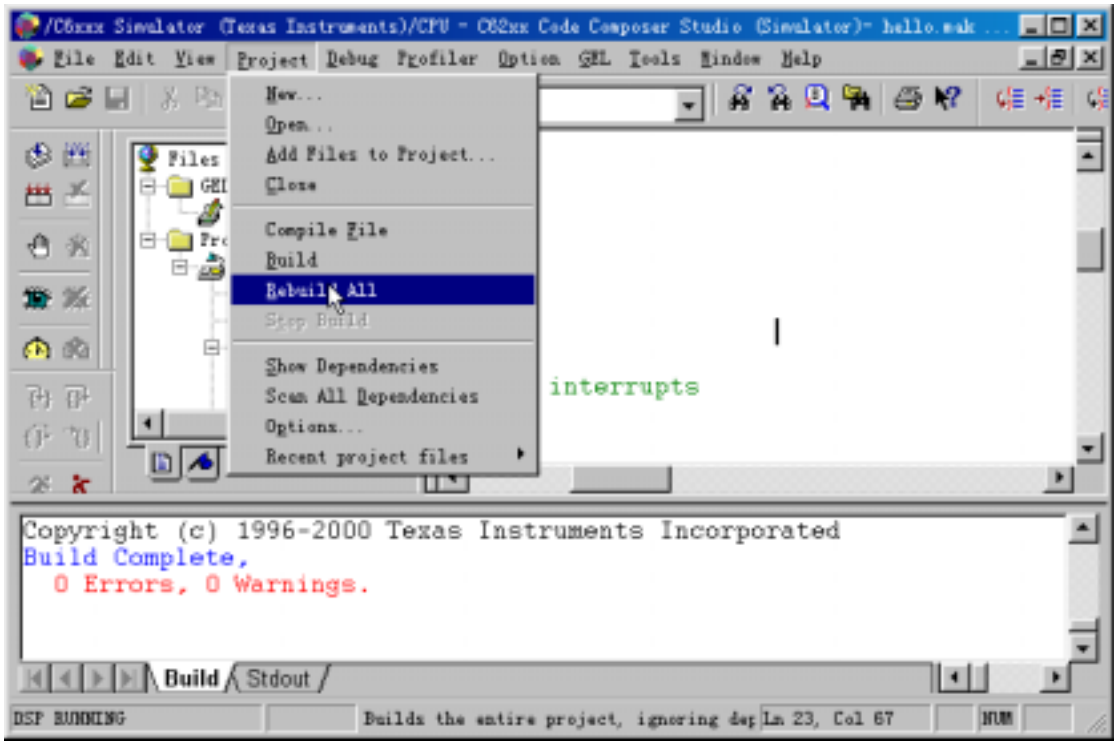
向量表，都是区别于纯软件编程的独到之处，熟悉以后会有更深的体会。



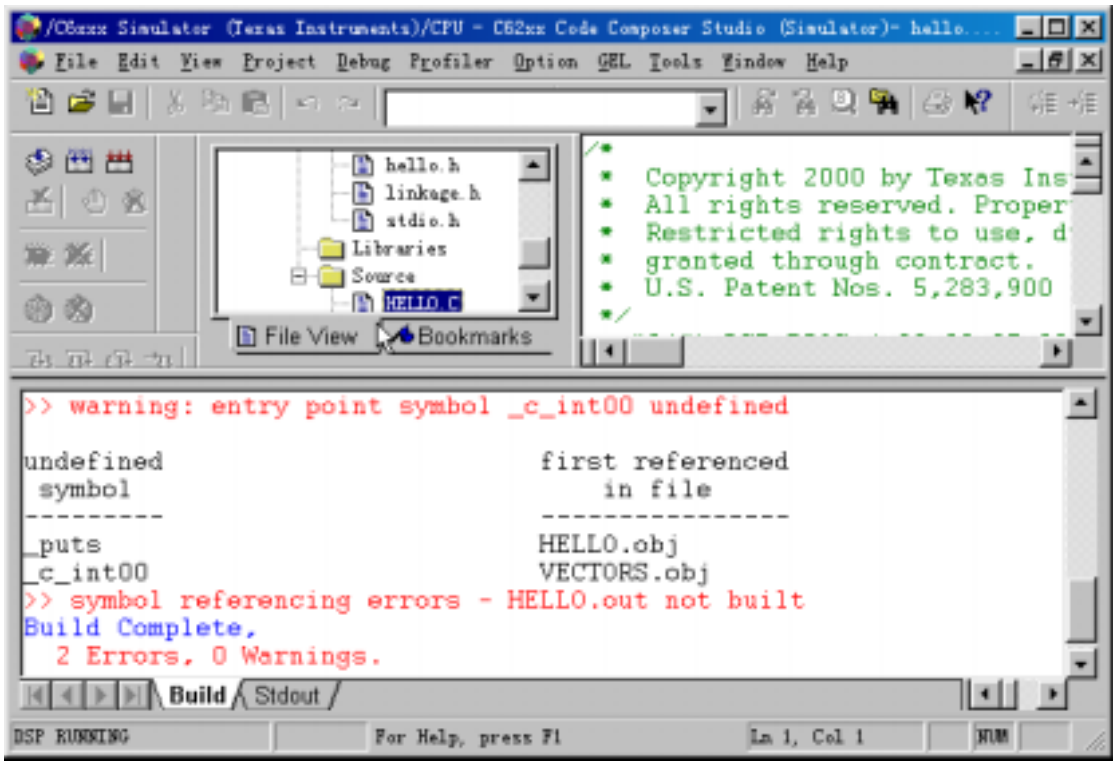
下图为 HELLO.CMD 文件的代码，MEMORY 分为程序空间 IPRAM 和数据空间 IDRAM，并分别给出了起始地址 origin 和长度 len，各段在 MEMORY 空间的分配也作了定义。对于实际的目标板硬件系统，由实际的存储器空间及 DSP 芯片上的存储空间决定。对于软件仿真，可以不考虑有没有 MEMORY 资源。



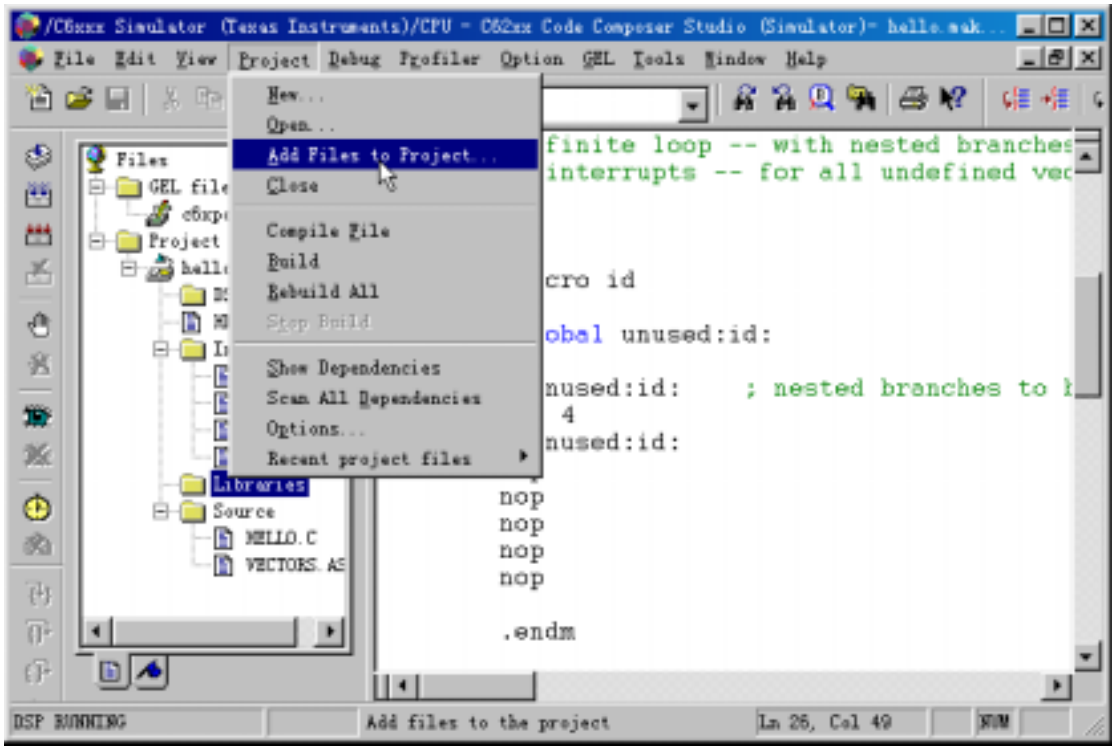
直接对该工程进行编译，会得到如下结果，试一下吧！也可以试一下快捷工具条上的按钮，随便点击鼠标右键，也会有意外的收获。怎么样？没有错误吧！



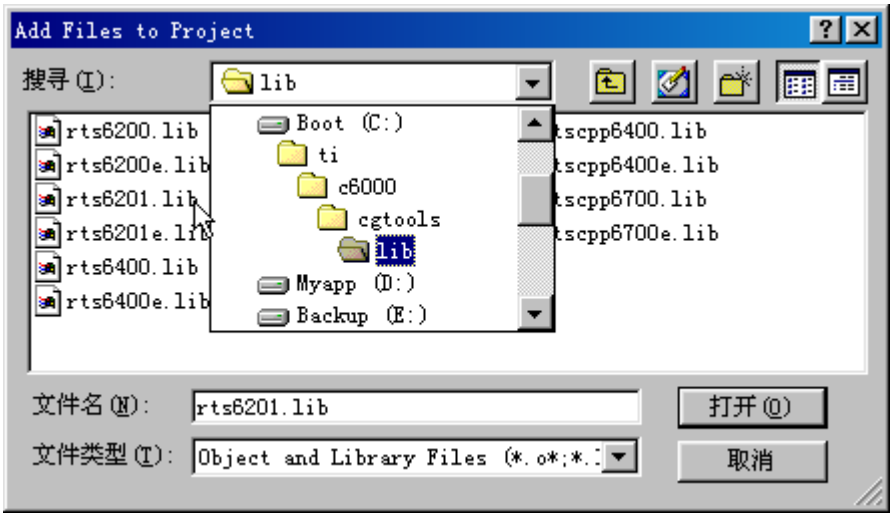
自己在编写工程项目文件时，经常会遇到下面的问题，没有 C 语言程序的入口函数，细心比较一下会发现工程文件中缺少了一个运行时支持库 RTS6201.LIB，不同的 DSP 芯片需要不同的运行时库来支持。



下面向项目工程中加上运行时库 RTS6201.LIB 来纠正刚才的编译错误，**同样的方法可以用来向工程中添加*.c、*.cmd、*.asm 文件**。*.h 文件在编译时会自己找到（当然需要在环境变量中设置好啦，一般不需要改动）。



