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。

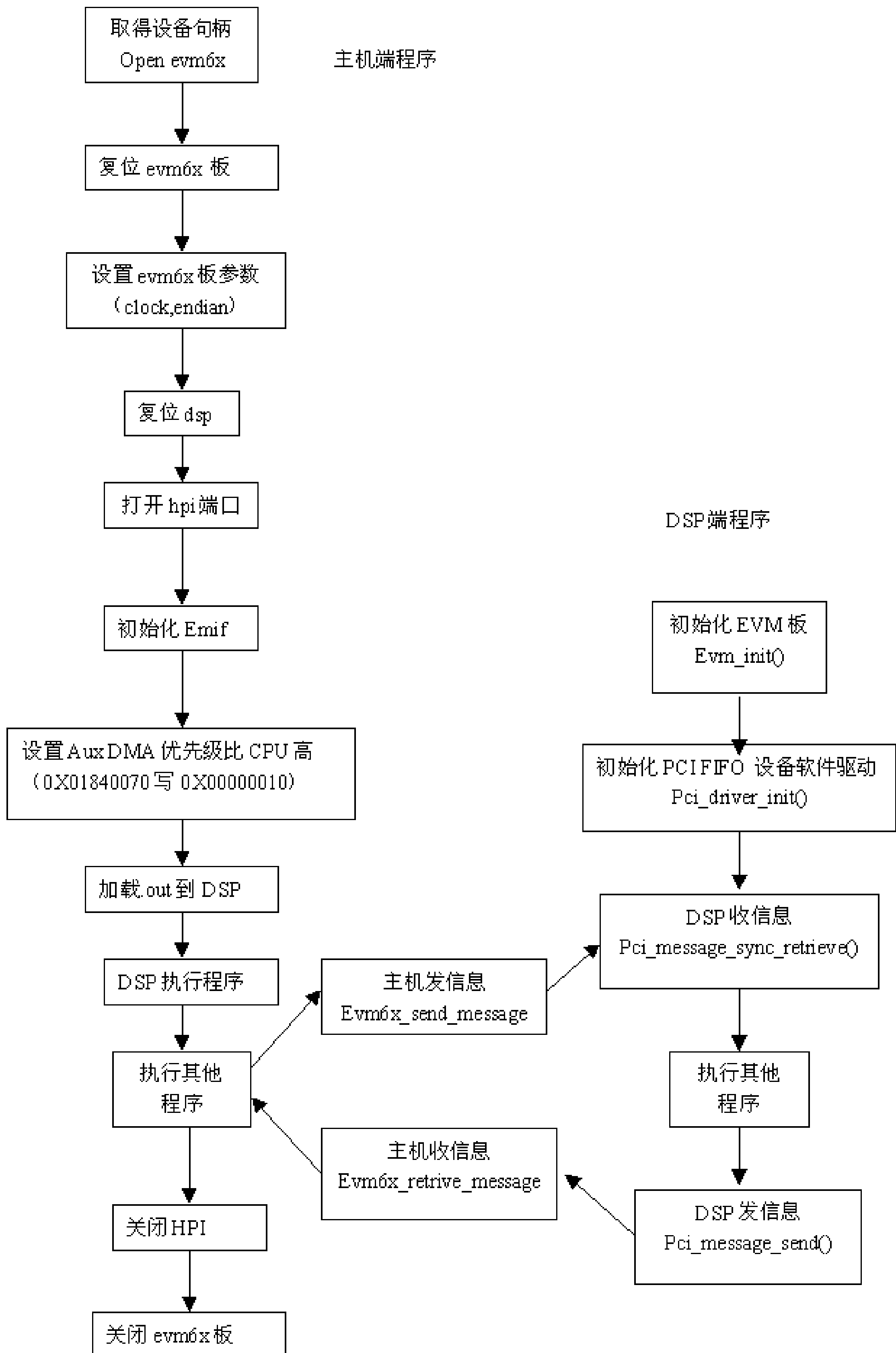
下面用闻亭公司的 C6xP 板硬件和闻亭公司的 PCI 仿真器为例，来实现 **Host** 和 **DSP** 通信。对于‘C6Xpa 板同样有效。

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_re-lease	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();
}
```

```c
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：

```c
/*   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, OXFF) )
                exit(0);
```

```
/*---------------------------------------------------*/
/* Set the boot mode and cause a DSP reset.         */
/*---------------------------------------------------*/
mode = iMp ? HPI_BOOT : HPI_BOOT_MAPO;
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，“ hal ted 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。