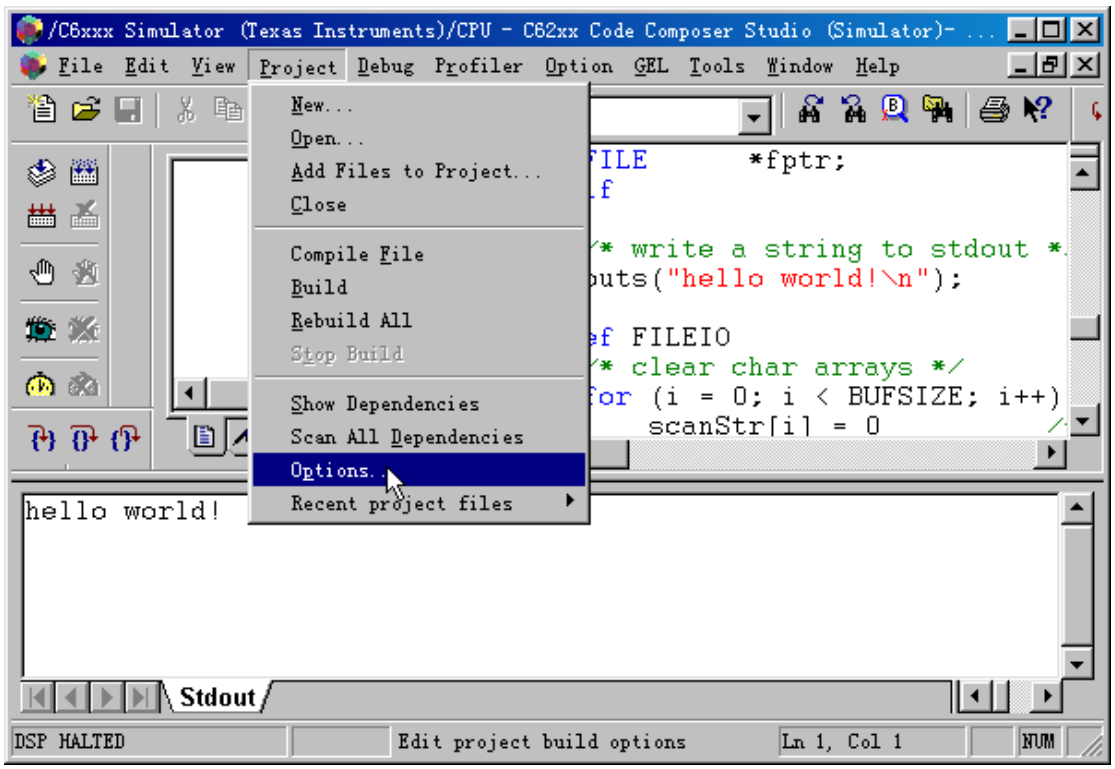
运行时库在 TI 的缺省路径下，注意将文件类型改为*.lib，

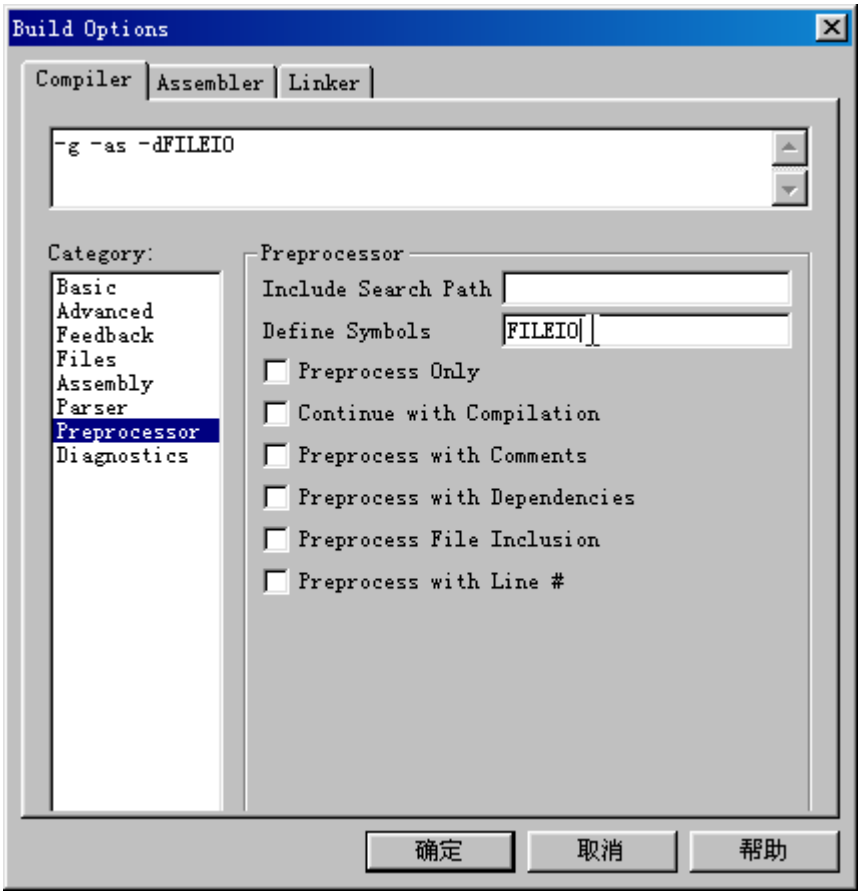


大家可能注意到，在 HELLO.C 文件中有这样的定义：

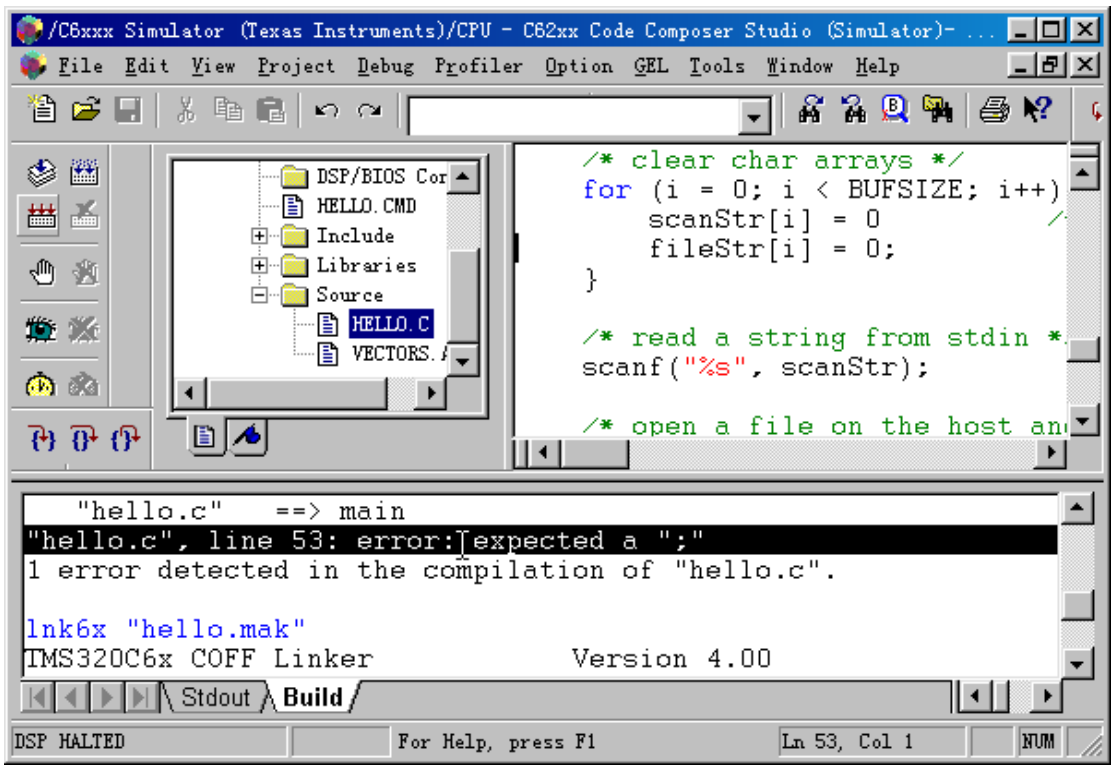
```
#ifdef FILEIO
    int       i;
    char      scanStr[BUFSIZE];
    char      fileStr[BUFSIZE];
    size_t    readSize;
    FILE      *fptr;
#endif
#ifdef FILEIO
    /* clear char arrays */
    for (i = 0; i < BUFSIZE; i++) {
        scanStr[i] = 0          /* deliberate syntax error */
        fileStr[i] = 0;
    }

    /* read a string from stdin */
    scanf("%s", scanStr);

    /* open a file on the host and write char array */
    fptr = fopen("file.txt", "w");
    fprintf(fptr, "%s", scanStr);
    fclose(fptr);

    /* open a file on the host and read char array */
    fptr = fopen("file.txt", "r");
    fseek(fptr, 0L, SEEK_SET);
    readSize = fread(fileStr, sizeof(char), BUFSIZE, fptr);
    printf("Read a %d byte char array: %s \n", readSize, fileStr);
    fclose(fptr);
#endif
```

其中还有一些变量的定义和对文件的操作，运行编译好的程序后好象这些语句都没有执行，因为在 CCS 的编译环境中这个参数还没有定义。按下图进行设置：





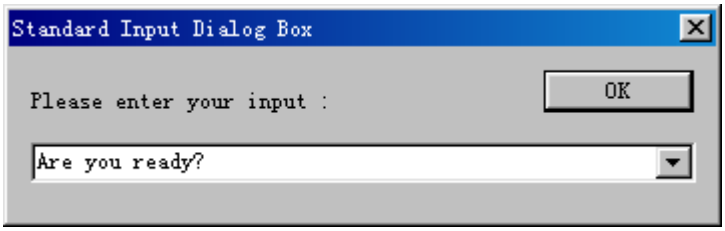
设置完成后可以进行重新编译，会发现新的错误（如果没有出现这个错误，说明设置的不对）。双击这个错误，在 HELLO.C 文件中，光标会出现在出错的地方。



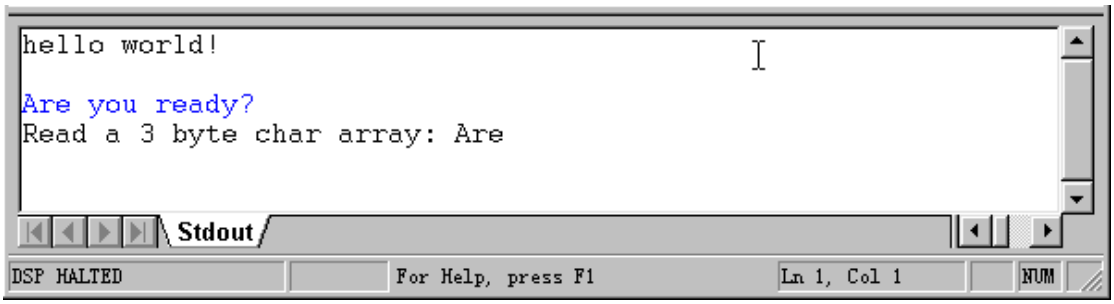
在第 52 行的这一句，可以看到语句的后面没有加“ 分号 ”，

```
scanStr[i] = 0
```

加上“ 分号 ”后重新编译，ok？！加载 hello.out 运行，会出现下面的输入界面，



输入一串文字并确定，在“ Stdout ”窗口会有下面的显示，



小结：在这里简单介绍了 CCS 的使用，包括 CCS 的设置、帮助文件的使用，（**TI 的帮助文件系统、详细地介绍了 CCS 的使用，强烈建议用户认真学习。**）

并以“ Hello World ”程序为例对 CCS 的使用中容易出现问题的一些地方作了一般的介绍，包括运行时库的添加、预编译定义设置等，用户在使用过程中会不断发现问题，通过使用 TI 的帮助文件及配套的资料会不断提高，不可急于求成，如果用户对 Visual C++比较熟悉，学起来会快很多；相反，那肯定要多花一些时间来学习了，学习 CCS 跟学习 Visual C++一样（简单/复杂？），但需要对硬件有一定的了解。

使用 CCS 进行 DSP 编程（二）

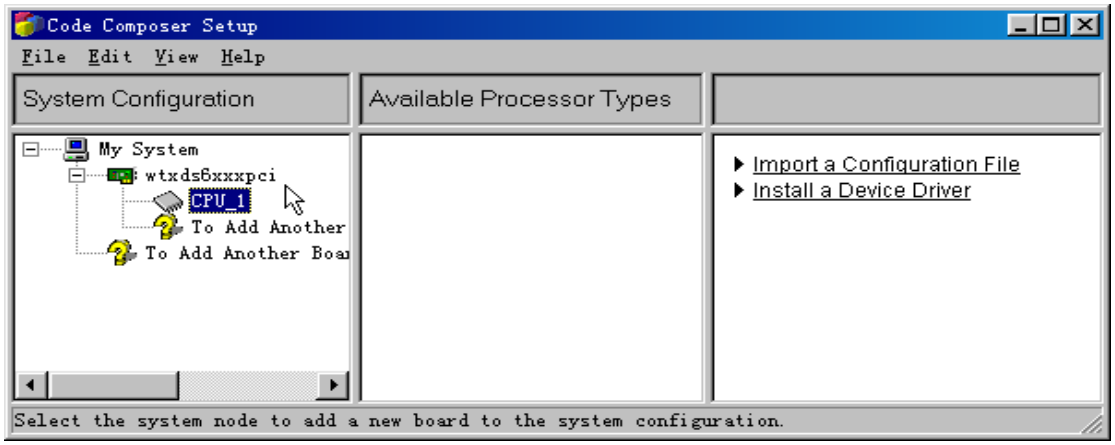
——实现 FFT

pacificxu

现在讨论使用 TI 公司的 CCS 进行 DSP 编程，首先假定读者对 CCS 的使用已经比较了解，如果读者还不太了解，请参阅《使用 CCS 进行 DSP 编程（一）——CCS 编程入门》及其他 CCS 的学习文档。

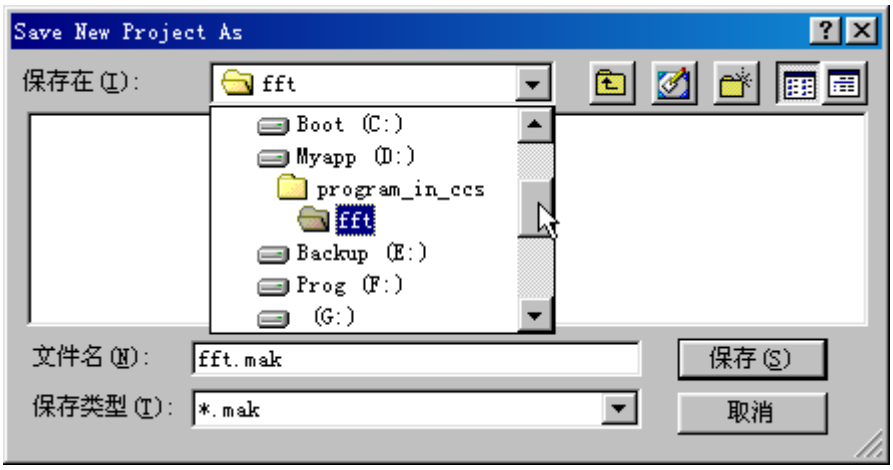
作数字信号处理的同志们总是喜欢用 FFT 来对信号处理系统做检验，下面用闻亭公司的 C6xP 板、C6xPa 板硬件实现 FFT 算法，本算法对其他的 C6x 板同样实用（只是硬件资源稍微有差异），并通过闻亭公司的 PCI 仿真器对目标板加载运行，运行结果在 CCS 中可视化显示。

首先启动 CCS Setup，对仿真器硬件进行设置，本人使用的是闻亭公司 PCI 的仿真器自带的驱动 wtxds6xxxpci.dvr，设置画面如下：

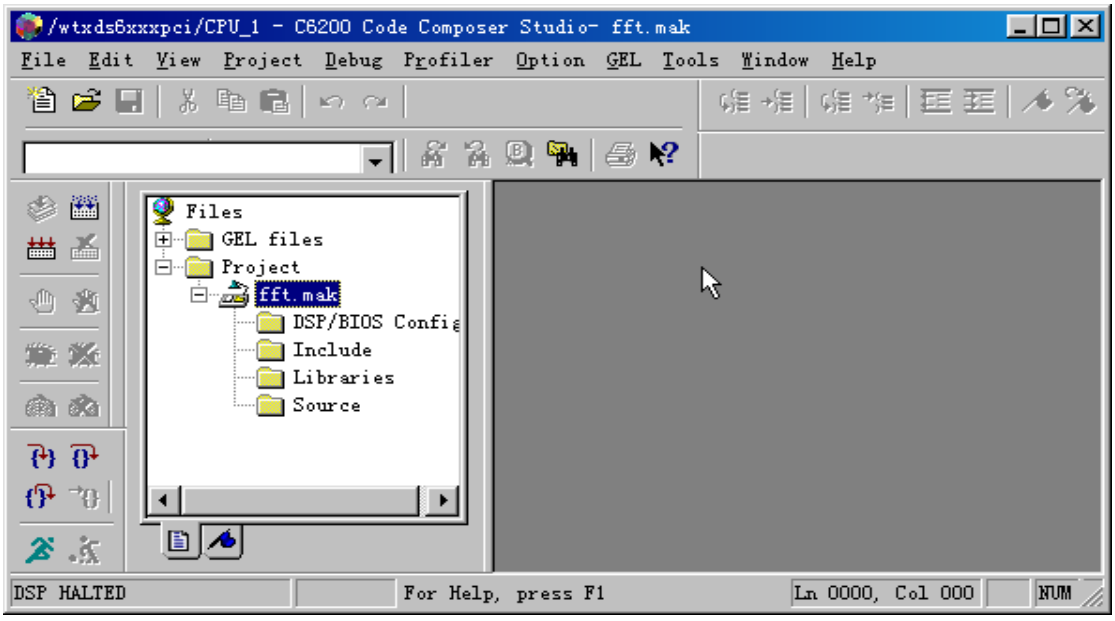


下面就可以运行 CCS 了，在 CCS 中，创建一个新的 Project，

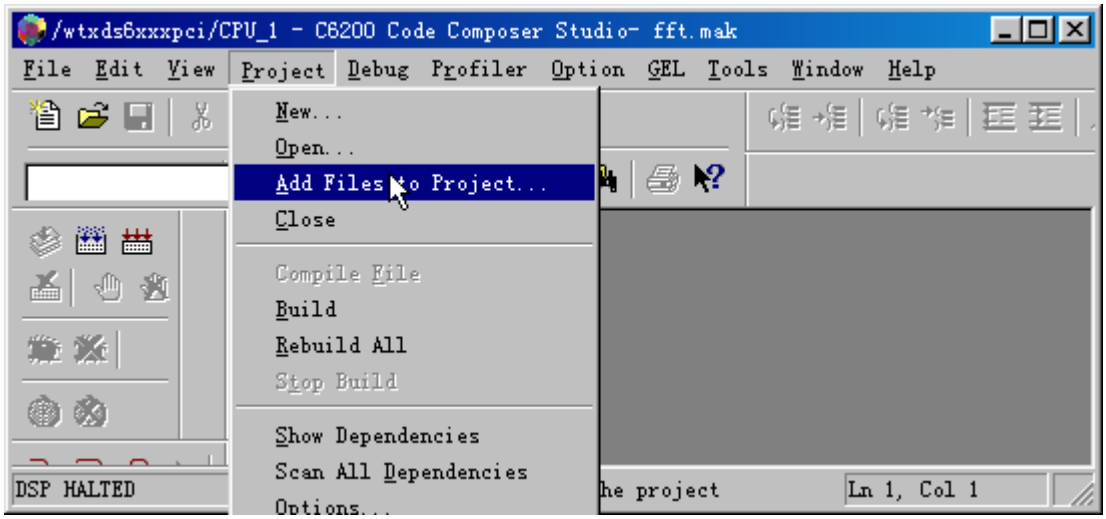
我的工程文件放置在如下的目录中，读者可以放在自己喜欢的目录下：



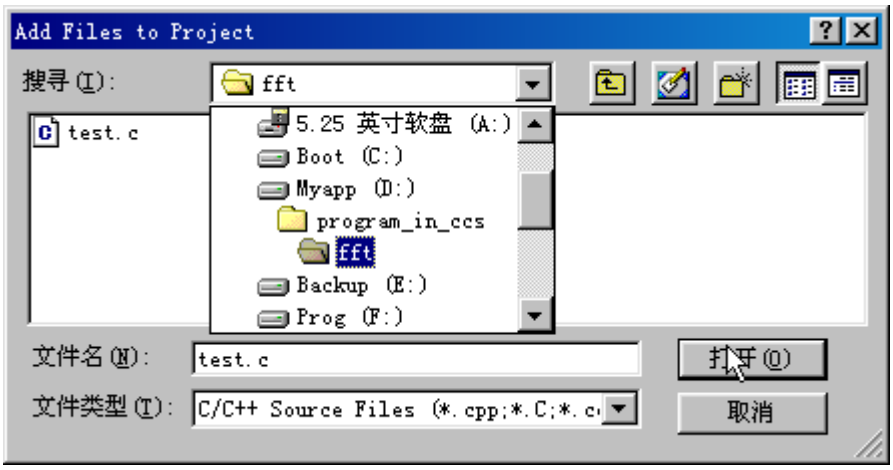
双击“ + ”展开 fft.mak，可以看到整个工程文件是空的，



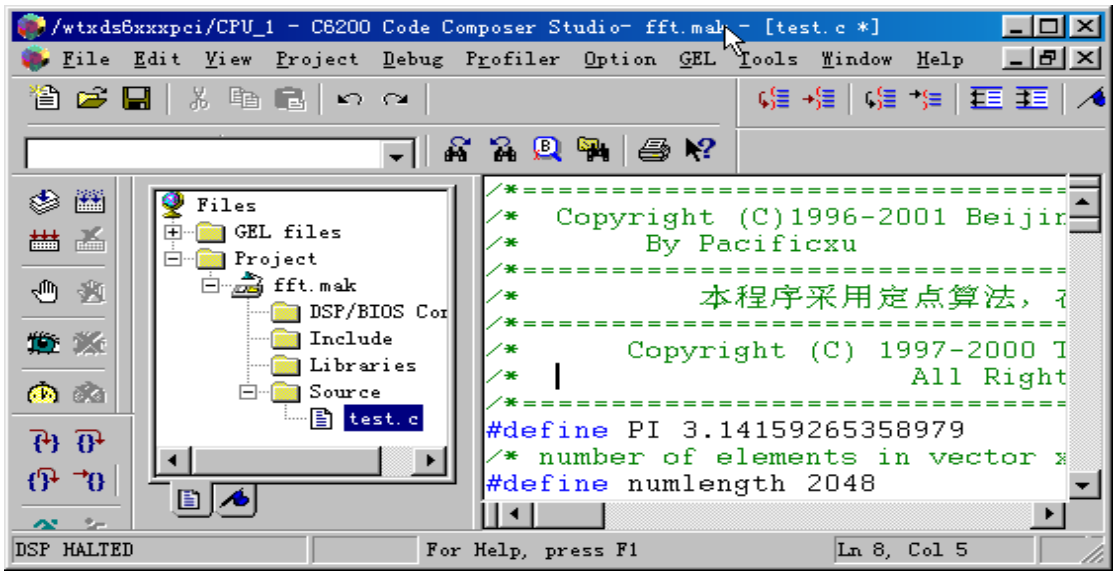
我们需要把*.c、*.cmd、*.lib 文件添加到工程文件中，



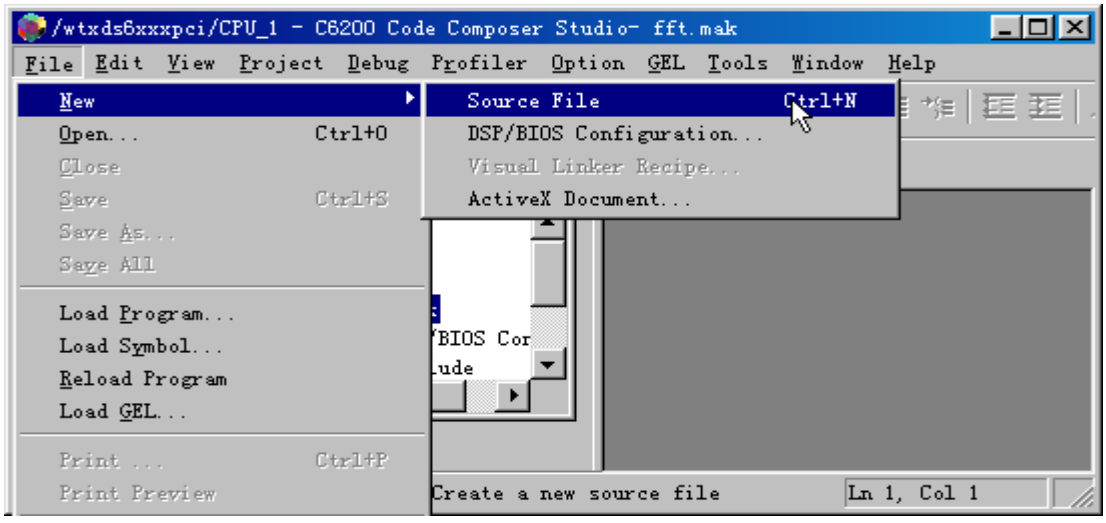
首先是*.c 文件，本例中是 test.c 文件，



同样的方法可以用来添加其他的文件。双击工程中的源文件，会在右边的窗口中看见原码：



如果*.c 文件不存在，可以在 CCS 集成开发环境中生成，



本例中 test.c 的主程序源代码如下：

```
/*============================================================================*/
/*  Copyright (C)1996-2001 Beijing Wintech Technology Co.,Ltd.                */
/*            By Pacificxu,          All Rights Reserved                      */
/*                        本程序采用定点算法，在浮点DSP芯片如6701上同样实用           */
/*============================================================================*/
#define PI 3.14159265358979
/* number of elements in vector x*/
#define numlength 2048
#include <C:\ti\c6000\cgtools\include\math.h>

/*声明函数*/
 void radix2(int n, short *xy, short *w);
 void bitrev_cplx(int *x, short *index, int nx);
 void digitrev_index(short *index, int n, int radix);

/*定义全局变量*/
static short index1[64],w1[numlength];
static short x1[numlength*2];
static int x2[numlength];
static int radix;
int nx1;
int i;

/*主函数开始*/
main()
{

  double delta;

  radix=2;                    /*基数——基2FFT*/
  nx1=numlength;              /*FFT计算点数*/

      /* init the index for fft bitrev */
  digitrev_index(index1,nx1, radix);

      /*init the FFT coefficients*/
  delta=2*PI/nx1;
      for (i=0;i<nx1/2;i++){
              w1[2*i]=32767*(-cos((double)i*delta));
              w1[2*i+1]=32767*(-sin((double)i*delta));
      }

/*构造要做FFT的序列，实部0、虚部0，实部1、虚部1...顺序构造，本例中实部为三个频率的信号，虚部为0*/
  for (i=0;i<nx1;i++){
              x1[2*i]=(short)((cos(PI*i/10.0)+cos(PI*i/20.0)+cos(PI*i/5.0))*0x80);
              x1[2*i+1]=0;
   }

/*做FFT算法*/
  radix2(nx1, x1, w1);
  for (i=0;i<nx1;i++){
              x2[i]=sqrt(x1[2*i]*x1[2*i]+x1[2*i+1]*x1[2*i+1]);
  }


    /*倒序*/
    bitrev_cplx(x2, index1, nx1);

}
```

调用的子程序有三个：

```
/*===========================================================================
/*      Copyright (C) 1997-2000 Texas Instruments Incorporated.
/*                      All Rights Reserved
/*===========================================================================
/*      生成倒序表子程序                                                    */
void digitrev_index(short *index, int n, int radix){
        int i,j,k;
        short nbits, nbot, ntop, ndiff, n2, raddiv2;
        nbits = 0;
        i = n;
        while (i > 1){
                i = i >> 1;
                nbits++;
                }

        raddiv2 = radix >> 1;
        nbot = nbits >> raddiv2;
        nbot = nbot << raddiv2 - 1;
        ndiff = nbits & raddiv2;
        ntop = nbot + ndiff;
        n2 = 1 << ntop;

        index[0] = 0;
        for ( i = 1, j = n2/radix + 1; i < n2 - 1; i++){
                index[i] = j - 1;
                for(k = n2/radix; k*(radix-1) < j;k /= radix)

                        j -= k*(radix-1);

               j += k;
        }
        index[n2 - 1] = n2 - 1;
}

/*           倒序子程序
/*           x    = Input Array to be Bit-Reversed
*           nx   = Number of points in array (must be a power of 2)
*           index = Array of ~sqrt(nx) created by the routine
*                   digitrev_index to allow the fast implementation of the
*                   bit-reversal
*     DESCRIPTION
*           This routine performs the bit-reversal of the input array x[].
*           where x[] is an array of length nx 16-bit complex pairs of data.
*           This requires the index array provided by the program below.
*           This index should be generated at compile time not by the DSP.

void bitrev_cplx(int *x, short *index, int nx)
{
           int     i;
           short       i0, i1, i3;
           short       j0, j1, j3;
           int     xi0, xi1,  xi3;
           int     xj0, xj1, xj3;
```

```
    short       t;
    int     a, b, ia, ib, ibs;
    int     mask;
    int     nbits, nbot, ntop, ndiff, n2, halfn;

    nbits = 0;
    i = nx;
    while (i > 1){
        i = i >> 1;
        nbits++;}

    nbot    = nbits >> 1;
    ndiff   = nbits & 1;
    ntop    = nbot + ndiff;
    n2      = 1 << ntop;
    mask    = n2 - 1;
    halfn   = nx >> 1;

    for (i0 = 0; i0 < halfn; i0 += 2) {
        b   = i0 & mask;
        a   = i0 >> nbot;
        if (!b) ia  = index[a];
        ib  = index[b];
        ibs = ib << nbot;

        j0  = ibs + ia;

        t   = i0 < j0;
        xi0 = x[i0];
        xj0 = x[j0];

        if (t){x[i0] = xj0;
            x[j0] = xi0;}

        i1  = i0 + 1;
        j1  = j0 + halfn;
        xi1 = x[i1];
        xj1 = x[j1];
        x[i1] = xj1;
        x[j1] = xi1;

        i3  = i1 + halfn;
        j3  = j1 + 1;
        xi3 = x[i3];
        xj3 = x[j3];
        if (t){x[i3] = xj3;
            x[j3] = xi3;}
    }
}

/*      频率抽取FFT算法函数体，用户如果对算法感兴趣，请参阅
                胡广书老师的《数字信号处理-理论、算法与实现》 第5章 快速傅立叶变换
/*      xy[] --- input and output sequences (dim-n)      (input/output)
```

```
*        n     --- FFT size                (input)
*        w[]   --- FFT coefficients (dim-n/2)     (input)
*
*
*      DESCRIPTION
*        This routine is used to compute FFT of a complex sequece
*        of size n, a power of 2, with "decimation-in-frequency
*        decomposition" method, ie, the output is in bit-reversed
*        order. Each complex value is with interleaved 16-bit real
*        and imaginary parts.

void radix2(int n, short xy[], short w[])
{
          short n1,n2,ie,ia,i,j,k,l;
          short xt,yt,c,s;

          n2 = n;
          ie = 1;
          for (k=n; k > 1; k = (k >> 1) ) {
            n1 = n2;
            n2 = n2>>1;
            ia = 0;
            for (j=0; j < n2; j++) {
              c = w[2*ia];
              s = w[2*ia+1];
              ia = ia + ie;

              for (i=j; i < n; i += n1) {
                l = i + n2;
                xt       = xy[2*l] - xy[2*i];
                xy[2*i] = xy[2*i] + xy[2*l];
                yt       = xy[2*l+1] - xy[2*i+1];
                xy[2*i+1] = xy[2*i+1] + xy[2*l+1];
                xy[2*l]    = (c*xt + s*yt)>>15;
                xy[2*l+1] = (c*yt - s*xt)>>15;
              }
            }
            ie = ie<<1;
          }
}
```

从上面的源程序可以看到，使用 CCS 的 C 语言编程跟普通的 C 语言编程没有太大的区别，这正是 TI 所追求的，兼容的 ANSI C 标准和如此的编译高效率也正是 TI 的领先之处。读者可以不必学习烦琐的汇编和线性汇编，直接对数字信号处理的算法进行研究，同时享受高速的处理速度，只有在对速度要求极严的条件下，不得不使用汇编和线性汇编，那时读者已经有了一定的基础，再学习汇编语言已是水到渠成。而使用 C 语言编程是大势所趋。

如果有人对算法本身感兴趣，请参阅胡广书老师的《数字信号处理—理论、算法与实现》 第 5 章 快速傅立叶变换，这里不在对算法进行展开讨论。

程序的结构本身很简单，使用过 C 语言的朋友一看就明白，不需要再做进一步的说明，需要指出几点，

1. 本程序中的 math.h 与 Visual C++中的 math.h 是不同的，TI 的 CCS 专门为数学计算作了运行时库，是利用硬件对计算作加速的，与 Visual C++中的速度是不可同日而语的。因此如果我们需要用到相应的“头文件”，就应该在 TI 的目录中查找，同时要包含相应的运行时库（*.lib）文件，我一直在强调这一点，初学者往往忽略这一点而出现许多编译链接错误。下面的演示中还会看到这一点。
2. 本例是以定点 DSP 芯片‘C6201 为例的。如果对定点运算还不太熟悉，只好找些文档来学习一下了，这里也不再展开。在浮点 DSP 芯片‘C6701 中本程序可以不加修改地运行，但浮点 DSP 芯片中可以直接进行有硬件支持的浮点运算，速度会更快。
3. CCS 的 C 语言中的数据类型是与硬件相关的，使用时需要注意。

   char 8 bits

   short 16 bits

   int 32 bits

   long 40 bits

   float 32 bits

   double 64 bits

   TMS320C6000 Online Programmer's Guide (SPRH048) Copyright?2000 Texas Instruments

接下来继续进行，向工程中添加*.cmd 文件。读者现在应该会执行这个操作了，如果没有就自己造一个吧，也可以拿别人的来改造一下。其中有些不太明白也没有关系，但是在与具体的硬件相关的地方还是要搞明白。

首先要搞明白目标板“target”上到底有多少存储空间，包括 DSP 芯片内部有多大空间，目标板上有多少外部存储器（SDRAM、SBSRAM、双口 RAM），以及它们的其始地址和长度，以我使用的闻亭公司‘C6Xpa 板为例，

存储器类型	起始地址	存储器大小		结束地址
SDRAM	0x2000000	4M*32bit	(0x400000*4)	0x3000000
SBSRAM	0x400000	128k*32bit	(0x20000*4)	0x480000
DPRAM	0x1400000	4k*32bit	(0x1000*4)	0x1404000

在这里还要强调一点，是“ *32bit ”，因此 SDRAM 的结束地址是

0x2000000 + 0x400000*4 = 0x3000000

而不是

0x2000000 + 0x400000 = 0x2400000

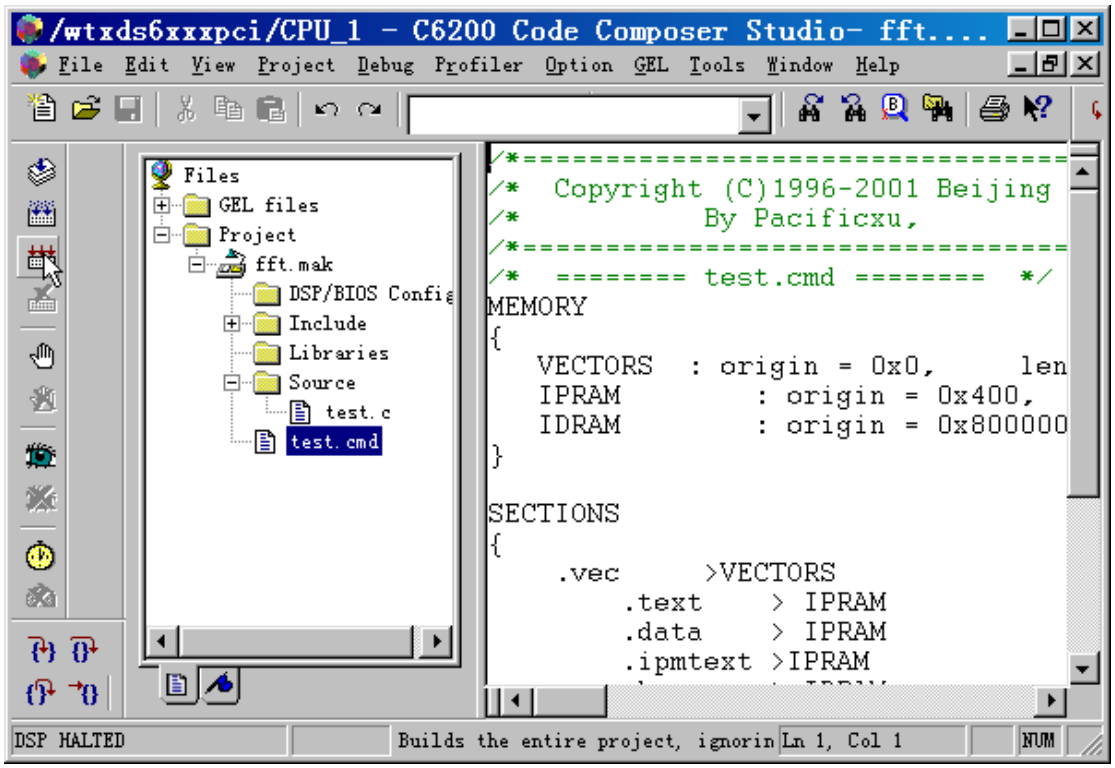
其他存储空间也是如此。许多同志们忽略了这一点，在使用数组、指针及对空间地址操作时造成“ 不可理解 ”的错误，“ 我往么个地址写数怎么找不到，#%^#&^%%^———坏 了！”，好好研究一下，就不好意思这么快下结论了。

下面是我用的*.cmd 文件的源代码，需要的话可以拿去用喔。

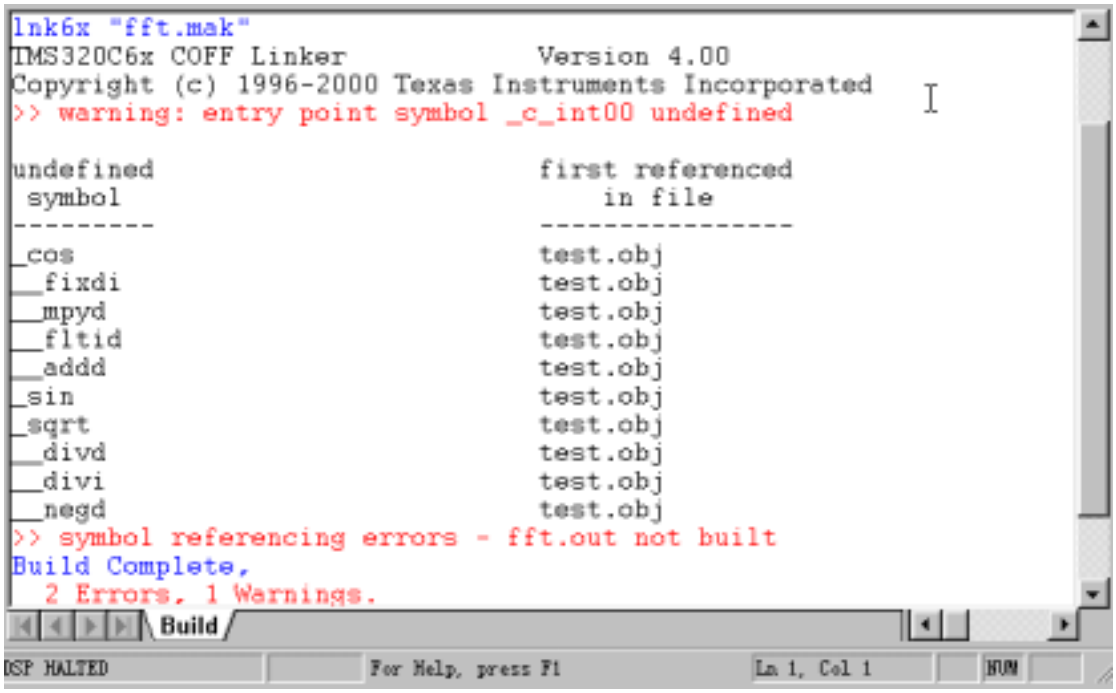
```
/*============================================================
/*  Copyright (C)1996-2001 Beijing Wintech Technology Co.,Ltd.
/*            By Pacificxu,           All Rights Reserved
/*============================================================
/*  ======== test.cmd ========  */
MEMORY
{
   VECTORS      : origin = 0x0,          len = 0x400
   IPRAM        : origin = 0x400,        len = 0xf000
   IDRAM        : origin = 0x80000000,   len = 0x10000
}

SECTIONS
{
        .vec      >VECTORS
        .text     > IPRAM
        .data     > IPRAM
        .ipmtext  >IPRAM
        .bss      > IDRAM
        .cinit    > IDRAM
        .const    > IDRAM
        .far      > IDRAM
        .stack    > IDRAM
        .cio      > IDRAM
        .sysmem   > IDRAM
}
```

里面“ .vec、.text …. ”的东西可以先不考虑，留给 CCS 去处理好了。只要保证定义的空间在物理上存在就可以了。

心急的朋友会开始编译了，



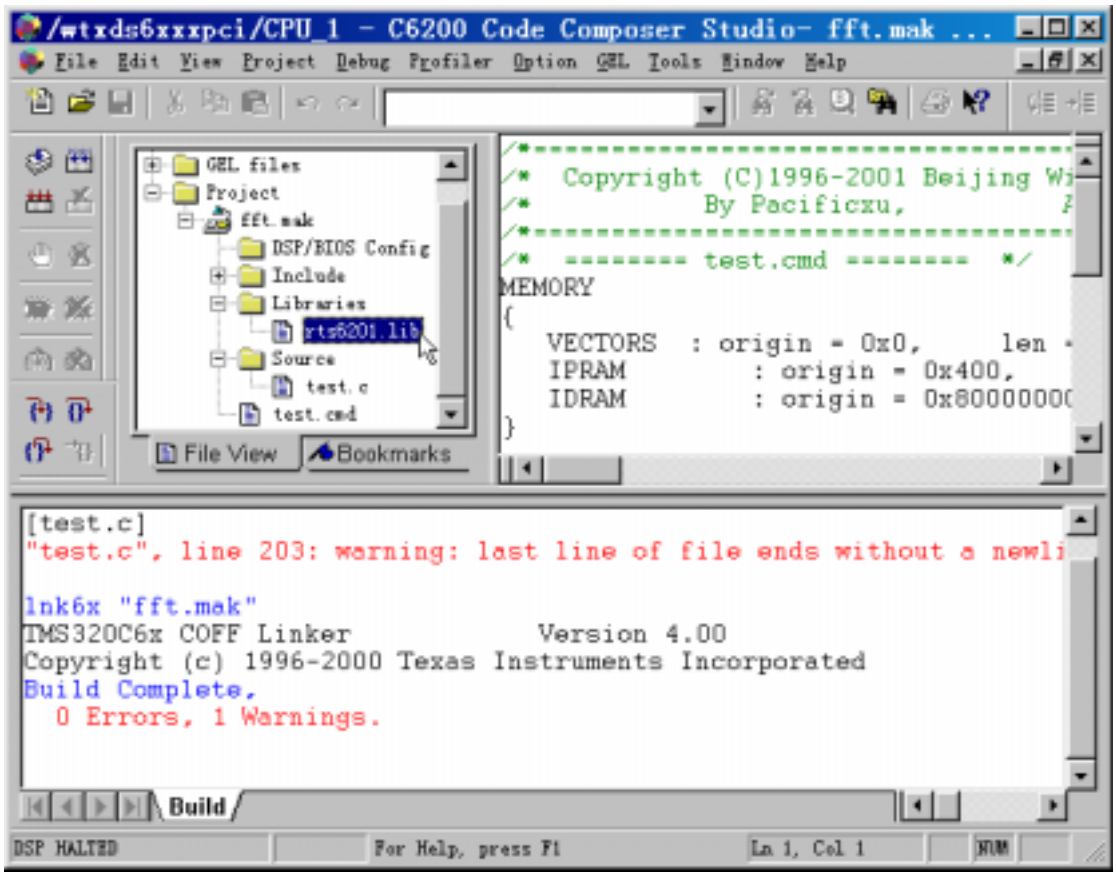
又会出现下面的结果，



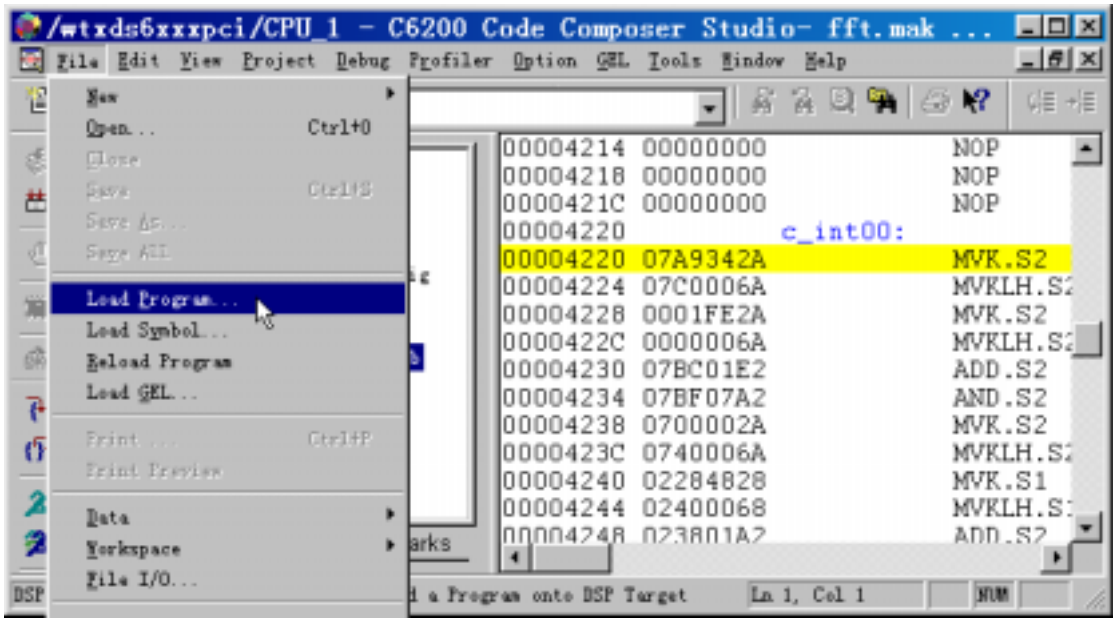
“ $%*^&^，又忘了加运行时库！”。（下次再忘就不应该了 !）。

这里展开一点，有些朋友说：“ 我包含了头文件<intr.h>，调用中断函数intr_hook()时怎么也编译链接通不过，去掉这一句就通过了。”大家现在就应该知

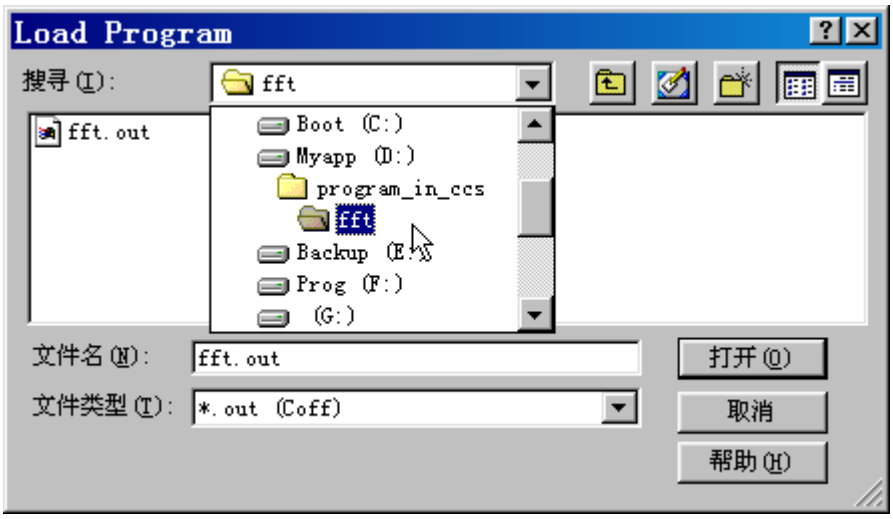
道怎么处理这个问题了吧？熟悉 C 语言编程的同志都知道“ *.lib ”的实质，中断函数 intr_hook()在“ dev6x.lib ”里。加上这个库问题估计该解决了吧！ TI 的运行时库都有具体的说明，如果大家实在懒得去看，又不想知其所以然，有一个绝招在此：“ Find all *.lib in TI 目录 ”一个一个试一下好了。



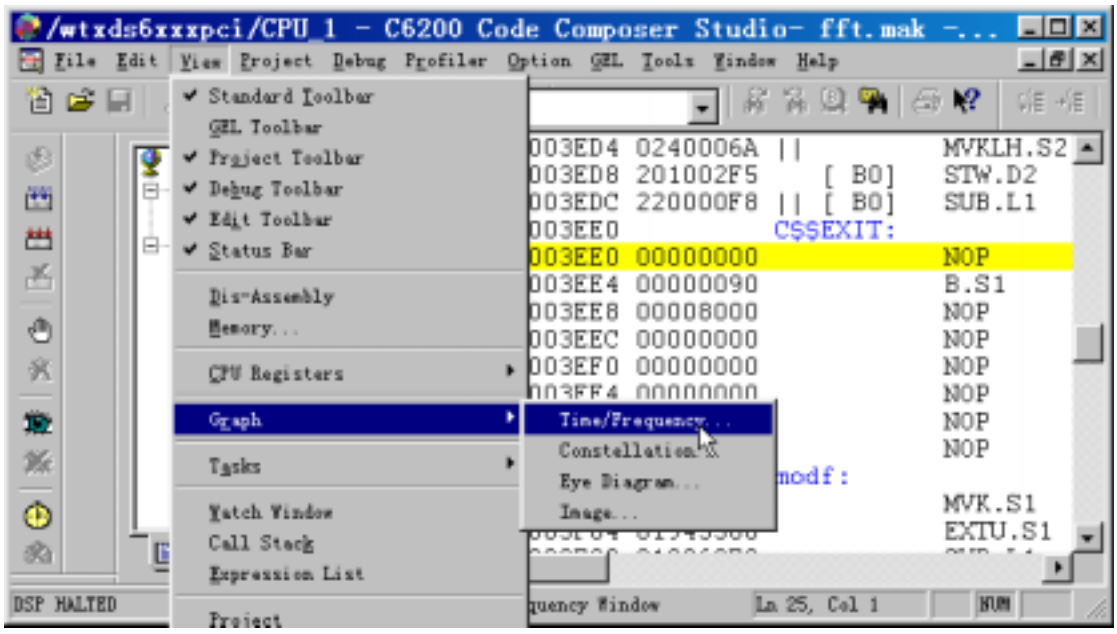
我们的问题解决了，接下来看一看能不能运行，执行对不对。



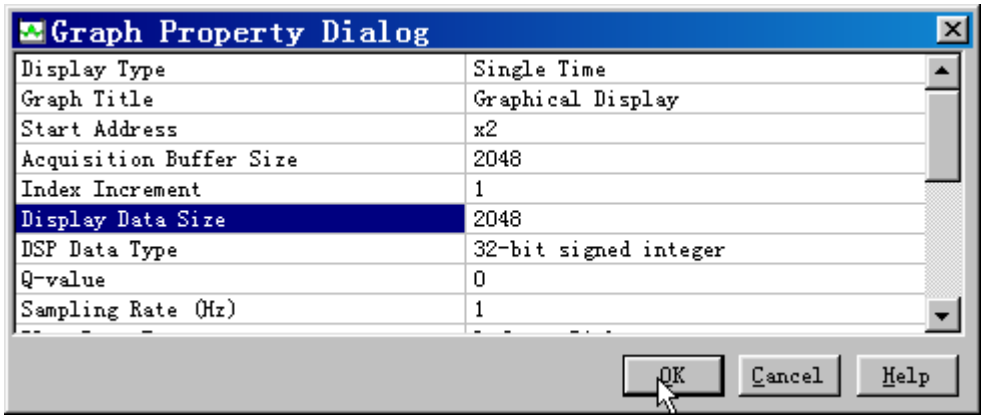
装入“ fft.out ”文件，



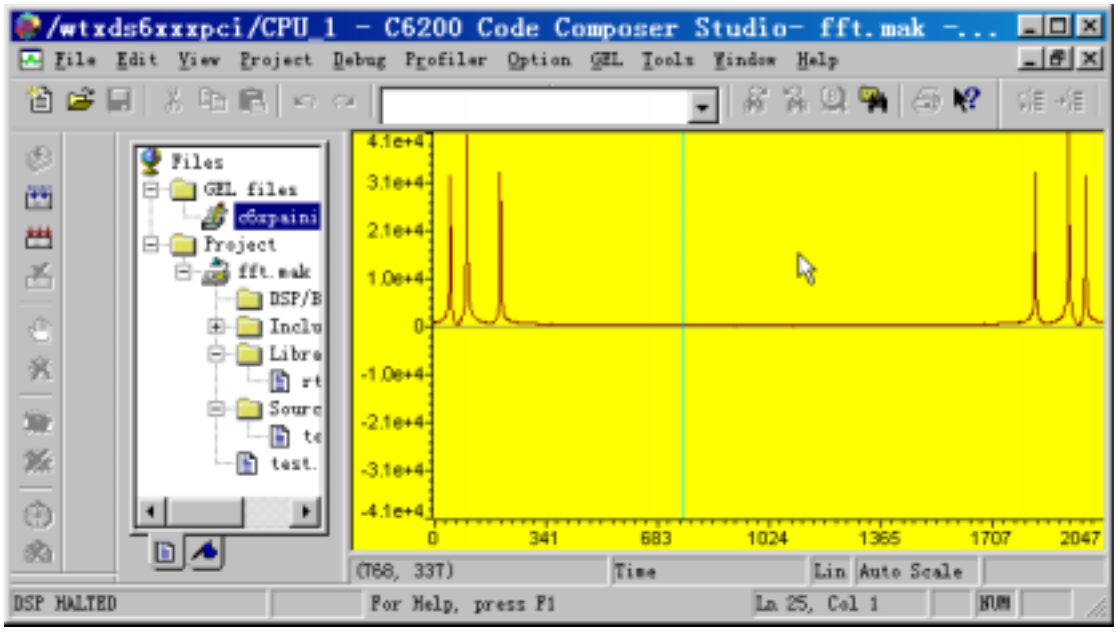
然后运行程序（F5）（热键、工具条、菜单都可以用啦），看一下结果对不对，顺便体验一下 CCS 提供的“ Very Good ”功能：



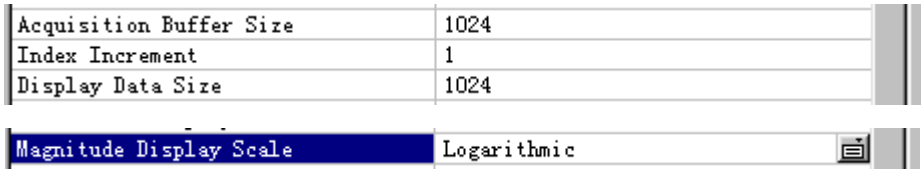
弹出对话框，设置一下吧，



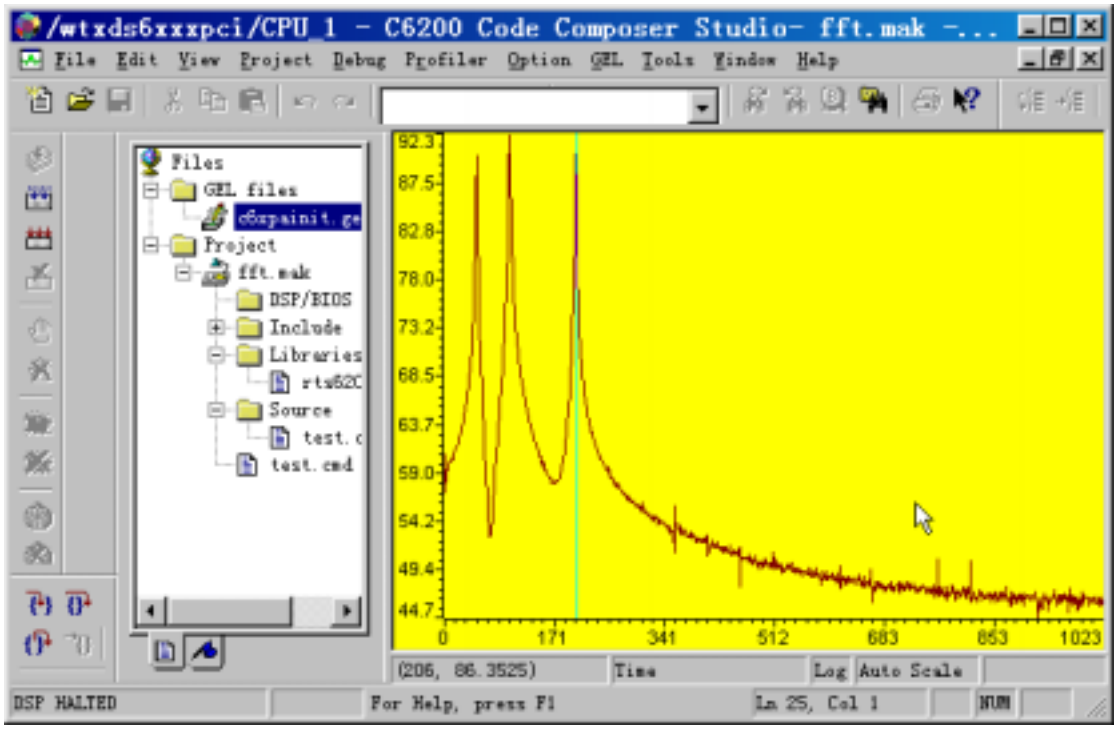
会出现下面的画面，



修改一下属性，



下面好看多了，

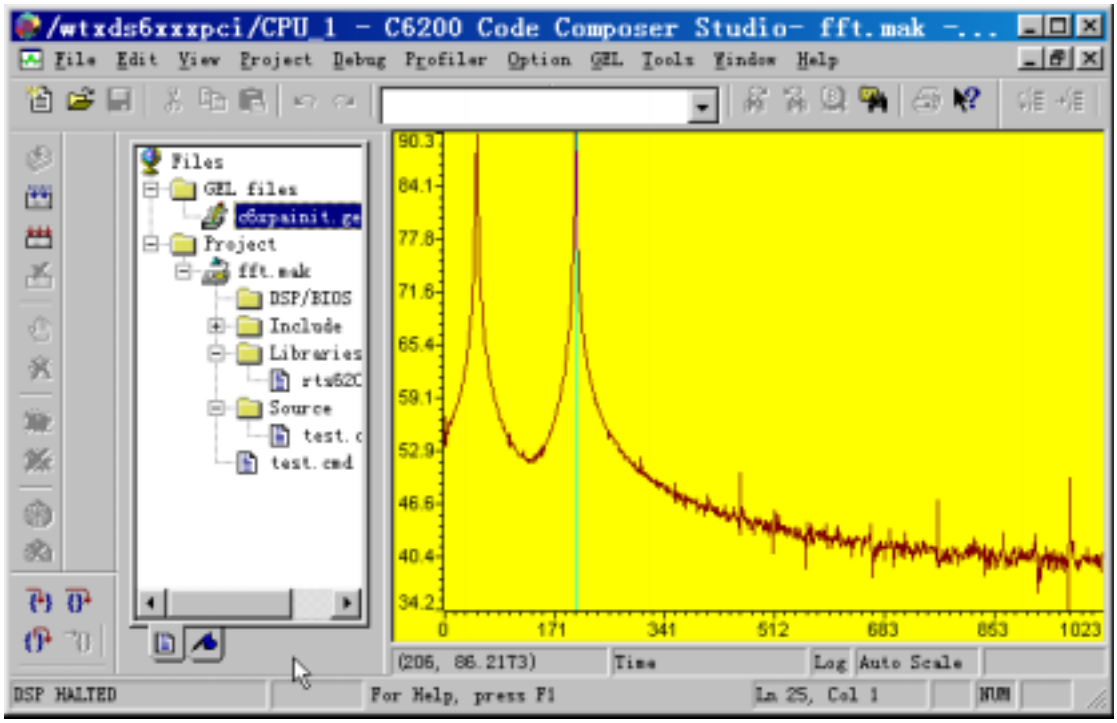


里面的三个谱峰看起来跟信号里造的三个频率的信号好象是对应的吧。在源文件

里进行些修改，例如将信号频率改为两个，

```
/*构造要做FFT的序列，按实部0、虚部0，实部1、虚部1...顺序构造，本例中实部为三
  for (i=0;i<nx1;i++){
                x1[2*i]=(short)((cos(PI*i/20.0)+cos(PI*i/5.0))*0x80);
                x1[2*i+1]=0;
   }
```

重新编译、链接、执行、显示，中间的谱线不见了。



好象我们成功了！(哗哗哗哗哗--------)

总结：在这里演示了使用 CCS 的 C 语言实现 FFT 程序，仅仅是实现而已。一个实际的系统，需要做许多优化处理，对核心算法进行详细的分析、规划，初学时可以先不考虑这些，等熟悉后在进行深入研究。

使用 CCS 进行 DSP 编程（三）

——实现 DMA 和 Interrupt

pacificxu

现在讨论在 CCS 进行 DSP 编程来实现 **DMA** 和 **Interrupt** 功能。假定读者对 CCS 的使用已经比较了解，并有了一定的 CCS 编程经验。如果读者还不太了解，请参阅《使用 CCS 进行 DSP 编程（一）——CCS 编程入门》、《使用 CCS 进行 DSP 编程（二）——实现 FFT》及其他 CCS 的学习文档。

下面用闻亭公司的 C6xPa 板硬件和闻亭公司的 PCI 仿真器为例，来实现 DSP 的 **DMA** 传输和硬件 **Interrupt** 功能。

首先来描述一下使用的硬件资源。闻亭公司的 C6xPa 板有两路独立的最高采样率为 40MHz 精度为 12bit 的 A/D，它与 DSP 的 EXT_INT7 相连，可以产生外部中断信号，通过 FPGA 的逻辑可以控制 A/D 的采集和采集多少数据产生一次中断，采集的数据放在 DPRAM 中（0x1400000 开始的地址空间），通过 **DMA** 传输到 DSP 芯片上的存储器中（0x80000000 开始的地址空间）。

在 C 语言环境中使用 **DMA** 和 **Interrupt** 功能，需要包含两个头文件<dma.h>和<intr.h>，同时要用到相应的运行时库文件“ csl6201.lib ”和“ dev6x.lib ”。对这两组头文件和运行时库文件，我们深入研究一下，看一看我们比较关心的函数有哪些。下一次用到这些函数时，别忘了带上相应的运行时库文件%*&^*&^喔。

在 dev6x.lib 库文件中，直接与实现 **DMA** 和 **Interrupt** 功能相关的函数有如下几个：

dma_init

dma_global_init

dma_reset

intr_reset

intr_init

intr_hook

intr_map

intr_isn

intr_get_cpu_intr

isr_jump_table

在csl6201.lib库文件中，直接与实现**DMA**功能相关的函数有如下几个：

DMA_AllocGlobalReg

DMA_GetEventId

DMA_GBL_PRIVATE

DMA_Open

DMA_Start

DMA_HCHA0

DMA_HCHA1

DMA_HCHA2

DMA_HCHA3

DMA_Wait

DMA_SetGlobalReg

DMA_ConfigA

DMA_ConfigB

DMA_Stop

DMA_AutoStart

DMA_Pause

DMA_Reset

DMA_GetGlobalReg

DMA_SetAuxCtl

DMA_Close

DMA_FreeGlobalReg

DMA_Init

DMA_GetStatus

我们只需其中的一部分便可以实现 **DMA** 和 **Interrupt** 功能。函数的具体使用说明和调用方式请参看 TI 的说明档。

下面直接看一下程序的源码，有一个整体的理解，然后再对具体的相关内容做一下说明，如果源码能够理解，就不需要再看具体说明了。为了便于读者理解，

本人对相应部分作了处理，尽量使结构清晰，突出对 **DMA** 和 **Interrupt** 功能的实现：

```
/*============================================================
/*  Copyright (C)1996-2001 Beijing Wintech Technology Co.,Ltd.
/*            By Pacificxu,           All Rights Reserved
/*============================================================

#ifndef CHIP_6201
#define CHIP_6201       1
#endif

/* includes */
#include "intr.h"
/* Interrupts Support - 'C6x peripheral support library*/
#include "dma.h"

/*DMA控制参数说明*/
DMA_CONFIG MyConfig1 = {
                        0x01000051, /* prictl */
                        0x00000000, /* secctl */
                        0x01400000, /* src */
                        0x80008000, /* dst */
                        0x00000400 /* xfrcnt */
                        };
/*函数定义*/

/*初始化中断子程序*/
void Init_int7();


 /*中断服务子程序*/
 void interrupt  Int7_ISR();

 /*调用DMA子程序*/
 void DMA_read_data1();

 /* variables definitions */

 /* DMA操作句柄*/
 static DMA_HANDLE hDma;


 main()
 {
         /*初始化中断*/
      Init_int7();

 /*通过FPGA对A/D采集进行控制，采样率40MHz，采集2k数据产生一次中断*/
      *(int *)0x3380000=0x1f;

         /*等待中断*/
        for(;;) ;

 }
```

```
/*中断服务子程序*/
void interrupt Int7_ISR()
{
        DMA_read_data1();
}

/*调用DMA子程序*/
void DMA_read_data1()
{
        hDma=DMA_Open(DMA_CHAANY,DMA_OPEN_RESET);
        DMA_ConfigA(hDma,&MyConfig1);
        DMA_Wait(hDma);
        DMA_Close(hDma);

}

/*初始化中断子程序*/
void Init_int7()
{
 /* Interrupts settings */
        intr_init();
/* it initializes the ISTP with the address of the global label
// vec_table, which is defined in intr_.asm, and resolved at link
// time. Defined in intr.c as a callable function, intr.c is a
// included with the 'C6x peripheral support library and should be
// compiled and linked with the rest of the program files
*/
        intr_map(CPU_INT7,ISN_EXT_INT7);
/* it places the indicated Interrupt Service Number
// (ISN) value in the appropriate field of the
// appropriate interrupt multiplexer register. Defined in
// intr.c as a callable function
*/
        INTR_CLR_FLAG(CPU_INT7);
/* it manually clears the selected interrupt by writing
// a 1 to the specified bit in the ICR. This is just to
// be sure that there's no unwanted/unexpected
// data in any of the bit fields of this register.
// Defined in intr.h as a macro . Even though
// this is not absolutely necessary, it is highly recommended.
*/
        intr_hook(Int7_ISR,CPU_INT7);

/* it places the function pointer indicated by the first
// parameter (a pointer to an ISR declared in C) into
// isr_jump_table[], at the location specified by the
// second parameter (ISR to invoke when servicing
// this interrupt).
*/
        INTR_ENABLE(CPU_INT_NMI);
/* it enables the non-maskable interrupt (NMI). If this
// interrupt is not enabled, the rest of the interrupts
// won't be seen/processed. Defined in intr.h as a macro.
*/
```

```
        INTR_ENABLE(CPU_INT7);
/* it enables CPU interrupt 14 by enabling its bit in the
// the interrupt enable register (IER).We have previously
// mapped this interrupt number with the cpu clock 0
// interrupt signal. Defined in intr.h as a macro.
*/
        INTR_GLOBAL_ENABLE();
/* it globally enables all maskable interrupts by setting the
// GIE bit in the control status register (CSR). If this bit is
// not enabled/set, the rest of the interrupts won't be seen
// nor processed. Defined in intr.h as a macro
*/
 }
```

接下来进行详细说明。

首先，包含头文件，

```
/* includes */
#include "intr.h"
/* Interrupts Support - 'C6x peripheral support library*/
#include "dma.h"
```

接着对DMA进行初始化赋值，

```
/*DMA控制参数说明*/
DMA_CONFIG MyConfig1 = {
                        0x01000051, /* prictl */
                        0x00000000, /* secctl */
                        0x01400000, /* src */
                        0x80008000, /* dst */
                        0x00000400 /* xfrcnt */
                        };
```

其中，各参数的含义如下：

UINT32 prictl	DMA primary control register value
UINT32 secctl	DMA secondary control register value
UINT32 src	DMA source address register value
UINT32 dst	DMA destination address register value
UINT32 xfrcnt	DMA transfer count register value

我们使用 DMA 的源地址为 0x01400000，目标地址为 0x80008000，传输数据长度为 1k*32bit，DMA 控制寄存器的值的具体含义请参看各对应 DSP 的 datasheet。

然后是对使用的函数进行引用说明，其他函数跟普通的 Visual C++函数没有太大的区别，除了以下的例外：

```
/*中断服务子程序*/
void interrupt  Int7_ISR();
```

其中，有一个关键字“interrupt”，它告诉 TI 的 C 编译器，这个函数是一个特殊

的中断服务函数，C 编译器会另眼看待。

接下来对 DMA 操作句柄定义，相应的头文件对 DMA_HANDLE 作了定义，我们这里可以直接使用：

```
/* DMA操作句柄*/
static DMA_HANDLE hDma;
```

主程序非常简单，

```
main()
{
        /*初始化中断*/
     Init_int7();

/*通过FPGA对A/D采集进行控制，采样率40MHz，采集2k数据产生一次中断*/
     *(int *)0x3380000=0x1f;

        /*等待中断*/
       for(;;) ;

}
```

实现 DMA 功能的子程序也很简单，只有四句：

```
/*调用DMA子程序*/
void DMA_read_data1()
{
        hDma=DMA_Open(DMA_CHAANY,DMA_OPEN_RESET);
        DMA_ConfigA(hDma,&MyConfig1);
        DMA_Wait(hDma);
        DMA_Close(hDma);

}
```

第一句：语法如下：

Function

```
DMA_HANDLE DMA_Open(
  int ChaNum,
  UINT32 Flags
);
```

Arguments

ChaNum	DMA channel to open: • DMA_CHAANY • DMA_CHA0 • DMA_CHA1 • DMA_CHA2 • DMA_CHA3
Flags	Open flags (logical OR of any of the following): • DMA_OPEN_RESET

Return Value

Device Handle	Handle to newly opened device

其中的 DMA_CHAANY 含义是“任意一个闲置的 DMA 通道”，并非有一个 DMA

通道叫 DMA_CHAANY，除了通道 0~3 的 4 个通道外，还有一个辅助 DMA 通道，是另做他用的。我们取得了对 DMA 操作的句柄，就可以进行初始化和使用了。

第二句：语法如下：

Function	`void DMA_ConfigA(`
	`  DMA_HANDLE hDma,`
	`  DMA_CONFIG *Config`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
	`Config`	Pointer to an initialized configuration structure
Return Value	`None`	

这里需要提前做 `DMA_CONFIG` 的初始化，这两步也可以用一步来实现，就用到另外一个函数：

Function	`void DMA_ConfigB(`
	`  DMA_HANDLE hDma,`
	`  UINT32 prictl,`
	`  UINT32 secctl,`
	`  UINT32 src,`
	`  UINT32 dst,`
	`  UINT32 xfrcnt`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
	`prictl`	Primary control register value
	`secctl`	Secondary control register value
	`src`	Source address register value
	`dst`	Destination address register value
	`xfrcnt`	Transfer count register value
Return Value	`none`	

它直接把各 DMA 寄存器的设置当作参数，一步到位。

第三句：语法如下：

Function	`void DMA_Wait(`
	`  DMA_HANDLE hDma`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
Return Value	`none`	

这个函数检测 DMA 的状态位，直到 DMA 结束后才退出，读者可以在下一次使用这个 DMA 通道前使用。DSP 可以执行其他的操作，或者执行此操作等待 DMA 传输结束。

第四句：语法如下：

Function	`void DMA_Close(`
	`  DMA_HANDLE hDma`
	`);`	

Arguments	`hDma`	Handle to DMA channel, see `DMA_Open()`
Return Value	`none`	

使用完 DMA 通道后，需要对它进行关闭，释放资源以备他用。

DMA 的使用是很简单的，复杂的工作都由 DSP 硬件和 TI 的库函数来完成了。我们要做的工作是理解这些，要想使用这些函数，不可避免要知道 DMA 各控制寄存器的具体含义，除了少数“ 天才 ”可以不学而知外，最好老老实实学习 TI 的文档）。#$&^@)(*&#——多么渴望天才的出现啊 !!!

下面来看看中断功能是怎样实现的。在中断服务子程序中，调用了 DMA，完成数据的从外部双口 RAM 到 DSP 片内的传输，

```
/*中断服务子程序*/
void interrupt Int7_ISR()
{
        DMA_read_data1();
}
```

而中断服务子程序是如何与硬件中断联系起来的呢？

中断的实现主要在中断初始化子程序里，在程序中对每一步的操作都进行了详细的说明。我们一步一步来分析一下：

首先是用于中断处理的函数：

Interrupt processing functions include the following:

- intr_get_cpu_intr(isn)
- intr_hook(void(*fp)(void),cpu_intr)
- intr_init()
- intr_isn(cpu_intr)
- intr_map(cpu_intr,isn)
- intr_reset()

The value *fp is a function pointer to the user supplied ISR, cpu_intr refers to the CPU interrupt number, and isn refers to the interrupt selection number that specifies the interrupt source to map to a given CPU interrupt.

然后是用于中断处理的宏定义：

In the following list of interrupt processing macros, bit refers to the bit position on which to operate, val refers to the field value, and sel is 0 for the low interrupt selector register and non-zero for the high interrupt selector register.

- INTR_CHECK_FLAG(bit)
- INTR_CLR_FLAG(bit)
- INTR_DISABLE(bit)
- INTR_ENABLE(bit)
- INTR_EXT_POLARITY(bit,val)
- INTR_GET_ISN(intsel,sel)
- INTR_GLOBAL_DISABLE( )
- INTR_GLOBAL_ENABLE( )
- INTR_MAP_RESET( )
- INTR_SET_FLAG(bit)
- INTR_SET_MAP(intsel,val,sel)

用到的一些助记符如下：

CPU Interrupt Numbers

Mnemonic	Value
CPU_INT_RST	0
CPU_INT_NMI	1
CPU_INT_RSV1	2
CPU_INT_RSV2	3
CPU_INT4	4
CPU_INT5	5
CPU_INT6	6
CPU_INT7	7
CPU_INT8	8
CPU_INT9	9
CPU_INT10	10
CPU_INT11	11
CPU_INT12	12
CPU_INT13	13
CPU_INT14	14
CPU_INT15	15

Interrupt Selection Numbers

Mnemonic	Value
ISN_DSPINT	0
ISN_TINT0	1
ISN_TINT1	2
ISN_SD_INT	3
ISN_EXT_INT4	4
ISN_EXT_INT5	5
ISN_EXT_INT6	6
ISN_EXT_INT7	7
ISN_DMA_INT0	8
ISN_DMA_INT1	9
ISN_DMA_INT2	10
ISN_DMA_INT3	11
ISN_XINT0	12
ISN_RINT0	13
ISN_XINT1	14
ISN_RINT1	15

大家觉得上面的内容还不够丰富，请参阅 TI 的文档 spru273b.pdf。

有了上面的理解，中断的实现过程就比较清楚了：

1. 初始化中断服务表指针（ISTP）：intr_init();
2. 选择用哪一个中断：　　intr_map(CPU_INT7,ISN_EXT_INT7);
3. 清中断：　　INTR_CLR_FLAG(CPU_INT7);
4. 中断服务子程序与中断号挂钩：　　intr_hook(**Int7_ISR**,CPU_INT7);
5. 打开非屏蔽中断：　　INTR_ENABLE(CPU_INT_NMI);
6. 打开所选中断：　　INTR_ENABLE(CPU_INT7);
7. 全局中断使能：　　INTR_GLOBAL_ENABLE();

读者可能会注意到，中断处理函数都是小写的，而宏定义都是大写的，在 C 语言的语法里是要注意的，否则会出现找不到函数或者函数未定义。

如果想用其他的中断源，可以按照上面的步骤依样而行，相信不会是什么太难的事情了。每一步不是必须的，顺序也不是固定的。**Int7_ISR** 是本人举例时用的中断服务子程序名，大家可以使用任意的名字，而其中的程序也是随意根据需要编写，没有太多的限制。读者如果对 DSP 的硬件很清楚，可以直接对中断寄存器进行赋值，不需要调用这些函数与宏定义。

相信现在大家对中断与 DMA 的实现已经心中有数了，但我还要强调一下，我所讲的实现是突出软件上的实现，进行 DSP 编程需要对硬件有足够必要的了解，否则会遇到某些难以理解的问题，我在这里尽量不涉及硬件，只是希望大家仔细对 TI 有关资料认真研究，避免我的介绍产生先入为主的不良影响。例如我在 DMA 中执行了对双口 RAM 的读操作，而双口 RAM 是连接在 EMIF 上的，因此，进行读操作之前就必须对 EMIF 进行初始化操作，否则，出错是必然的，而且很难找出错误原因，切记切记。

使用 CCS 进行 DSP 编程（四）

——实现 Host 和 DSP 通信

pacificxu

首先对题目进行一下解释，之所以取这个名字，是为了与前面三篇文章相对应，连成一个系列，这里不仅仅涉及使用 CCS 进行 DSP 编程，主机端的程序便是用 Visual C++实现的。通信包括许多手段：中断、mailbox、直接数据传输等等，这里并不一一列举。

现在讨论实现 **Host** 和 **DSP** 通信。假定读者对 CCS 的使用已经比较了解，并有了一定的 CCS 编程经验。如果读者还不太了解，请参阅《使用 CCS 进行 DSP 编程（一）——CCS 编程入门》、《使用 CCS 进行 DSP 编程（二）——实现 FFT》、《使用 CCS 进行 DSP 编程（三）——实现 DMA 和 Interrupt》及其他 CCS 的学习文档。

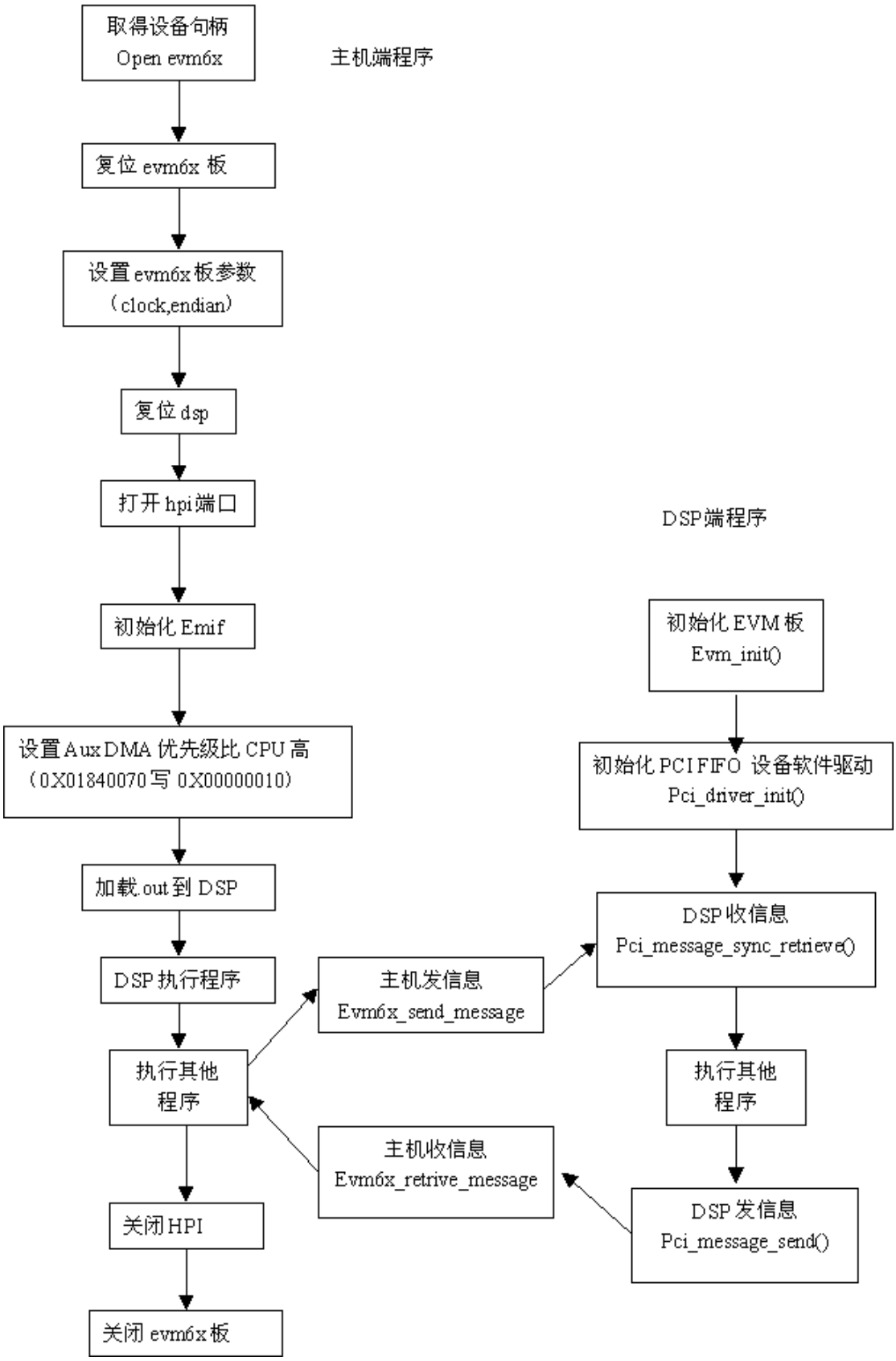
下面用闻亭公司的 C6xP 板硬件和闻亭公司的 PCI 仿真器为例，来实现 **Host** 和 **DSP** 通信。对于‘ C6Xpa 板同样有效。

闻亭公司的 C6xP 板是一款具有 PCI 接口的高速信号处理 EVM 板，接口芯片是 AMCC 的 S5933，兼容 PCI Local Bus Revision 2.1 协议。PCI 接口比较适合用来进行 Host 和 DSP 的高速大数据量数据交换。主机通过 HPI 接口可直接访问 DSP 的所有存储空间，允许主机初始化 DSP，可以从主机加载程序。

前面几篇文章所讲的都是从 JTAG 接口加载程序，这样比较适合于程序的开发调试，对于实际的系统来说，大部分都是系统自己从 EEPROM 或 Flash 加载，现在我们可以从主机通过应用程序来加载，基于此，许多耗时的算法 PC 机不能实时完成的可以由 DSP 来完成。

这个过程可以这样来描述：PC 机执行应用程序，加载算法到 DSP 端，并将需要处理的数据传送到 DSP，DSP 计算完成后将数据传回 PC，整个过程由 PC 来控制启动、工作、完成，使用起来比较方便。当然，DSP 算法还需要首先用仿真器通过 JTAG 接口调试好才行。

接下来看看实现这个功能的一个典型系统框图：



在这个框图里，我简化了主机 PC 执行程序的其他部分，突出了与 DSP 进行通信有关的内容。

对任何一个系统，复位状态必须是确定的，这样才有一个确定的前提，对硬件电路如此，对于软件编程也是如此。因为这里的关于主机与 DSP 通信是针对确定物理硬件的（闻亭公司‘ C6Xp 板和‘ C6Xpa 板），编程是建立在对应的支持库上的，未使用闻亭公司‘ C6Xp 板或‘ C6Xpa 板的读者可能对一些地方不太理解，但原理性的地方应该是一致的。

对于 PCI 插卡与操作系统的关系，我不做过多的说明。闻亭公司‘ C6Xp 板和‘ C6Xpa 板可以看作标准的 PCI 插卡，“ 驱动级 ”有两个文件支持， evm6x.vxd（对于 NT4.0 是 evm6x.sys）和 evm6x.dll，我们所关心的是 evm6x.dll，它提供了类似于 WIN32 的 API 一样的函数接口，用户可以直接在 Visual C++和 C++ Builder 下调用。

With the provided low-level driver and user mode DLL, a user application on the host can:

- Reset and configure the 'C62x
- Load and execute code
- Send and receive messages
- Send and receive data streams
- Access board resources via the HPI

在应用程序中，需要包含头文件 evm6xdll.h，在这个头文件中，包含了对以下函数的定义：

Function	Description
evm6x_abort_read	Terminate the current read request
evm6x_abort_write	Terminate the current write request
evm6x_board_type	Return board type
evm6x_clear_message_event	Clear the incoming message event
evm6x_close	Close a driver connection to a board
evm6x_coff_display	Display COFF section information
evm6x_coff_load	Load a COFF image into DSP memory
evm6x_cpld_read_all	Read the contents of the CPLD registers
evm6x_dll_revision	Read EVM DLL Revision Information
evm6x_generate_nmi_int	Generate an NMI to a DSP
evm6x_hpi_close	Close the HPI for a DSP
evm6x_hpi_fill	Fill DSP memory using the HPI

evm6x_hpi_generate_int	Interrupt a DSP using the HPI
evm6x_hpi_open	Open the HPI for a DSP
evm6x_hpi_read	Read DSP memory using the HPI
evm6x_hpi_write	Write DSP memory using the HPI
evm6x_init_emif	Initialize the EMIF registers
evm6x_mailbox_read	Read from an EVM mailbox
evm6x_mailbox_write	Write to an EVM mailbox
evm6x_nvram_read	Read NVRAM memory on a board
evm6x_nvram_write	Write NVRAM memory on a board
evm6x_open	Open a driver connection to a board
evm6x_read	Read data from a boards PCI interface
evm6x_read_single	Read a single byte, short or long, using the HPI
evm6x_reset_board	Completely reset a board
evm6x_reset_dsp	Reset the DSP on a board and set boot mode
evm6x_retrieve_message	Get a message sent by a DSP
evm6x_send_message	Send a message to a DSP
evm6x_set_board_config	Set the board configuration settings
evm6x_set_timeout	Set the data transfer time out value
evm6x_unreset_dsp	Release the DSP from its reset state
evm6x_user_semaphore_get	Acquire user semaphore
evm6x_user_semaphore_release	Release user semaphore
evm6x_user_semaphore_wait	Wait until semaphore available
evm6x_write	Write data to a boards PCI interface
evm6x_hpi_write_single	Write a single byte, short or long, using the HPI

读者可以对**比较感兴趣的函数**做进一步的了解，这些函数的参数和使用方法在 TI 的文档（spru308.pdf）中有详细的说明，我就不一一说明。顺便说一下，如果读者同时有几块类似的 PCI 插卡，可以通过使用不同的板卡索引分别对不同的板卡进行操作：

```
#include <evm6xdll.h>
HANDLE evm6x_open(
        int     board_index,
        BOOL    exclusive_flag);
```

对第一块卡，*board_index* 为 0，第二块卡，*board_index* 为 1，……，对每一块确定的卡都有相应的操作句柄对应，相互之间互相不影响。

在 DSP 端的 CCS 编程中，同样会用到相关的头文件：**board.h** 和 **pci.h**，以及相关的运行时库为 **drv6x.lib**。其中与主机交换信息的函数列表如下：

Function	Description
pci_driver_int(void)	Initializes the PCI Driver
pci_fifo_open	Opens the driver for the PCI FIFO device
pci_fifo_close	Closes the driver for the PCI FIFO device
pci_fifo_async_send	Starts an asynchronous PCI FIFO send operation
pci_fifo_sync_send	Starts a synchronous PCI FIFO send operation
pci_fifo_async_receive	Starts an asynchronous PCI FIFO receive operation
pci_fifo_sync_receive	Starts a synchronous PCI FIFO receive operation
pci_message_send	Sends a message immediately
pci_message_async_send	Starts an asynchronous message operation
pci_message_sync_send	Starts a synchronous message operation
pci_message_retrieve	Retrieves a message immediately
pci_message_async_retrieve	Starts an asynchronous retrieve message operation
pci_message_sync_retrieve	Starts a synchronous retrieve message operation
amcc_nvram_read	Reads a byte from NVRAM
amcc_nvram_write	Writes a byte from NVRAM
amcc_mailbox_read	Reads a AMCC mailbox
amcc_mailbox_write	Writes to a AMCC mailbox

现在，对我们要使用的硬件和软件资源都已经有了一定的了解，可以开锅造饭了。对 DSP 端的程序，读者可以根据自己的需要来编写，我直接采用 Chest Nut 先生的程序来做例子，大家使用时可别忘了饮水思源啊^%(&^$%%&(：_：

```
/*****************************************************/
/*              By chest nut,    Tsinghua            */
/*****************************************************/
#include "gather.h"

/*****************************************************/
/* Main() -- Main DSP side process                    */
/*****************************************************/
void InitDrv()
{
        // Driver Init;
        evm_init();
        pci_driver_init();
}
```

```
int main()
{
        unsigned int nMessage, i;
        unsigned int *pData;

        pData = (unsigned int *)SDRAM_ADDR;

        InitDrv();

        while (TRUE)
        {
                if (pci_message_sync_retrieve(&nMessage) == ERROR)
                        return FALSE;

                // Begin Process Data
                for (i = 0; i < DATA_LENGTH; i++)
                {
                        pData[i] = 1;
                }
                if (pci_message_send(nMessage) == ERROR)
                        return FALSE;

        }

        return(TRUE);
}
```

其中引用的头文件 gather.h 源文件如下：

```
/*   By chest nut       */
#ifndef _GATHER_H
#define _GATHER_H

#ifndef CHIP_6201
#define CHIP_6201       1
#endif

#include <stdio.h>
#include <stdlib.h>

/* Peripheral Support Include Files      */
#include <pci.h>

/* EVM/McEVM Driver Include Files        */
#include <board.h>
#include <intr.h>
#include <regs.h>

// DEFINES
#define UINT32  unsigned int
#define OK       0
#define ERROR   -1

#define DATA_LENGTH             1024
#define SDRAM_ADDR              0x02000000

#endif
```

程序中首先对使用的硬件进行初始化，接下来进行死循环，DSP 端一直读主机发送的消息，读到后，将 0x02000000 开始的 1024 个空间用 1 来填充。然后向主机发送消息，继续循环。同步和异步消息的收发有些不同，罗列如下：

#include <pci.h>

- int pci_message_send(unsigned int message);

此程序发送一个 32bit 的 message，如果它不能立即放入 mailbox，返回错误，message 未被发送。

Return:

returns OK or ERROR, 可能出错条件包括：外发信息邮箱不空或 HINT 未 clear

Note：

此函数检测邮箱 empty/full flags 和 HINT bit，如果邮箱 1 为空且 HINT is clear，信息被放入邮箱 1，并把 HINT bit 置 1。

- int pci_message_async_send(unsigned int message, int wait_for_ack，
  pci_msg_callback *p_callback)

此函数执行发送信息操作，立即返回，操作结束会调用 callback。

Parameters:

- message is the 32bit value to be sent to the host
- wait_for_ack determines when the operation is considered complete and the callback function is called.
- p_callback is the funciton pointer for the callback routine

Return:

- returns OK or ERROR,
  possible error conditions include: another send in progress

Notes:

- this function uses interrupts internally, it does not poll
- if wait_for_ack is TRUE then callback is not called until the message has been read by the host, if it is FALSE then callback is called as soon as the message is placed into the mailbox

- int pci_message_sync_send(unsigned int message, int wait_for_ack)

此函数发送 32bit message 到主机，操作完成后才返回。

Parameters:

- message is the 32bit value to be sent to the host
- wait_for_ack determines when the operation is considered complete
  and the callback function is called.

Return:

- returns OK or ERROR,
  possible error conditions include: another send in progress

Notes:

- this function implemented by calling pci_message_async_send() and
  waiting internally for the operation to complete.

- if wait_for_ack is TRUE then the operation is not complete until the
  message has been read by the host, if it is FALSE then the operation
  is complete as soon as the message is placed into the mailbox

- int pci_message_retrieve(unsigned int *p_message)

此函数接主机发送的信息，如果信息不存在，返回 ERROR。
Parameters:
- p_message is the location to store the 32bit value sent by the host
Return:
- returns OK or ERROR,
  possible error conditions include: incoming message mailbox not full
Notes:
- this function checks mailbox empty/full flags then if mailbox 1 is
  full the message is read from the mailbox 1 register

- int pci_message_async_retrieve(unsigned int *p_message,
  pci_msg_callback *p_callback)

此程序执行异步信息接收操作，立即返回。操作结束时调用 callback 函数。
Parameters:
- p_message is the location to store the 32bit value sent by the host
- p_callback is the funciton pointer for the callback routine
Return:
- returns OK or ERROR,
  possible error conditions include: another retrieve in progress
Notes:
- this function uses interrupts internally, it does not poll

- int pci_message_sync_retrieve(unsigned int *p_message)

此程序接收主机发送的 32bit message，直到完成操作后才返回。
Parameters:
- p_message is the location to store the 32bit value sent by the host
Return:
- returns OK or ERROR,
  possible error conditions include:another retrieve in progress
Notes:
- 此函数通过调用 pci_message_async_retrieve() 并 waiting internaly for the operation to complete

读者可以根据自己的实际情况选择同步或异步消息发送或接收。

对于主机端的程序，读者可以采用喜欢的方式来编程，但需要包括以下的内容：

```
#include <windows.h>
#include "evm6xdll.h"

void WriteWord2Mem(LPVOID hHpi,ULONG ulDataAddr,ULONG ulDataWord);

/*-------------------------------------------------------------*/
/* main()                                                      */
/*-------------------------------------------------------------*/

void main(int argc, char *argv[])
{
        // Board device handle
        HANDLE hBd = NULL;
        // Board index
        short  iBd = 0;
        // Exclusive open = TRUE
        BOOL   bExcl = 1;
        // Map selector = MAP1
        short  iMp = 1;
        // DSP boot mode
        EVM6XDLL_BOOT_MODE mode;
        // HPI interface handle
        LPVOID hHpi = NULL;

        // COFF file name
        char coffNam[] = "gather.out";

        // COFF load verbose mode = FALSE
        BOOL bVerbose = 0;
        // Clear bss mode = FALSE
        BOOL bClr = 0;
        // Dump mode = FALSE
        BOOL bDump = 0;

        /*-------------------------------------------------------*/
        /* Open a driver connection to a specific EVM6x board.*/
        /*-------------------------------------------------------*/
        hBd = evm6x_open( iBd, bExcl );
        if ( hBd == INVALID_HANDLE_VALUE ) exit(1);

        /*-------------------------------------------------------*/
        /* Cause a hardware reset on the target board.         */
        /*-------------------------------------------------------*/
        if ( !evm6x_reset_board(hBd) ) exit(2);

        /*-------------------------------------------------------*/
        /* Set board configuration.                            */
        /*-------------------------------------------------------*/
        if ( !evm6x_set_board_config(hBd, DSP_CLOCK_NORMAL,
                LITTLE_ENDIAN_MODE, 0XFF) )
                exit(0);
```

```
/*-----------------------------------------------------*/
/* Set the boot mode and cause a DSP reset.            */
/*-----------------------------------------------------*/
mode = iMp ? HPI_BOOT : HPI_BOOT_MAP0;
if ( !evm6x_reset_dsp(hBd,mode) ) exit(3);

/*-----------------------------------------------------*/
/* Establish a connection to the HPI of a target board.*/
/*-----------------------------------------------------*/
hHpi = evm6x_hpi_open(hBd);
if ( hHpi == NULL ) exit(4);

/*-----------------------------------------------------*/
/* Initialize EMIF registers                           */
/*-----------------------------------------------------*/
if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

/*-----------------------------------------------------*/
/* set Aux DMA priority higher than CPU                */
/*-----------------------------------------------------*/
/*  Due to the default priority of the auxiliary DMA
channel used for HPI accesses, the CPU can prevent
HPI accesses from completing for an indeterminate
amount of time.  This can occur when the CPU is very
active on the external memory interface, such as while
executing  code from external memory.  This condition

manifests itself as a hung PCI bus.  To prevent this
condition, the value 0x10 can be written to the DMA
Auxiliary Control Register of the 6201
(at address 0x01840070).  This elevates the priority
of the auxiliary DMA channel above all other DMA
channels and above the CPU.          */
/*-----------------------------------------------------*/
WriteWord2Mem(hHpi,0x01840070/*Addr*/,0x00000010/*Data*/);

/*-----------------------------------------------------*/
/* Read a COFF file and write the data to DSP memory.*/
/*-----------------------------------------------------*/
if (!evm6x_coff_load(hBd,hHpi,coffNam,bVerbose,bClr,bDump))
 exit(8);

/*-----------------------------------------------------*/
/* Release the DSP from the halted state               */
/*-----------------------------------------------------*/
if (!evm6x_unreset_dsp(hBd)) exit(10);

/* Here you can add you own code */


/*-----------------------------------------------------*/
/*Close the HPI session started with evm6x_hpi_open()*/
/*-----------------------------------------------------*/

if (!evm6x_hpi_close(hHpi)) exit(9);
```

```
        /*--------------------------------------------------------*/
        /* Close a previously opened driver connection to a board.*/
        /*--------------------------------------------------------*/
        if (!evm6x_close(hBd)) exit(16);

        exit(0);
}

/*-------------------------------------------------------------*/
/*  Write one word (32 bits) to DSP memory                     */
/*-------------------------------------------------------------*/
void WriteWord2Mem( LPVOID hHpi,ULONG ulDataAddr,ULONG ulDataWord )
{
        ULONG       ulLength;
        ULONG       ulReturnedLength;

        ulLength = 4;
        ulReturnedLength = ulLength;
        if ( !evm6x_hpi_write( hHpi, &ulDataWord,
                &ulReturnedLength, ulDataAddr) ) exit(6);

        if ( ulLength != ulReturnedLength ) exit(7);
}
```

主机作了初始化后，可以加载程序到 DSP 运行，将这时主机可以向 DSP 发送消息，并等待 DSP 返回消息，一切工作完成后，别忘了关闭 HPI 及 Evm 板。如果读者觉得这样主机（PC）有点浪费，可以单独开一个 Thread，来进行消息的收发。

下面是一个主机向 DSP 写数据并发送消息的子程序：

```
void CHostDlg::OnSenddata()
{
        ULONG ulLength = DATA_LEN * 4, i;
        EVM6XDLL_MESSAGE hMessage = 0x0000FFFF;
        char szData[10];

        for (i = 0; i < DATA_LEN; i++)
        {
                m_ulData[i] = i;
        }

        // Using Hpi to transfer data to dsp
        if (!evm6x_hpi_write(m_hHpi,
             (ULONG *)m_ulData, &ulLength, SDRAM_ADDR))
                        exit(0);

        if (ulLength != (DATA_LEN * 4))
                exit(0);

        // Indicate data transfer complete
        evm6x_send_message(m_hBd, &hMessage);
}
```

当然读者会使用自己的函数名称来实现这些功能，更不需要在 main()里实现这些功能了。用户加载自己的 DSP 程序时，只要将其中的文件名换成自己的文件名即可：

```
    // COFF file name
    char coffNam[] = "gather.out";
```

其他的初始化步骤搬过来用就行了。

好了，不再罗嗦了。六祖曰：“ 迷时师度，悟时自度 ”。大家要学会 “ 自度 ”，“ 遇事反求诸己 ”，这样才能快速提高。本人水平如此，也只能点到为止了。

题后话：

本来写到这里就觉得差不多了，又有人来问：“ 我的程序在 CCS 下运行正常，在主机端加载后怎么不正常？ ”，理由是往 DSP 的某一空间写某一确定数，主机来读取，得不到预期的结果。

我们来看一下程序是怎么写的（举此例，并无他意）：

```
if(!evm6x_coff_load(h_board,h_hpi,CoffName,bVerbose,bClr,bDump))
{
        MessageBox("Coff file Load error!","!Load Error",MB_OK);
        dspflag=7;
}
ULONG endflag=0;

while(!endflag)//读程序运行的结束标志,若结束，则endflag=1;
{
        if(!evm6x_hpi_read(h_hpi,&endflag,&ul_len,0x03000004)||(ul_len!=4))
        {
                MessageBox("Read error!","!Read Error",MB_OK);
                exit(8);
        }
}

ULONG mm[10];
if(!evm6x_hpi_read(h_hpi,mm,&ul_len,0x03000008)||(ul_len!=4*10))
{
        MessageBox("Read error!","!Read Error",MB_OK);
        exit(8);
}

evm6x_hpi_close(h_hpi);
evm6x_unreset_dsp(h_board);
evm6x_close(h_board);
```

我们首先需要弄明白，下面的函数到底是什么意思。

evm6x_unreset_dsp	Release the DSP from its reset state

具体一点：

> The evm6x_unreset_dsp( ) function releases the DSP from the halted state invoked by the evm6x_reset_dsp( ) function with the *mode* parameter set to HPI boot (see page 2-40). Use this function in conjunction with an evm6x_reset_dsp( ) call only. The return value indicates the success of the function call.
>
> — The *h_device* parameter is the handle returned from a successful evm6x_open( ) call.
>
> The function returns TRUE or FALSE to indicate the success of the operation.
>
> In the following example, the evm6x_unreset_dsp( ) function releases the DSP from the halted state that results from resetting the DSP into HPI boot mode.

那么，“ halted state ”和“ releases the DSP from the halted state ”到底怎么理解呢？

我们从主机往 DSP 载入程序后，DSP 并不自动运行程序，需要一个启动信号，把它从“ halted ”状态唤醒，evm6x_unreset_dsp()这个函数完成的就是这个工作，那么它在程序中的位置也就清楚了，“ 米放到锅里，过了好久了，怎么还没熟？ ”，“ 哦，没有开电源！————嘻 ”。到此，上面的问题也就不言自明了。

本来想强调一下，又觉得没有这个必要了。大家都是明白人。